# Веб-компоненты в действии

Веб-компоненты в действии

Бен Фаррелл

MANNING

ДМК ПРЕСС

Бен Фаррелл

# Веб-компоненты в действии

# *Web Components in Action*

**BEN FARRELL**
**Foreword by Gray Norton**

MANNING
Shelter Island

# *Веб-компоненты в действии*

**БЕН ФАРРЕЛЛ**
**Предисловие Грея Нортона**

ДМК Пресс
издательство
Москва, 2020

**УДК 004.42**
**ББК 32.972**
**Ф25**

**Фаррелл Б.**
Ф25 Веб-компоненты в действии / пер. с англ. Д. А. Беликова. – М.: ДМК Пресс, 2020. – 462 с.: ил.

**ISBN 978-5-97060-856-2**

Один из основных факторов, способствующих трансформации интернета в последние годы, – широкое внедрение разработки пользовательского интерфейса на основе компонентов. В этой книге подробно описываются рабочие процессы, которые дают вам полный контроль над стилями и поведением компонентов и существенно упрощают их создание, совместное и повторное использование в проектах.

В первой части рассмотрено получение простого компонента с нуля. Вторая часть посвящена улучшению организации проекта. В третьей части освещаются принципы совместной работы с несколькими компонентами, позволяющей решать более сложные задачи. Для всех примеров предоставляется исходный код.

Издание предназначено для веб-разработчиков, имеющих опыт работы с HTML, CSS и JavaScript.

УДК 004.42
ББК 32.972



ISBN 978-1-617-29577-5 (англ.)
ISBN 978-5-97060-856-2 (рус.)

*Моей удивительной жене,*
*которая пишет гораздо более захватывающие книги,*
*по сравнению с книгами о веб-разработке,*
*посвященные драконам и катастрофам*

# Оглавление

# Содержание

# Предисловие

Интернет прошел долгий путь. То, что началось три десятилетия назад как относительно простое средство публикации, совместного использования, обнаружения и потребления контента, превратилось в мощную и гибкую платформу приложений, поддерживающую невероятное количество вариантов использования. Между тем сфера его присутствия расширилась, и теперь выход в интернет осуществляется не только с настольных компьютеров, но и с устройств всех типов.

В результате этого постепенного преобразования мы, веб-разработчики, преследуем постоянно меняющуюся цель. Сегодняшние веб-сайты на несколько порядков сложнее по сравнению с их ранними предшественниками, и ожидания, связанные с пользовательским интерфейсом, значительно выросли.

К счастью, наш инструментарий также не стоял на месте. Сама веб-платформа приобрела сотни новых возможностей, а последующие поколения библиотек, фреймворков и инструментов постоянно совершенствуют современный уровень развития технологий, помогая нам удовлетворять растущие требования.

Одним из основных факторов, способствующих трансформации интернета в последние годы, стало широкое внедрение разработки пользовательского интерфейса на основе компонентов. Разделение нашей работы на компоненты, каждый из которых отвечает за структуру, стиль и поведение части пользовательского интерфейса, помогло нам управлять сложностью и создавать более сложные сайты.

Компоненты можно повторно использовать в каком-либо проекте или совместно в разных проектах, что повышает нашу эффективность. Дизайн-системы можно выразить в виде наборов готовых к использованию компонентов, обеспечивающих согласованность, которые позволяют командам сосредоточиться на конкретных потребностях продукта.

Популярные фреймворки помогли осуществить революцию компонентов, и сегодня большинство компонентов специфичны для конкретного фреймворка или библиотеки. Но параллельно предпринимались многолетние усилия по созданию мощной, нативной модели компонентов для веб-платформы.

Веб-компоненты – это общий термин для нового семейства функций веб-платформ, предлагающих прямую поддержку разработки на основе компонентов. Пользовательские элементы позволяют расширять словарь HTML, определяя собственные теги, которые легко работают со встроенными в браузер тегами и могут использоваться в одних и тех же местах, независимо от фреймворка. Технология Shadow DOM позволяет вам применять инкапсуляцию в нативном стиле, гарантируя, что CSS-правила компонента не будут непреднамеренно нарушать – и не будут нарушаться – форматирование страницы.

Вам, наверное, интересно, какие преимущества дают веб-компоненты по сравнению с компонентными моделями. С одной стороны, веб-компоненты обещают повысить совместимость, упрощая обмен компонентами даже между комплектами технологий. Модель общих компонентов также снижает риск блокировки, позволяя вам выполнять больше работы по мере изменения набора инструментов с течением времени.

Книга, которую вы сейчас держите в руках, исключительно своевременна. Путь к стандартизации и поддержка веб-компонентов претерпевали взлеты и падения, но я рад сообщить, что цель уже видна: все, кроме одного из популярных браузеров, уже поддерживают веб-компоненты, а когда состоится официальный релиз следующей версии Microsoft Edge, головоломка будет завершена.

Пользовательские элементы, Shadow DOM и другие функции веб-компонентов по своей природе являются низкоуровневыми примитивами. Некоторые разработчики будут использовать эти функции только косвенно, поскольку поддержка веб-компонентов во фреймворках увеличилась с ростом поддержки браузеров. Многие из самых популярных фреймворков теперь облегчают разработку и совместное использование веб-компонентов, и стал появляться целый новый класс инструментов, ориентированных на веб-компоненты.

Но вы также можете использовать функции веб-компонентов напрямую, по отдельности или сочетая их. Читая эту книгу, вы подробно изучите каждую функцию и то, как они связаны друг с другом, что даст вам возможность сделать правильный выбор для себя и своей команды.

Бен Фаррелл использовал веб-компоненты с момента их возникновения в самых разных приложениях. В ходе своей работы он накопил огромное количество ценных знаний и обнаружил множество эффективных шаблонов, которыми он поделится с вами на этих страницах.

Бен приводит примеры, демонстрируя разные концепции с помощью убедительных проектов, которые освещают реалистичные варианты использования. Вы, конечно, многому научитесь, но также непременно найдете здесь идеи и код, которые можно применить непосредственно в своих собственных проектах.

Решив заняться веб-компонентами и взяв эту книгу, вы сделали хороший выбор. Наслаждайтесь этим путешествием!

*Грей Нортон*,
технический руководитель /
менеджер проекта Polymer, GOOGLE

# От автора

Для меня знакомство с веб-компонентами началось в 2013 году. Я помню, что работал над забавным небольшим проектом с использованием Angular версии 1 и изучал некоторые аспекты управления CSS и классами, которые Angular в то время плохо обрабатывал. Я знал, что мог бы легко сделать то, что мне нужно, в простом HTML, CSS и JavaScript, но Angular затруднял это только потому, что то, что я делал, находилось за пределами проторенных троп.

Примерно в это же время я почувствовал, что действительно начинаю овладевать Angular, поэтому написал в блоге несколько постов о некоторых интересных, нетипичных подходах. Тогда же волнение по поводу Angular стало угасать, и только начиналось волнение по поводу React.

Честно говоря, я был разочарован. Я долго смотрел на цикл, в котором чувствовал себя пойманным в ловушку. В течение всего двух или трех лет я постоянно учился и получал хорошие знания по фреймворкам JS. Ни один из этих фреймворков не был совместим друг с другом. Я дошел до того, что почувствовал, что действительно могу сосредоточиться на своем проекте без фреймворка на заднем плане, но затем неожиданно появилось нечто новое, что заставило меня почувствовать, что мне нужно вернуться на круги своя.

В то же самое время Google была выпущена библиотека Polymer как очень ранняя и нестабильная версия. Создание отдельных компонентов, которые могли бы существовать где угодно, звучало как удивительное обещание. Первоначально мне нравилось то, чего она пыталась достичь, но API, предшествующий первой версии, который постоянно менялся, и тот факт, что я заменял свой рабочий процесс еще одним фреймворком, заставил меня все переосмыслить. Я начал изучать предлагаемые веб-стандарты, которые сделали возможным создание библиотеки Polymer, и увидел огромный потенциал. Я понял, что это была не библиотека Polymer, которой я восхищался, – в действительности это были веб-компоненты.

Я начал вести блог и дискуссии о веб-компонентах. Примерно в это же время присоединился к Adobe. Это было важно, потому что моя команда работала над небольшими прототипами с одним, может быть, двумя разработчиками проекта. Это означало, что я мог экспериментировать с технологией и инструментами по своему выбору. Почти для каждого проекта я продолжал продвигать веб-компоненты, экспериментируя и постоянно улучшая рабочий процесс для работы с ними.

Конечно, это было непросто. Иногда я полностью был лишен почвы под ногами! Поскольку веб-компоненты стали стандартом, которым они являются сегодня, мы увидели, что изменение API и функции стали устаревшими, но у меня не было выбора, потому что мне действительно нравится работать как можно ближе к браузеру, используя только

HTML, JS и CSS, и я рассматривал веб-компоненты как средство обеспечения структуры своих проектов, а не для того, чтобы они превратились в спагетти-код.

Я еще не был полностью убежден в жизнеспособности веб-компонентов. С одной стороны, я пока не использовал Shadow DOM. Я не хотел увлекаться чем-то, что поддерживала только Google, у которой была сомнительная поддержка полифилов. Но затем веб-компоненты появились в браузере Safari, и Mozilla также пообещал, что будет поддерживать их. Вишенкой на торте стал момент, когда браузеры начали поддерживать модули JS и импорт нативно, и я смог правильно разделить код и, что более важно, HTML и CSS. Когда все это произошло, я знал, что веб-компоненты начинают реализовывать свой потенциал.

Конечно, все происходило очень медленно в течение нескольких лет. Многие разработчики, которые изначально были в восторге от веб-компонентов, потеряли терпение, и я не виню их. Сначала я обратился к издательству Manning по поводу книги о веб-компонентах – до того, как произошли некоторые важные ключевые события, например когда крупные компании-разработчики популярных браузеров объединились, чтобы завершить версию спецификации номер 1. В то время Manning не было уверено, особенно из-за того, что книги в этой области не публиковались, поскольку было неизвестно, чем все это закончится.

Был ли я настроен слишком оптимистично или просто провел с ними достаточно времени, чтобы узнать потенциал веб-компонентов, но издательство связалось со мной через год, чтобы сделать еще одно предложение. Даже тогда, в начале 2018 года, дело все равно могло бы принять дурной оборот, если бы другие компании-разработчики браузеров решили пойти на попятную. Кроме того, в то время я не подходил к разработке веб-компонентов так, как это делало большинство разработчиков, используя импорт HTML в качестве отправной точки. Тем не менее на протяжении этой книги класс LitElement от команды Polymer начал действовать в очень похожей со мной манере, используя шаблонные литералы для хранения разметки и стиля. Это, в сочетании с поддержкой веб-компонентов, когда над ними работала и компания Microsoft, осенью 2018 года позволило мне вздохнуть с облегчением, зная, что подходы, описанные в моей книге, идут в ногу с настоящим и будущим веб-компонентов. Я определенно продолжу совершенствовать свой рабочий процесс, по мере того как новые функции появляются в браузере и придумываются сообществом, но я очень рад нынешнему положению веб-компонентов, поскольку моя книга скоро будет опубликована. И конечно же, мне не терпится поделиться всем этим с читателями!

# *Благодарности*

Данная книга была бы невозможна без всех тех удивительных людей, которые помогали мне на протяжении этого пути. Я хочу поблагодарить своих друзей из Северной Каролины и замечательных людей, которые проводят и посещают конференцию NCDevCon, за то, что они почти постоянно слушали мои доклады о веб-компонентах в Yammer. В частности, я хотел бы поблагодарить Эдриана Помилио за то, что он поразил меня своим выступлением в 2011 году, в котором были показаны пользовательские элементы, прежде чем они стали чем-то особенным.

Я также хотел бы поблагодарить членов команды GE Design System за то, что они были моими «сообщниками» в этом деле, в то время когда веб-компоненты были абсолютным новшеством, и мы были уверены, что все остальные считают нас безумцами. В частности, я хотел бы поблагодарить Мартина Рэгга, Джеффа Райхенберга и Джона Роджерсона за то, что они копались со мной в деталях при написании этой книги о новом способе создания сайтов. Еще хотел бы поблагодарить команду Google Polymer за помощь и руководство в течение этого времени, а также их технического руководителя Грея Нортона за написание предисловия к книге.

В Adobe я хотел бы поблагодарить всю команду Adobe Design (и за ее пределами) за поддержку и искреннюю радость по поводу публикации моей первой книги.

Конечно, моя жена Ребекка Гомес Фаррелл не только поддерживала меня, но и сама оказалась замечательным писателем и редактором. Помимо того что она принесла мне крепкий напиток, когда он мне понадобился, она помогла новому писателю стать лучше, давая стоящие, профессиональные советы.

Я хотел бы поблагодарить редакционную команду издательства Manning, в которую входят редакторы-консультанты по аудитории Кристен Уоттерсон, Кевин Харрелд и Ребекка Райнхарт, а также редактор-консультант по технической аудитории Дуглас Дункан, технический корректор Мэтью Уэлк, редактор по производству Энтони Калькара, редактор Ребекка Деуэль-Гальегос и корректор текста Тиффани Тейлор. Наконец, я хотел бы поблагодарить рецензентов, чьи отзывы и понимание сыграли важную роль в формировании этой книги, в том числе Альберто Чарланти, Алисию Бейкер, Бирну Себарте, Клайва Харбера, Дэниела Купера, Эрнана Гарсия, Джеймса Карелла, Джона Ларсена, Хуана Асенсио, Джастина Каллеха, Оливера Ковача, Пьетро Маффи, Рональда Бормана, Рассела Доуна Кахолес, Райана Барроуз, Серхио Арбео, Стефана Троста, Томаса Оверби Хансена, Тимоти Р. Кейна и Кумара С. Унникришнана (TR Technology & Ops).

# Об этой книге

Книга «*Веб-компоненты в действии*» не диктует, какие подходы должны использовать разработчики. Вместо того чтобы рассказывать читателям, что делать, я использую более исследовательский подход, чтобы охватить основы веб-компонентов. Вы должны признать, что хотя эксперты могут сказать вам, что такое хороший рабочий процесс на сегодняшний день, захватывающий момент касательно стандартов состоит в том, что их можно создавать таким образом, которого никто не ожидает.

В этой книге я стремлюсь дать вам отличные идеи и рабочие процессы для начала работы. Я также надеюсь дать вам знания для дальнейшего использования веб-компонентов, способами, которые я еще не рассматривал, и для проектов, с которыми я не сталкивался.

## Кому следует прочитать эту книгу

Данная книга предназначена для веб-разработчиков, которые интересуются веб-компонентами и хотят узнать больше о стоящих за ними стандартах и о том, как они объединяются с другими веб-технологиями для создания автономных компонентов или приложений.

Она также подходит для разработчиков, которым нужны идеи о том, как освободиться от сложных фреймворков или библиотек и вернуться к написанию простого HTML, JS и CSS без каких-либо шагов сборки.

## Как организована эта книга: дорожная карта

Эта книга состоит из трех частей, охватывающих 15 глав и приложение.

В первой части приводятся первые шаги, описывающие получение простого компонента с нуля:

- в первой главе описывается, что имеется в виду, когда речь идет о веб-компонентах и различных стандартах, которые объединяются для их создания;
- вторая глава рассказывает о создании самого первого веб-компонента, а также знакомит вас с концепциями минимума, необходимыми для создания чего-то полезного;
- третья глава выводит минимальный компонент на следующий уровень, делая его многоразовым;
- в четвертой главе подробно описывается API веб-компонентов и жизненный цикл в сравнении их с другими API и жизненными циклами, с которыми вы, возможно, сталкивались;
- в пятой главе вы познакомитесь с модулями для более подходящего повторного использования кода и организации проекта.

Вторая часть основана на минимальном компоненте и охватывает концепции улучшения рабочего процесса разработчика и организации проекта:

- шестая глава подробно описывает использование модулей для разделения и импорта логики представления, такой как HTML и CSS, для лучшей организации вашего компонента;
- седьмая глава посвящена альтернативному, но не предпочтительному способу организации вашего компонента с помощью HTML-импорта, разбивая его на части, которые также относятся к другим аспектам веб-компонентов;
- восьмая глава знакомит вас с технологией Shadow DOM и рассказывает о ее пользе для защиты и инкапсуляции вашего компонента;
- в девятой главе мы продолжаем изучение Shadow DOM, чтобы охватить его CSS-аспекты;
- в десятой главе исследуются проблемы, которые могут возникнуть у разработчиков веб-компонентов с CSS в Shadow DOM, и способы избежать их или преодолеть.

Третья и последняя часть посвящена совместной работе с несколькими компонентами, чтобы создать нечто большее:

- в одиннадцатой главе рассматриваются освещенные ранее концепции, которые используются для создания нового, более отточенного компонента, основанного на уже созданных дочерних компонентах;
- в двенадцатой главе мы продвигаем этот абсолютно новый компонент, чтобы быть более готовым к промышленной эксплуатации благодаря применению инструментов сборки, которые позволяют использовать его в старых браузерах, не поддерживающих веб-компоненты;
- в тринадцатой главе мы дополняем этот компонент, написав для него тесты, которые выполняются в трех разных контекстах, чтобы изучить различные варианты, доступные для разработчиков веб-компонентов;
- в четырнадцатой главе обсуждается передача сообщений между вашими компонентами и подробно рассматривается распространенный шаблон проектирования;
- в пятнадцатой главе рассказывается о будущем веб-компонентов, а также о возможностях, которые они могут предоставить сегодня, скрывая сложность и делая все, от видеоэффектов в реальном времени до смешанной реальности, более простым в использовании.

Наконец, в приложении рассказывается о новых функциях Java Script (ES6/ES2015) и о том, как они помогают веб-компонентам.

## О коде

Исходный код предоставляется для всех примеров в этой книге и доступен для скачивания с веб-сайта издательства Manning по адресу www.manning.com/books/web-components-in-action и из репозитория GitHub,

который можно найти по адресу https://github.com/bengfarrell/webcomponentsinaction. Репозиторий организован в папки для каждой главы, и в них обычно находятся вложенные папки для каждого раздела. Исключения составляют случаи работы над большим примером, охватывающим всю главу.

Код можно выполнить только с помощью браузера, и его не нужно компилировать до тех пор, пока вы не дойдете до следующих глав, посвященных инструментам сборки. Как правило, для запуска соответствующего HTML-файла, который служит примером, понадобится простой HTTP-сервер, но только для того, чтобы решить проблемы, связанные с разными источниками.

В этой книге содержится множество примеров исходного кода, как в виде пронумерованных листингов, так и встроенных в обычный текст. В обоих случаях исходный код форматируется `шрифтом фиксированной ширины`, `подобным этому`, чтобы отделить его от обычного текста. Иногда код также выделяется **`жирным шрифтом`**, чтобы выделить код, который изменился по сравнению с предыдущими шагами в этой главе, например когда к существующей строке кода добавляется новая функция.

Во многих случаях оригинальный исходный код переформатируется; мы добавили разрывы строк и переработали отступы, чтобы обеспечить доступное пространство страницы в книге. В редких случаях даже этого было недостаточно, и списки содержат маркеры продолжения строки (➥). Кроме того, комментарии в исходном коде часто удаляются из листингов, когда описание кода есть в тексте. Аннотации к коду сопровождают многие листинги, выделяя важные понятия.

## Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

## Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.дмк.рф на странице с описанием соответствующей книги.

## Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

## Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Manning очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу dmkpress@gmail.com со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

# *Об авторе*

Бен Фаррелл – опытный старший разработчик в компании Adobe, работающий в команде разработчиков прототипов Adobe Design. Бен вместе со своей командой помогает формировать и реализовывать пользовательский опыт продуктов и функций на промежуточном уровне между проектированием и разработкой. Всю свою карьеру он занимался веб-технологиями, но также работал над отмеченными наградами проектами с использованием самых разных платформ и языков.

# *Об иллюстрации на обложке*

Рисунок на обложке книги «Веб-компоненты в действии» озаглавлен как *Bourgeois de Londre* (лондонский буржуа). Эта иллюстрация взята из коллекции костюмов разных стран Жака Грассе де Сен-Совёра (1757–1810) под названием *Costumes Civils Actuels de Tous le Peuples Connus*, опубликованной во Франции в 1788 году. Каждая иллюстрация прекрасно нарисована и раскрашена вручную. Богатое разнообразие коллекции Грассе де Сен-Совёра ярко напоминает нам о том, как культурно были отделены города и регионы мира всего 200 лет назад. Изолированные друг от друга люди говорили на разных диалектах и языках. На улицах или в сельской местности было легко определить, где они жили и какова была их профессия или положение в жизни, по их одежде.

С тех пор наш стиль одежды изменился, и разнообразие по регионам, столь богатое в то время, исчезло. Сейчас трудно отличить жителей разных континентов, не говоря уже о разных городах, регионах или странах. Возможно, мы обменяли культурное разнообразие на более разнообразную личную – определенно, на более разнообразную и динамичную технологическую жизнь.

В то время когда трудно отличить одну компьютерную книгу от другой, издательство Manning празднует изобретательность и инициативу компьютерного бизнеса с помощью обложек книг, основанных на богатом разнообразии региональной жизни двух веков назад, возвращенной к жизни рисунками Грассе де Сен-Совёра.

# Часть I

# Первые шаги

В последнее время вы, наверное, все чаще и чаще слышали о веб-компонентах. Во многом это связано со всеми популярными, современными браузерами, которые стали поддерживать их в последние месяцы. Это относится и к Microsoft Edge, поскольку вы уже можете скачать предварительный просмотр для разработчиков, пока мы ожидаем официального релиза с поддержкой Chromium. Впрочем, это может немного сбить с толку, если вы посмотрите глубже, чтобы увидеть, что такое веб-компоненты!

Мало того что набор стандартов, составляющих веб-компоненты, немного изменился с течением времени, в действительности веб-компонент можно создать с помощью пользовательских элементов как таковых! Вы можете создать собственный элемент, который будет находиться на вашей HTML-странице, как и любой другой элемент, предоставляемый браузером. Что еще более важно, с помощью API пользовательских элементов ваш элемент может получить специальную логику, чтобы стать полнофункциональным, крошечным интерактивным компонентом, который выглядит простым снаружи и может работать вместе с любым другим элементом на странице.

В первой части этой книги мы расскажем, как создать свои первые пользовательские элементы, а также о некоторых передовых практиках, касающихся них. В конце первой части, даже просто исследуя эту единственную концепцию, вы создадите веб-компоненты, которые действительно полезны в реальных ситуациях, позволяя им быть обернутыми как единый фрагмент, управляя своими собственными зависимостями, возможно, включая другие вложенные веб-компоненты, готовые для размещения на HTML-странице.

# 1

# Фреймворк без фреймворка

***Эта глава охватывает следующие темы:***

- что такое веб-компонент;
- теневая модель DOM;
- пользовательские элементы;
- библиотеки Polymer и X-Tag;
- функции ES6/ES2015.

Привет, и спасибо за то, что читаете эту книгу! Вот уже несколько лет я использую веб-компоненты практически во всех проектах веб-разработки, которые у меня были.

Будучи веб-разработчиками, наша работа заключается в выборе правильных инструментов для любого конкретного проекта. Это может стать сложной задачей, потому что важны не только насущные потребности проекта. Потребности вашей команды также важны, а еще важен вопрос о том, является ли проект частью более обширной экосистемы в вашей компании, как он будет поддерживаться и как *долго* его нужно поддерживать. Список можно продолжить.

Конечно, эти решения не являются уникальными для веб-разработчиков, но одним из основных отличий между нами и многими разра-

ботчиками программного обеспечения является то, что веб-сообщество выпустило изумительное количество инструментов, библиотек и фреймворков. Может быть, трудно не отставать от них всех – настолько, что уже давно обсуждается «усталость от фреймворков».

Принятие этих новых инструментов, кажется, происходит с молниеносной скоростью. Отложив на мгновение фреймворки, даже что-то такое же нишевое, как и исполнители задач для создания JavaScript-проектов, за последние несколько лет кардинально изменилось. Я был свидетелем перехода от Grunt в 2012 году к Gulp всего пару лет спустя, и теперь наблюдается тенденция действовать по минимуму при использовании Node.js NPM (Node Package Manager) для запуска сценариев сборки. Говоря о менеджерах пакетов, мы, разработчики, колебались, делая выбор между NPM, Bower и Yarn, для запуска наших зависимостей пользовательского интерфейса.

Инструменты сборки и менеджеры пакетов – это одно. Это небольшие, но важные части нашего процесса веб-разработки. Тем не менее такая же ситуация происходит с тем, как мы на самом деле создаем свои приложения и пользовательский интерфейс, что, возможно, является самой главной и важной частью веб-разработки.

Определенным разработчикам может быть трудно поспеть за этим, хотя изучать новый фреймворк или библиотеку интересно. Некоторые из них имеют более крутую кривую обучения, чем другие, и во многих случаях вы изучаете «систему» фреймворка, а не фундаментальные концепции HTML, JS и CSS.

Если речь идет о разработчике в команде или компании, тут есть дополнительные проблемы. В начале проекта вам нужно будет договориться о том, какие инструменты вы будете использовать для разработки в течение жизненного цикла проекта. Это включает в себя инструменты сборки, тестирования и, конечно же, любые фреймворки или библиотеки. Не каждый согласится с лучшим вариантом. Если команда большая и работает над большим количеством проектов, может быть заманчиво позволить разработчикам выбирать свои собственные инструменты для каждого проекта. В конце концов, полезно проанализировать потребности проекта и использовать соответствующие инструменты. Но таким образом мы также игнорируем неизбежное, когда разработчики должны работать сообща, чтобы создать общие части пользовательского интерфейса или интегрировать недавно принятую дизайн-систему, обязательную для всей компании. В конце концов, использование различных инструментов и фреймворков может выйти вашей команде боком.

Если все соглашаются, охотно или нет, использовать один и тот же фреймворк, какое-то время все может быть замечательно. Даже тогда, спустя два или три года, фреймворки могут устаревать. Использование старых технологий начинает сказываться, особенно для начинающих разработчиков в вашей команде, которые хотят, чтобы их навыки соответствовали всему остальному веб-сообществу. В этот момент ваша организация сталкивается с выбором: переделать весь комплекс тех-

нологий, используя новый фреймворк, или оставить старый и столкнуться с ощущением, что это не является инновационным местом для работы.

Это несомненно сложная проблема и решение! Конечно, возникает вопрос: «Какова альтернатива?» Я говорил с довольно многими людьми, которые хотят освободиться от этой постоянной мешанины с фреймворками по ряду причин. «Почему нельзя просто использовать простой HTML, JS и CSS?» Это распространенный вопрос. Одним из главных преимуществ отказа от фреймворка является возможность сосредоточиться на основных концепциях веб-разработки, а не на изучении специфических для фреймворка навыков, которые можно или нельзя перенести в следующий популярный фреймворк. Еще одним огромным преимуществом является возможность опробовать небольшие библиотеки и микрофреймворки, которые решают конкретные задачи в вашем проекте. Барьер входа в эти и даже новые инструменты сборки пользовательского интерфейса намного ниже, учитывая, что вы не боретесь с конкретной средой разработки, предоставляемой последним популярным фреймворком.

Современные фреймворки чрезвычайно полезны и решают большие проблемы, но почему мы больше не слышим об использовании так называемого «чистого JavaScript», учитывая желание разработчиков попробовать что-то другое? В некоторой степени слышим. Рассмотрим результаты опроса, проведенного State of JavaScript в 2017 году: https://2017.stateofjs.com/2017/front-end/results/. Вы увидите, что разработка без применения фреймворков занимает второе место по популярности, уступая только React.

Тем не менее мы не знаем, почему люди утверждают, что предпочитают не использовать фреймворк, а чистый JS. Что создают эти разработчики? Какие инструменты/процессы они используют? Мне было бы любопытно узнать, создают ли они что-то вроде фреймворка для решения проблемы отсутствия структуры и организации кода, что обычно предоставляют современные фреймворки.

Этот последний момент, касающийся структуры и организации кода, заключается в том, что веб-разработка без фреймворка была для меня незапланированной в прошлом, и именно поэтому я всегда обращался к последним фреймворкам. Без структуры ваш код превращается в спагетти. Сохранение и написание новых функций может стать безумием без предсказуемой организации проекта. Тем не менее мне хотелось освободиться от больших, полномасштабных фреймворков; когда я впервые увидел веб-компоненты, я увидел огромную возможность, позволяющую сделать именно это.

Итак… как это сделать? Чтобы действительно решить этот вопрос, нужно понять, что такое веб-компоненты. Прежде чем углубиться в детали, мы будем использовать календарь с возможностью выбора даты в качестве примера, с которым мы все, вероятно, сталкивались. Хотя это и не веб-компонент, по сути, это аналогичная концепция, если заглянуть внутрь.

## 1.1 Что такое веб-компоненты?

Современные популярные фреймворки сегодня в основном предлагают возможность многократного использования кода в виде *компонентов*, или *модулей*. В целом это совместно используемые и автономные фрагменты кода (HTML/JS/CSS), которые предлагают визуальный стиль и интерактивность и, возможно, имеют API или параметры, которые можно настраивать.

Подумайте о том, что уже есть в вашем браузере. И учтите, что у нас уже есть многократно используемые модульные элементы, которые предлагают стиль и интерактивность и поставляются с API.

Конечно, я говорю о тегах HTML, или элементах DOM. Они отображаются в DOM и имеют определенный тип функциональности. Теги `<div>` или `<span>` достаточно универсальны и используются для хранения текста или смеси элементов. Элементы `<button>` или `<input>` более специфичны по функциональности и стилю. Когда вы помещаете кнопку в свой HTML-код, она выглядит как стандартная кнопка, а когда вы нажимаете на нее, она ведет себя как кнопка. Это похоже на различные стили `<input>`, независимо от того, хотите ли вы создать календарь с возможностью выбора даты, ползунок или поле ввода текста.

### 1.1.1 Календарь с возможностью выбора даты

Возьмем, к примеру, календарь с возможностью выбора даты. Чтобы создать его, просто поместите приведенный ниже тег в HTML-код:

```
<input type="date">
```

Выглядит легко, не так ли? Так и есть! То, что вы на самом деле получаете из этого простого тега, довольно сложно, но все это обрабатывается вашим браузером. Этот тег (при использовании типа `"date"`) предлагает поле для ввода текста, и вы можете щелкнуть по месяцу, дню или году и перейти вверх или вниз. Кроме того, если щелкнуть по стрелке, указывающей вниз, откроется представление календаря, с которым пользователь может взаимодействовать, чтобы выбрать дату, как показано на рис. 1.1. Следует отметить, что на мобильном телефоне он действует немного иначе. Он не будет открываться, как это происходит в настольном браузере. Вместо этого появится модальное окно.

Более того, этот календарь имеет *свойства*, которые можно запросить, включив значение. В этом можно убедиться, зарегистрировав свойство в консоли JS:

```
console.log( document.querySelector('input').value );
```

Когда я регистрирую его, то вижу текущее значение календаря в своей консоли. Он также отправляет *события*, которые я могу прослушать, когда значение изменяется или отправляется. Я также могу вызывать *методы* для календаря для перехода по датам.

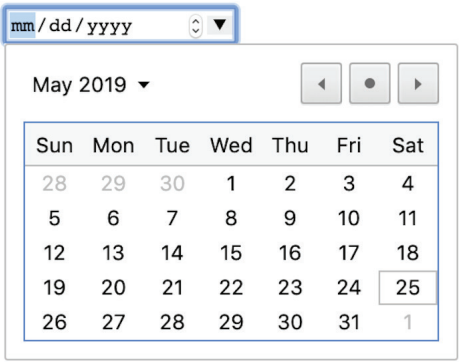



**Рис. 1.1 Расширенный пользовательский интерфейс календаря с возможностью выбора даты**

Календарь с возможностью выбора даты является отличным примером повторно используемых компонентов или модулей с довольно сложным визуальным стилем и шаблонами взаимодействия, которые должны программироваться компаниями-разработчиками браузеров. Они работают в разных ситуациях. Также этот календарь представляет собой отличный пример популярной концепции веб-компонентов под названием Shadow DOM (Теневая модель DOM).

## 1.1.2 Теневая модель DOM

Теневая модель DOM – это способ изолировать ваш веб-компонент и предотвратить непреднамеренные последствия более крупного приложения. Когда вы откроете инструменты разработчика для просмотра DOM, то просто увидите тег `<input type="date">`. Однако если вы используете браузер Chrome и активируете пункт «Показывать теневую модель DOM агента пользователя» в настройках инструментов разработчика, тот же самый тег `<input>` приобретает иной вид и выглядит, как показано на рис. 1.2.

Под `#shadow-root` намного больше разметки! Лично первое, на что я обращаю внимание при изучении этого кода, – всплывающее окно календаря. Хотя было бы замечательно увидеть этот фрагмент в HTML и CSS, его там нет, потому что этот фрагмент пользовательского интерфейса является частью вашей собственной ОС, которую ваш браузер просто показывает посредством элемента.

Тем не менее у нас есть значительное количество элементов, скрытых в нашей «теневой» DOM, которые все появляются в элементе поля ввода.

Если присмотреться, можно заметить, что в нашей теневой модели DOM размещены теги `<div>` и `<span>`. Вам может прийти в голову, что это опасно! Почему? Что же, в таблице стилей CSS своего приложения я вполне мог бы сделать так, чтобы во всех тегах `<div>` был указан синий фон с очень большим размером шрифта, а все теги `<span>` отображались с прозрачностью 10 %. Если бы вы не знали о существовании этой дополнительной разметки, вы могли бы случайно испортить все календари с возможностью выбора даты, за исключением одной важной вещи: теневая DOM защищает внутреннюю работу вашего веб-компонента из-

вне. Стили ваших синих/больших тегов `<div>` не будут проникать в теневую DOM. Более того, вы не сможете написать код Java Script, чтобы попытаться заполучить кнопку календаря `clear` и манипулировать ей:

```
let myElement = document.getElementById('clear');
```

```
▼<input type="date" name="bday">
  ▼#shadow-root (user-agent)
    ▼<div pseudo="-webkit-datetime-edit" id="date-time-edit" datetimeformat=
    "M/d/yy">
      ▼<div pseudo="-webkit-datetime-edit-fields-wrapper">
          <span role="spinbutton" aria-placeholder="mm" aria-valuemin="1"
          aria-valuemax="12" aria-label="Month" pseudo="-webkit-datetime-edit-
          month-field">mm</span>
          <div pseudo="-webkit-datetime-edit-text">/</div>
          <span role="spinbutton" aria-placeholder="dd" aria-valuemin="1"
          aria-valuemax="31" aria-label="Day" pseudo="-webkit-datetime-edit-
          day-field">dd</span>
          <div pseudo="-webkit-datetime-edit-text">/</div>
          <span role="spinbutton" aria-placeholder="yyyy" aria-valuemin="1"
          aria-valuemax="275760" aria-label="Year" pseudo="-webkit-datetime-
          edit-year-field">yyyy</span>
        </div>
      </div>
      <div pseudo="-webkit-clear-button" id="clear" style="opacity: 0;
      pointer-events: none;"></div>
      <div pseudo="-webkit-inner-spin-button" id="spin"></div>
      <div pseudo="-webkit-calendar-picker-indicator" id="picker"></div>
</input>
```

**Рис. 1.2 Активирование соответствующих настроек в инструментах разработчика Chrome позволяет нам увидеть скрытую теневую модель DOM тега input**

Когда мы пытаемся получить этот элемент, потому что он находится в пределах теневой DOM, оказывается, что элемент не найден, и наша переменная `myElement` имеет значение null. На рис. 1.3 показаны различные попытки с CSS и JS.

Таким образом, Shadow DOM защищает область видимости корня теневого дерева (shadow root). Да, вы можете использовать его, где угодно. Но это имеет огромное значение в пользовательском элементе, который вы создали, чтобы избежать непреднамеренной поломки, когда разработчик устанавливает правило CSS, имя которого совпадает с именем, которое вы использовали в своем компоненте, или когда тот же разработчик запрашивает элемент по классу и что-то в вашем пользовательском элементе выбирается случайно.

Как вы можете себе представить, календарь с возможностью выбора даты является полезным элементом для дополнения нескольких других полезных элементов, которые мы применяем ежедневно. Многие элементы используются в семантических целях, например тег `<footer>`, а другие имеют определенный API и стиль, например теги `<button>`, `<option>` и `<video>`.

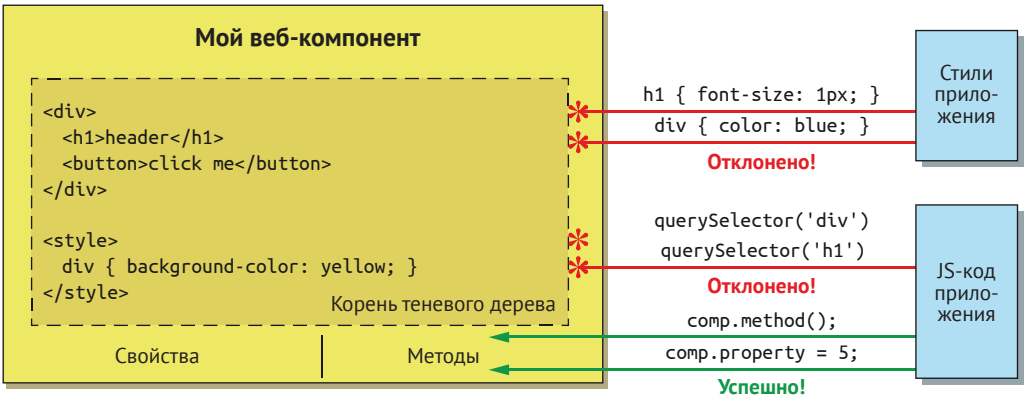



**Рис. 1.3 Теневая DOM защищает ваш компонент от непреднамеренных последствий, когда CSS или JS может повлиять на стили и узлы внутри, не подлежащие изменению. Вместо этого у вашего компонента будет пользовательский API, с которым можно будет взаимодействовать, применяя методы и свойства**

## 1.1.3 Что имеют в виду, когда говорят «веб-компоненты»?

Каким бы чудесным ни был календарь с возможностью выбора даты и любой другой элемент, разве не было бы удивительно, если бы мы могли создавать свои собственные элементы с собственным визуальным стилем, внутренней логикой, возможностью повторного использования и инкапсуляцией?

Это то, что люди имеют в виду, когда говорят о веб-компонентах. В дополнение к инкапсуляции, предоставляемой Shadow DOM, мы можем использовать API пользовательских элементов для создания собственных компонентов, которые выполняют функции, соответствующие нашим потребностям.

Для меня это обещание веб-компонентов. Я хочу взять что-то, что меня интересует, и создать многократно используемый фрагмент, которым могу поделиться со всем миром, своей командой или просто с самим собой, чтобы использовать его в нескольких проектах там, где мне это нужно. С другой стороны, может существовать фрагмент интерфейса, который я нахожу скучным создавать снова и снова. С помощью веб-компонентов я могу создать его один раз, использовать в нескольких проектах и дополнять по мере необходимости. И даже лучше: возможно, кто-то еще создал веб-компонент для того, что нужно мне, и у меня нет времени или опыта, чтобы воссоздать его. Он может поделиться им со мной, и я могу просто использовать его как обычный элемент DOM.

## 1.1.4 Проблемная история импорта HTML

К сожалению, некоторые в сообществе веб-разработчиков считают обещание веб-компонентов нарушенным. Я, конечно, не могу винить их за это. Говоря о конкретных технических функциях, предлагаемых веб-

компонентами, видение начало разваливаться после того, как первоначальная шумиха вокруг веб-компонентов поутихла несколько лет назад.

Приблизительно в 2015 году было широко известно, что стандартный веб-компонент будет создаваться с использованием трех новых функций:

- пользовательские элементы;
- Shadow DOM;
- импорт HTML.

Пока я еще даже не упомянул об импорте HTML. Эта концепция так и не была принята в качестве стандарта. Фактически вначале Google была в значительной степени ответственна за создание рабочих проектов веб-компонентов. Google взяла на себя задачу создавать API-интерфейсы и отправлять их в Chrome в качестве обнадеживающего эксперимента, чтобы посмотреть, будут ли запущены веб-компоненты. Однако ничего не получилось; другие разработчики браузеров в то время не планировали поставлять эту функцию. Firefox, в частности, хотел подождать, чтобы увидеть, насколько большую сенсацию произведут модули ES6/ES2015, и, возможно, когда-нибудь импортировать не только JS, но и HTML.

Импорт HTML был довольно большой потерей. С самого начала планы Google по доставке веб-компонентов зависели от него. Импорт HTML, как показано на рис. 1.4, был фрагментом HTML-кода для объявления разметки или структуры компонента, а также включал в себя код JS, определяющий логику компонента. Импорт HTML был основной точкой входа для веб-компонентов, и без них мы были в растерянности относительно того, как использовать веб-компоненты с разметкой и стилем вообще.

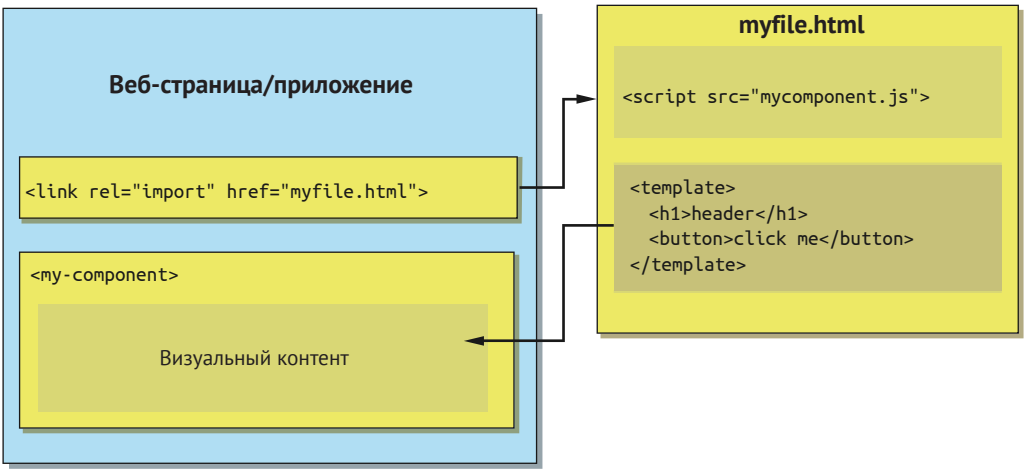



**Рис. 1.4 С помощью импорта HTML-файл, содержащий определение вашего компонента и его разметку, можно импортировать прямо в ваш документ**

Shadow DOM был не намного лучше в то время. Chrome был единственным браузером, принявшим его. Firefox принял его только в октяб-

ре 2018 года, и мы ожидаем, что Microsoft Edge выпустит его, хотя уже доступна предварительная версия для разработчиков.

Shadow DOM и API пользовательских элементов также перешли с версии 0 на 1. Для пользовательских элементов это было несколько проблемным, учитывая, что разработчикам, которые были знакомы с веб-компонентами в это непростое время, было предложено перейти на новый API.

Учитывая все это, вряд ли можно критиковать разработчиков, которые называли веб-компоненты «нарушенным обещанием» и переключались на фреймворк. Могу заверить, что в 2015 году было немного сложно правильно с ними работать, особенно при работе с браузерами, отличными от Chrome.

### 1.1.5 Библиотеки Polymer и X-Tag

Еще одним аспектом того, что имеется в виду, когда речь идет о веб-компонентах, были библиотеки, появившиеся в то время, когда использовали веб-компоненты в качестве основы. Из-за нестабильности, связанной с простыми компонентами без использования фреймворков в то время, библиотеки Polymer от компании Google (https://polymer-library.polymer-project.org) и X-Tag от Mozilla (https://x-tag.github.io) были тем, что люди считали веб-компонентами, или, по крайней мере, единственным способом работы с ними.

Библиотека Polymer проделала большую работу по продвижению стандартов и рабочих процессов, и теперь похоже, что 3.0 является последней официальной версией функции, поскольку она переходит в режим обслуживания. Вместо этого команда разработчиков разбивает некоторые основные инструменты на гораздо более мелкие и более целевые решения, такие как lit-html и LitElement в рамках проекта Polymer. Эти основные инструменты и функции хорошо согласуются с подходом, не предусматривающим использование фреймворков, который я описал в этой книге.

Несмотря на то что команда проделала отличную работу над серией надежных выпусков и сейчас работает над тем, чтобы сосредоточиться на более мелких и дополнительных функциях, в первые дни библиотека Polymer до выхода версии 1.0 была слегка шаткой. Как и ожидалось с любой библиотекой до версии 1.0, API-интерфейсы немного изменились, тем более что они пытались не отставать от меняющихся спецификаций и отсутствия Shadow DOM в каждом браузере, кроме Chrome. С Shadow DOM было особенно трудно иметь дело. Полнофункциональные полифилы, включающие инкапсуляцию CSS, были слишком трудными и влияли на производительность. Чтобы компенсировать это, в качестве облегченной реализации был придуман Shady DOM, который можно было использовать для полизаполнения.

Это было непростое время для веб-компонентов в целом, и библиотека Polymer казалась еще одним фреймворком или библиотекой, которая должна была конкурировать с более надежными библиотеками, не имевшими отношения к промежуточным веб-стандартам.

### 1.1.6 Современные веб-компоненты

Несмотря на эти тяжелые времена, я остановился на веб-компонентах. Я успешно использовал их для проектов, но не был полностью удовлетворен, пока не начал использовать некоторые новые функции языка JS. Функция жирной стрелки оказалась отличным способом управления областью видимости при работе с событиями мыши или таймерами. Что еще более важно, ключевое слово `import` и концепция модулей были огромными.

С помощью этого ключевого слова я смог отойти от хрупкого беспорядка, когда мне нужно было убедиться, что каждый JS-файл, который я хотел использовать, был привязан к тегу `<script>` на моей главной HTML-странице. Каждый веб-компонент может нести полную ответственность за импорт собственного кода. Это означало, что на главной HTML-странице я мог заставить один тег `<script>` на основе модуля импортировать веб-компонент, который содержал все мое приложение. Каждый дочерний компонент просто импортирует все, что ему нужно.

Это открыло возможности для многократно используемых модулей кода, написанных на чистом JS, и дало мне возможность создавать несколько уровней наследования, когда я хотел, чтобы мои компоненты имели общий API и были немного умнее базового API `HTMLElement`. Наконец, я мог хранить свой HTML/CSS в отдельном файле template.js, который мог импортировать, отделяя свои визуальные проблемы от логики контроллера компонента.

Последней огромной функцией JS, которая сделала работу с веб-компонентами приятным удовольствием, был *шаблонный литерал*. Я не только мог хранить свой HTML/CSS в отдельном файле шаблона, но и мог заменить выражения-заполнители в своей разметке переменными и вкладывать несколько шаблонов вместе, используя функции Java Script.

Эти функции ES6/ES2015 неожиданно сделали веб-компоненты приятными для работы.

Даже ранее, работая с ныне устаревшим HTML-импортом, я считаю, что сочетание модулей и шаблонных литералов – гораздо лучший путь, если сравнивать.

Как я уже говорил, Shadow DOM поддерживается на 99 %. На это ушло какое-то время, но все разработчики популярных браузеров поддерживают эту технологию. Мы просто ждем, когда Microsoft выпустит предварительную версию Edge для всех. Лично я только что начал работать с Shadow DOM, после того как ее стал поддерживать Firefox.

В то же время какой бы хорошей ни была технология Shadow DOM, она также не является обязательной. Правда, она дает дочерним элементам нашего компонента хорошую защиту от проникновения стилей и JS, что приводит к неблагоприятным последствиям, но это новое решение проблемы, которая всегда у нас была. Поэтому, если нам нужно подождать поддержки браузера несколько месяцев или просто отказаться от него в краткосрочной перспективе, это еще не конец света. Тем не менее я достаточно долго сдерживал свое волнение относительно Shadow DOM

из-за предыдущей поддержки в браузере; теперь, когда мы собираемся пересечь финишную черту, я взволнован, потому что оказывается, что это так здорово – использовать данную технологию.

Как бы я ни волновался за будущее веб-компонентов, я не слышал ни о каком современном видении относительно них, особенно что касается разработчиков, которых ранее они приводили в смятение. Если бы мне пришлось переопределить «обещание веб-компонентов» на 2019 год, это бы уже не были те три обязательных свойства: пользовательские элементы, Shadow DOM и импорт HTML.

Для меня видение веб-компонентов 2019 года складывается из набора инструментов функций ES6/ES2015 и тега `<template>`, когда, и если вам это нужно, все они служат пользовательскому элементу в качестве основной функции. Как только технология Shadow DOM будет поддерживаться во всех браузерах в ближайшем будущем, она также станет важным дополнением к нашему набору инструментов. Это видение того, как я буду подходить к веб-компонентам в данной книге. Мы подробно рассмотрим пользовательский элемент, а затем изучим рабочие процессы вокруг всех дополнительных инструментов в нашем наборе.

## 1.2 Будущее веб-компонентов

Предсказывать будущее всегда непросто, особенно это касается интернета, где все меняется в безумном темпе. Тем не менее у нас есть несколько убедительных подсказок, указывающих на то, что может произойти с веб-компонентами после 2019 года.

Мы уже видели эксперименты с React, Angular (https://angular.io/guide/elements) и Vue (https://vuejsdevelopers.com/2018/05/21/vue-js-web-component/), где показана компиляция компонентов в каждом из этих фреймворков в отдельный веб-компонент, работающий абсолютно независимо от фреймворка, который сделал эти компоненты. Кроме того, такие инструменты, как StencilJS (https://stenciljs.com) и Svelte (https://svelte.technology), позволяют использовать фреймворк и выполнять компиляцию в автономные веб-компоненты.

Что это значит? Вскоре мы все можем создавать компоненты без какого-либо фреймворка или с помощью фреймворка на свой выбор. Мы будем использовать веб-компонент, созданный React в Angular, или веб-компонент, созданный Vue, на веб-странице без фреймворка. Искусственные стены между разработчиками и их фреймворками могут относительно скоро разрушиться, как показано на рис. 1.5. И все это благодаря веб-компонентам.

Эта концепция может даже распространяться на совместную работу совершенно разных языков. Одно приложение может иметь разные компоненты, разработанные в JS, Typescript и CoffeeScript; учитывая, что каждый из них является модульным компонентом, предоставляющим API, это не имеет значения. Что еще более безумно, с появлением WebAssembly мы могли видеть, что в таких языках, как C ++, Lua, Go и т. д.,

код, скомпилированный в байт-код и обернутый веб-компонентом, выглядит как совершенно нормальный элемент извне, одновременно позволяя получать высокопроизводительную графику, которая может работать быстрее, чем обычно в JS.

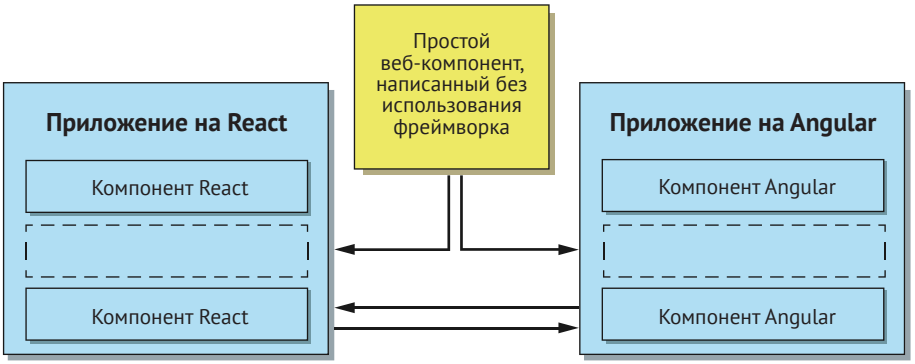



**Рис. 1.5 Веб-компоненты могут в будущем преодолеть разрыв между популярными фреймворками. В этих фреймворках можно использовать не только веб-компоненты без фреймворка, но уже существующие экспериментальные проекты по компиляции компонентов в React, Angular или Vue для независимого запуска компонентов, которые можно применять где угодно**

Я также думаю, что использование модулей ES6/ES2015 и импорта изменит наш взгляд на библиотеки и фреймворки. Мы уже видим два похожих инструмента, lit-html и hyperHTML, для расширенного управления разметкой. Оба из них имеют модули, которые разработчики могут импортировать и вместо того загружать целую библиотеку для решения целевой проблемы. Вам разрешается участвовать или отказаться от участия в любое время, когда захотите, на протяжении своего проекта.

В этой связи я думаю, мы увидим гораздо больше удивительных библиотек. Вы будете импортировать только то, что вам нужно и когда вам это нужно. Людям может быть скучно с веб-компонентами как новой блестящей парадигмой, но я вижу, как мы опираемся на эти основы с помощью импортируемых скриптов и библиотек. Новый подход проекта Polymer, когда команда переводит свою исходную библиотеку в режим обслуживания, кажется, точно соответствует этому. Время покажет, будут ли основные фреймворки разделять функции, как это сделала команда Polymer в случае с lit-html, на отдельные операции импорта, которые можно использовать вне фреймворка. Но мне это кажется неизбежным, особенно если смотреть на другие языки, которые всегда обладали функционалом импорта.

## 1.3 За пределами одного компонента

До сих пор я много говорил о веб-компонентах как об отдельных компонентах, но как бы я ни любил автономные веб-компоненты, они не будут

особенно полезны, если не будут работать вместе для создания вашего приложения.

Задолго до того, как появились веб-компоненты, у нас были отличные способы взаимодействия с обычными элементами DOM. Мы можем использовать те же методы, чтобы придать структуру всему, что мы создаем с помощью веб-компонентов, так же как делаем это с обычными тегами `<div>`, `<video>` или `<input>`.

### 1.3.1 Веб-компоненты как и любой другой элемент DOM

Для начала каждый элемент имеет своего рода публичный API. Под этим я подразумеваю, что вы можете получать и устанавливать свойства для своего элемента и вызывать функции. Например, в случае с элементом video вы можете вызывать функции `pause()` и `play()` для управления воспроизведением видео. Вы также можете уточнить продолжительность видео, проверив свойство `duration`. Наконец, для того чтобы перескочить на определенный момент в своем видео, можно настроить свойство `currentTime`.

Очевидно, что методы и функции для объектов распространены повсеместно в программировании. Элементы DOM ничем не отличаются, как можно увидеть по рис. 1.6; более того, пользовательские веб-компоненты также не являются исключением.

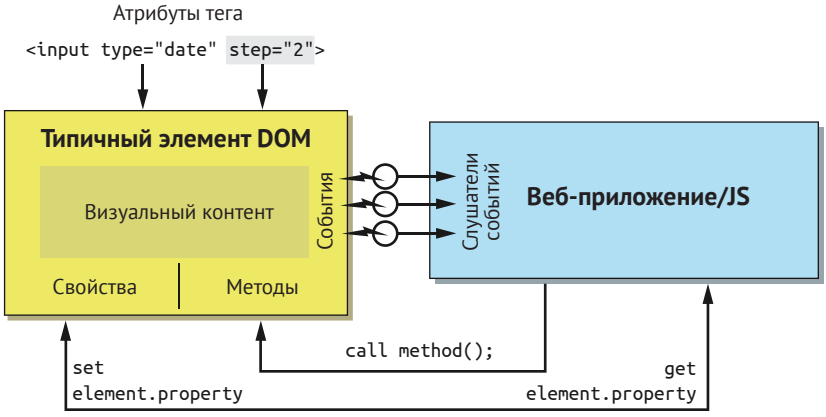



**Рис. 1.6 DOM-элементы имеют различные свойства, методы, события и атрибуты, которые используются, чтобы сообщать элементу, как действовать и взаимодействовать с внешним миром**

Несколько похожими на свойства являются *атрибуты*. Вы видите их все время в HTML-коде. У такого простого тега, как `<img>`, есть атрибут `src`, который указывает на местоположение изображения. Атрибуты представляют собой простую концепцию, но они удобны для предоставления вашему веб-компоненту различного поведения в зависимости от того, как вы хотите, чтобы он работал. Более того, у веб-компонентов есть API-интерфейс, позволяющий внутренне прослушивать изменения в атрибутах.

В предыдущем примере с элементом video атрибуты, предоставляемые тегом, не соответствуют свойствам, предоставляемым API. Хотя мы можем настроить свойство `currentTime`, мы не можем установить тот же атрибут для тега. В противовес этому неоднократно в случае с веб-компонентами, которые вы создаете, у вас будет возникать желание использовать передовой метод *отражения*. При настройке свойств вам понадобится обновить атрибут (и наоборот), чтобы эти атрибуты и свойства были синхронизированы. Конечно, это не жесткое правило, а всего лишь общепринятый передовой метод. До появления веб-компонентов не обязательно было придерживаться отражения. Хорошим примером того, когда что-то может пойти не так, является атрибут `value` тега `<input>`. Здесь этот атрибут устанавливает начальное значение, но при его изменении атрибут остается прежним. Запрос свойства `value` посредством JS вернет самое последнее значение, при условии что оно было изменено. Это сбивает с толку! Но мы просто принимаем это, потому что так всегда работал тег `<input>`. При создании новых веб-компонентов, вероятно, лучше избегать этой путаницы и отражать атрибуты и свойства. В этом смысле атрибут или свойство `muted` элемента `video` является хорошим примером отражения.

Наконец, возможно, вам понадобится прослушать изменения из вашего пользовательского веб-компонента. Мы постоянно используем события в других сценариях. Возьмем, к примеру, нажатие на кнопку. Как правило, для прослушивания клика мы делаем следующее:

```
mybutton.addEventListener('click', functionToCall );
```

Вы также можете создавать и отправлять собственные пользовательские события. Это можно делать из любого места, но они особенно удобны, когда вам нужно, чтобы ваше приложение или другие компоненты внутри него слушали события, поступающие от вашего веб-компонента.

## 1.3.2 От отдельного компонента к приложению

Говорить об отдельных компонентах – это одно, но что делать, когда вам нужно создать целое веб-приложение? Веб-компоненты могут быть настолько большими или маленькими, как вам нужно. Вы можете создать несколько чрезвычайно детализированных компонентов, таких как кнопки, а затем вложить их в более крупный веб-компонент, например в пользовательскую панель инструментов.

Ваш компонент панели инструментов может обрабатывать более мелкие детали работы с кнопками, возможно, включать и выключать их или отключать определенные кнопки при определенных обстоятельствах.

Нашу панель инструментов наряду с другими компонентами, показанными на рис. 1.7, можно дополнительно вложить в другой родительский компонент и т. д. Это может продолжаться до тех пор, пока в вашем теге `<body>` не будет единственного веб-компонента.

Веб-компоненты и JavaScript без фреймворка могут многое предложить для разработки веб-приложений. Но по мере роста вашего прило-

жения растет и его сложность. Координировать взаимодействие ваших компонентов друг с другом становится все труднее.

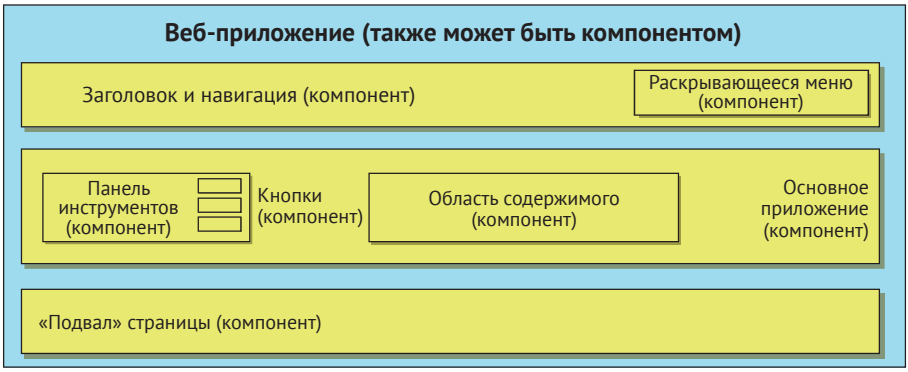



**Рис. 1.7 Пример веб-приложения, состоящего из веб-компонентов, которые сами состоят из большего количества веб-компонентов. Эта иерархия может распространяться на нечто маленькое, как пользовательская кнопка, или быть таким же большим, как все приложение, обернутое как веб-компонент**

Иногда можно обнаружить, что даже с присущей вам структурой, которую дают веб-компоненты, этого недостаточно для создания вашего комплексного приложения. Возможно, вы захотите обратиться к популярным фреймворкам и библиотекам, чтобы они помогли вам со структурированием. Такие фреймворки, как Angular, предлагают привязку данных, шаблоны MVC и многое другое. Конечно, они могут быть полезны при создании традиционного веб-приложения. С другой стороны, мы можем написать и импортировать простой код JS на базе проверенных временем шаблонов проектирования, которые уже давно используются, избегая этих более крупных фреймворков.

Например, собственные события DOM могут не сработать в вашем случае. Зачастую вам нужно, чтобы одна часть вашего веб-приложения передавала сообщение совершенно другой части вашего приложения, и вам не нужно беспокоиться о том, как происходит событие в DOM. Вы *могли бы* обратиться к такой библиотеке, как RXjs или Redux, но это может быть излишним. Вместо этого можно написать простую шину событий с небольшим количеством кода. На рис. 1.8 и 1.9 сравниваются два этих подхода.

На рис. 1.8 у вас могут, например, быть компоненты ввода формы, содержащиеся в веб-компоненте. Эти компоненты ввода могут инициировать изменения ввода текста, выпадающие изменения и многое другое, все для этого родительского компонента. Хорошим примером этого может быть компонент палитры цветов с вводом цвета RGB и ползунками. Родительский веб-компонент (палитра цветов), в котором размещены эти входные компоненты, должен будет передать цвет своему родительскому веб-компоненту в другом событии, чтобы сообщить шестнадцатеричное значение цвета.

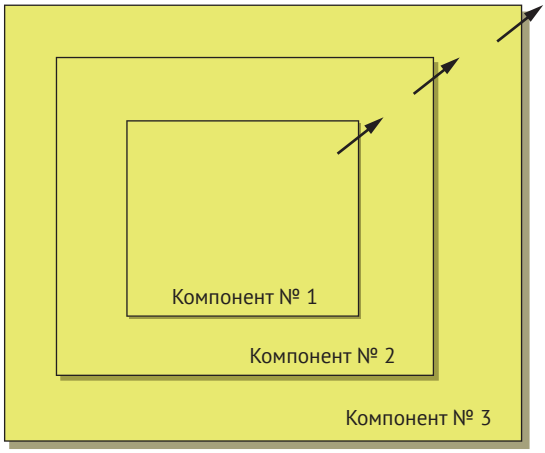



**Рис. 1.8 События естественным образом всплывают из вложенных элементов**

Это естественное всплывание событий может прекратиться, если то, чей цвет вы решили изменить, находится на другой стороне вашего DOM в ином разделе дерева DOM. В таком случае вам нужно будет использовать иную стратегию, например шину событий (рис. 1.9).

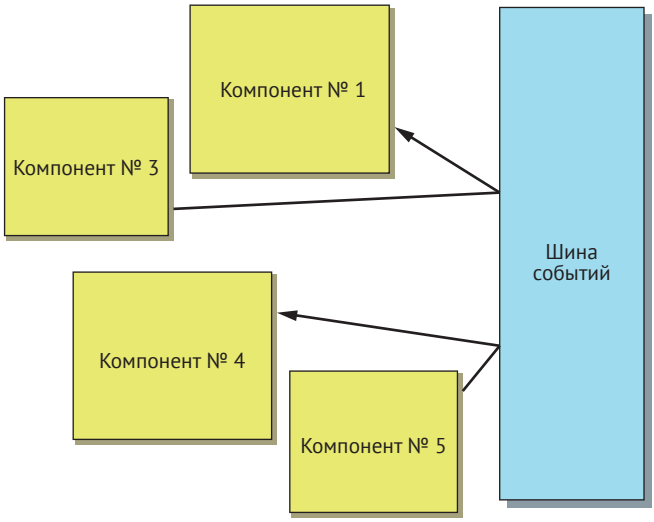



**Рис. 1.9 Если обычное всплывание событий нежелательно, используя небольшое количество кода, вы можете создать систему шины событий для маршрутизации событий туда, куда вы хотите**

Также можно найти золотую середину, используя микрофреймворки. Микрофреймворки могут быть отличным минималистичным способом организации вашего приложения и добавления определенных функций, не слишком обдумывая его, как это делает более крупный фреймворк. Беспокойство по поводу мельчайших деталей в ваших пользовательских

веб-компонентах, а также организация более крупного приложения с помощью этих небольших библиотек может быть прекрасным способом. Даже минималистичные решения для привязки данных и маршрутизации можно найти и через NPM.

## 1.4 Ваш проект, ваш выбор

В конце концов, несмотря на то что для использования веб-компонентов без фреймворка есть веские аргументы, ваш проект и ваша команда в конечном итоге будут влиять на то, что вы используете при создании чего-либо для интернета. Как и любой новый стандарт, веб-компоненты пока не дают ответов на все вопросы. Равно как и ни один популярный фреймворк.

Бывают случаи, когда у вас имеется чрезвычайно простое веб-приложение, и современный фреймворк может быть идеальным ответом, поскольку он справляется со всем, что вам нужно сделать. В других случаях вы можете работать над типом проекта, в котором фреймворки просто мешают. Решения, которые вы можете выбрать, охватывают широкий спектр вариантов, причем некоторые из них перекрываются.

Даже если веб-компоненты без фреймворка не являются правильным решением для вас, однажды ваш любимый фреймворк, скорее всего, будет создан с их использованием, хотя это может быть и неочевидно. Знакомство с основами на базе веб-стандартов любого фреймворка – это всегда хорошая идея, даже если вы не используете их напрямую.

Несмотря на несколько запутанный перерыв в работе веб-компонентов несколько лет назад, сейчас мы находимся в том месте, где они являются реальным вариантом для создания вашего следующего проекта.

Я уверен, что в ближайшие годы мы увидим новые идеи и методы для вашего рабочего процесса веб-компонентов, но эти идеи будут основаны на стандартах, которые я буду рассматривать в данной книге, а также на новейших и появляющихся текущих рабочих процессах. Мы будем изучать веб-компоненты на атомарном уровне, вплоть до приложений, созданных из множества компонентов, а также рассмотрим, как управлять вашим HTML и CSS, организовывать проекты, и многое другое. Я надеюсь, что вы так же волнуетесь, как и я, о будущем интернета!

## Резюме

Из этой главе вы узнали:

- что за последние несколько лет веб-компоненты превратились из рабочего проекта, принадлежащего Google, в настоящий веб-стандарт, принятый всеми современными браузерами;
- о Shadow DOM как дополнительной, но важной функции, находящейся на грани принятия всеми браузерами;

- о месте веб-компонентов в современных фреймворках, а также независимой части любой экосистемы;
- о потенциальном будущем веб-компонентов с постоянно расширяющимся сообществом модулей JS в духе библиотек Polymer Project, таких как lit-html и litelement, а также других библиотек, таких как hyperHTML;
- об отдельном веб-компоненте в сравнении с целым приложением, состоящим из веб-компонентов.

# 2

# Ваш первый веб-компонент

***Эта глава охватывает следующие темы:***

- основа практически каждого элемента, который вы используете: `HTMLElement`;
- расширение классов для создания пользовательских элементов;
- предоставление логики и интерактивности пользовательским элементам;
- использование пользовательских элементов после определения их с помощью метода `customElements.define`.

Как я и обещал в начале этой книги, мы начнем с малого. К счастью, в случае с веб-компонентами, даже когда мы начинаем с малого, все равно можно сделать что-то значимое. После прочтения этой главы вы узнаете, как создать свой первый веб-компонент и просматривать его прямо в браузере! В последующих главах данной книги мы рассмотрим ключевые концепции более подробно, а здесь начинаются основы. В конце этой главы мы обсудим варианты, когда ваш браузер не поддерживает пользовательские элементы, как в случае последнего выпуска Edge для потребителя (на момент написания этих строк) или IE. Однако до тех пор, пожалуйста, используйте браузеры Chrome, Firefox или Safari, если хотите работать с примерами кода.

## 2.1 Знакомство с HTMLElement

До изучения основ веб-компонентов я на самом деле не знал, что такое `HTMLElement`. Возможно, и вы тоже – его не так просто встретить, потому что, хотя это основная концепция работы DOM, мы, как правило, никогда не работали с ней напрямую до сих пор, потому что когда вы добавляете элемент на свою страницу, он просто работает. Вам не обязательно знать, как тег `<input>` связан с тегом `<button>` или как тег `<div>` связан с тегом `<img>`.

Чтобы было понятно, нам нужно разобраться с концепцией наследования. Это популярная концепция в объектно-ориентированном программировании, с которой мы познакомимся позже в данной книге, исследуя возможность повторного использования кода, но в качестве краткого объяснения позвольте мне начать с примера.

### 2.1.1 Ускоренный курс по наследованию

> **ПРИМЕЧАНИЕ** Если вы уже знакомы с наследованием в объектно-ориентированном программировании, перейдите к разделу 2.1.2, чтобы изучить наследование в отношении вашего любимого элемента DOM.

Представьте, что вы в зоопарке. Находясь там, вы замечаете, что у всех животных есть нечто общее. *Животные* должны есть, дышать, спать и передвигаться. Конечно, одни животные отличаются от других. У *млекопитающих* есть мех, они рожают потомство, а не откладывают яйца, и они теплокровные. Млекопитающие имеют все основные характеристики животных, но существуют дополнительные правила, когда вы называете кого-то млекопитающим. Можно пойти еще дальше и рассматривать таких млекопитающих, как тигры, львы и пантеры, как виды семейства *кошачьих*. У кошачьих также есть некие общие черты, например усы, когти, и они плотоядные.

В объектно-ориентированном программировании можно сказать, что представитель семейства кошачьих наследует от млекопитающего, а млекопитающее наследует от животного. Если бы вы писали код, то могли бы начать с определения объекта `Animal` (или *класса*, чтобы быть более конкретным), как показано на рис. 2.1. У него могут быть функции, которые вы можете вызвать, чтобы заставить его дышать (`breath()`), спать (`sleep()`) и есть (`eat()`).

Затем можно создать объект `Mammal`. Было бы утомительно снова и снова писать код функций `breath()`, `sleep()` и `eat()` для объекта `Mammal`. Поскольку все это похоже на `Animal`, мы можем применять наследование; при создании объекта `Mammal` мы говорим, что `Mammal` расширяет `Animal`. `Mammal` автоматически получает все функциональные возможности объекта `Animal`, но мы можем добавить более специфические функциональные возможности, например функцию `growFur()`. Мы можем даже создать объект `Feline`, который наследует от `Mammal`, и поскольку `Mammal` наследует от `Animal`, `Feline` будет обладать всеми функциями `Mammal` и `Animal`.

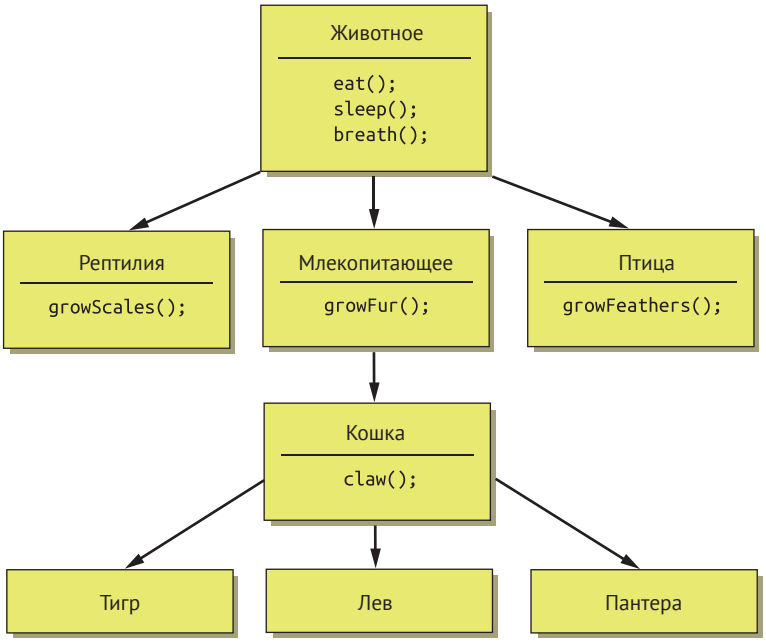



**Рис. 2.1 Не совсем научный пример наследования в животном мире**

Наследование является основной функцией объектно-ориентированного программирования, обычно используемой в других языках, а теперь и в JavaScript (JS), через классы. Если вы незнакомы с этой новой функцией JS, прочтите о ней в приложении «ES2015 для веб-компонентов».

## 2.1.2 Наследование в ваших любимых элементах

Наш пример наследования очень похож на `HTMLElement`. За некоторыми исключениями, такими как SVG, любой элемент, который вы помещаете в HTML или DOM, наследуется от `HTMLElement`.

Хотя `HTMLElement` не является нижней ступенью цепочки наследования в том, что касается браузера (точно так же, как мы можем продолжать переходить от «животного» к «многоклеточному организму», к «живому существу» и т. д.), он служит отправной точкой для веб-компонентов.

В качестве реального примера наследования для реально существующих элементов можно привести теги `<span>`, `<div>` и `<button>`, которые создаются из `HTMLSpanElement`, `HTMLDivElement` и `HTMLButton-Element` соответственно. В свою очередь, все они наследуются от `HTMLElement`. На самом деле вы сами можете убедиться в этом. Откройте консоль браузера и введите это:

```
document.createElement('div').constructor
```

В ответ вы получите следующее:

```
ƒ HTMLDivElement() { [native code] }
```

Здесь мы создаем новый элемент  и спрашиваем, что это за конструктор. Конструктор – это то, что вызывается первым, когда вы создаете подобный объект. Вам сообщают, что конструктор является функцией-создателем определенного класса, в данном случае HTMLDivElement.

Не стесняйтесь экспериментировать с вашими любимыми элементами! Еще один вариант, который можно попробовать, – кнопка:

```
document.createElement('button').constructor
```

что дает нам

```
ƒ HTMLButtonElement() { [native code] }
```

Как видно из наших экспериментов и из рис. 2.2, элементы, которые мы используем постоянно, получены из общего источника: HTMLElement.

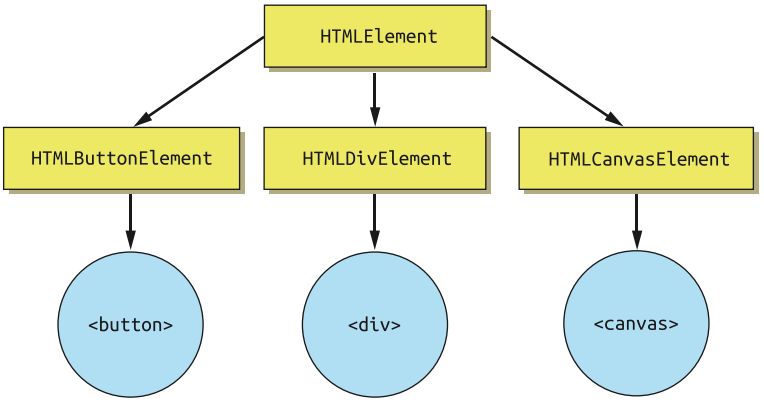



**Рис. 2.2 Хотя существует большое количество классов, которые наследуют от HTMLElement, здесь приводятся три класса, производящих распространенные элементы DOM, используемые нами постоянно, с фактическими тегами, которые мы пишем в нашем HTML-коде**

## 2.2 Правила именования вашего элемента

Есть один интересный момент касательно HTML, который заключается в том, что вы можете создать любое имя для тега и добавить его на свою страницу, и он будет работать как .

Попробуйте использовать это на своей странице:

```
<randomElement>Hi!</randomElement>
```

Вы увидите текст «Hi!», как если бы использовали тег . Теперь вопрос в том, от кого мы наследуем здесь. Попробуем ввести приведенный ниже код в нашей консоли:

```
document.createElement('randomElement').constructor;
```

В ответ мы получим:

```
ƒ HTMLUnknownElement() { [native code] }
```

Вы ожидали увидеть `HTMLUnknownElement`? Вероятно, нет! Мы только что создали недопустимый элемент. Поскольку он недействителен, он наследует от особого класса `Unkown`, и мы не можем расширить его функциональность.

Почему он недопустим? Не потому, что мы не можем придумывать собственные имена элементов при создании собственных компонентов, а потому, что существует соглашение об именах, которому необходимо следовать. Это соглашение является простым жестким требованием для спецификации пользовательского элемента, а именно использовать знак тире (-) в имени вашего элемента. Это позволяет браузеру различать пользовательские и собственные элементы. Если подумать, это не лишено смысла.

Не только такие же читатели, как и вы, будут создавать собственные пользовательские компоненты, но и сами браузеры, скорее всего, также будут предлагать новые элементы.

Обычно веб-компоненты, вероятно, будут использоваться в качестве крошечных фрагментов общего пользовательского интерфейса. Если что-то полезное, например индикатор выполнения, было создано не только вами, но и другими разработчиками веб-компонентов *и* это стало в браузерах нативной функцией, можете представить себе, какой будет беспорядок, если все создадут что-то с именем `<progressbar>`.

Просто добавьте знак тире (-) в имя своего элемента. Если желаемое имя элемента – `<progressbar>`, попробуйте другой вариант с тире: `<progress-bar>`. В идеале можно было бы дать ему *пространство имен*. Пространство имен используется для обозначения какой-то группы, к которой принадлежит ваш компонент. Например, в коллекции элементов Polymer любой компонент пользовательского интерфейса, созданный с помощью дизайн-системы Material, имеет пространство имен `paper`. Если вы зайдете в репозиторий GitHub (https://github.com/PolymerElements), то найдете там `paper-tooltip`, `paper-dropdown-menu` и `paper-toggle-button` (рис. 2.3). У некоторых из них есть два знака тире, и это совершенно нормально. Вам нужен один или несколько таких знаков, чтобы имя было допустимым. Здесь можно сделать важный вывод: Google определяет пространство имен для обозначения набора связанных компонентов, а затем именует конкретный компонент после знака тире. Вы, конечно же, не обязаны следовать той же логике – вам просто нужен этот знак.

Давайте вернемся к `randomElement`, но на этот раз используйте в его названии знак тире, чтобы следовать соглашению:

```
document.createElement('random-element').constructor;
```

Хорошие новости! Вот что мы увидим в нашей консоли:

```
ƒ HTMLElement() { [native code] }
```

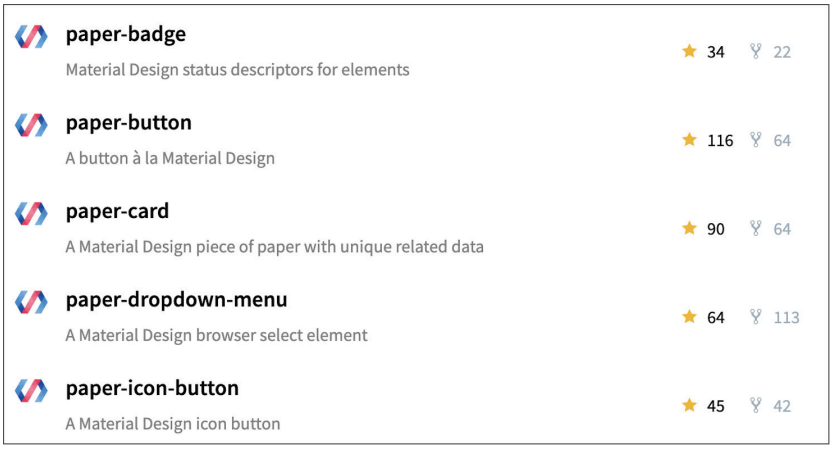



**Рис. 2.3 Небольшая выборка элементов `paper` от Google. Обратите внимание, что эти связанные веб-компоненты пользовательского интерфейса имеют префикс `paper`. Google также использует префикс `iron` для основных элементов и `neon` для элементов, связанных с анимацией**

## 2.3 Определение вашего пользовательского элемента (и обработка столкновений)

Конечно, одно дело – придумать имя для тега и создать его, и совсем другое – фактически предоставить логику и определение тега перед его созданием. Было бы довольно бесполезно создавать собственный тег без какого-либо пользовательского поведения. Нам понадобится выйти за пределы `HTMLElement` и переопределить его, используя собственную логику.

К счастью, это легко сделать! На мой взгляд, это самая большая и полезная часть API веб-компонентов. С помощью одной простой строки кода JS и пустого класса, расширяющего `HTMLElement`, мы можем взять желаемое имя элемента и дать ему значение:

```
customElements.define('my-custom-tag', class extends HTMLElement {});
```

Однако тут есть одна загвоздка, и она не повлияет на вас, пока вы не займетесь более сложными вещами. Тем не менее хорошо бы поднять этот вопрос сейчас: метод `customElements.define` выдаст ошибку, если вы уже определили тег. Это обязательно произойдет позже, когда мы будем использовать более новую функцию JS под названием `import`, где мы включаем наш элемент везде, где нам нужно, чтобы ссылаться на что-то в нем.

Пока же можно сымитировать подобное плохое поведение, вызывая метод `customElements.define` два раза подряд:

```
customElements.define('my-custom-tag', class extends HTMLElement {});
customElements.define('my-custom-tag', class extends HTMLElement {});
```

Мы получаем следующую ошибку:

```
Failed to execute 'define' on 'CustomElementRegistry': this name has already
been used with this registry
```

К счастью, с этим достаточно легко справиться. Мы можем установить, был ли наш пользовательский элемент уже определен, спросив, возвращает ли `customElements.get('my-custom-tag')` что-либо. Оборачивая его в оператор `if/then`, мы гарантируем, что наш элемент определяется только при первом вызове:

```
if (!customElements.get('my-custom-tag')) {
   customElements.define('my-custom-tag', class extends HTMLElement {});
}
```

Теперь расширение `HTMLElement` для определения пользовательского элемента является очень мощным, но не пускайтесь во все тяжкие. Вы, наверное, думаете, что в случае с `HTMLDivElement` или `HTMLButtonElement` расширение также сработает. Было бы неплохо создать кнопку для функций, которые у нее уже есть, например возможность отключить ее или легко работать с формами. К сожалению, ни в одном из браузеров это пока не представляется возможным. В то время как спецификация `customElement` говорит, что все в порядке, Safari еще не реализовал эту функциональность, и поэтому лучше подходить к расширению других элементов с осторожностью либо вообще этого не делать. `HTMLElement` – единственное определение нативных элементов, которое в настоящее время разрешено расширять и создавать пользовательские элементы откуда угодно. Все остальное будет выглядеть так, как будто оно работает, но когда вы будете использовать свой элемент, то получите ошибку:

```
Uncaught TypeError: Illegal constructor: autonomous custom elements must
     extend HTMLElement
```

Также обратите внимание на слова «must extend» (*должны расширять*) в тексте ошибки. Даже если передать `HTMLElement` без расширения его в `customElements.define`, как в `customElements.define('my-element', HTMLElement)`, это приведет к такому поведению, когда вы будете использовать свой новый элемент.

## 2.4 Расширение HTMLElement для создания логики пользовательского компонента

Как вы только что видели, проще всего написать свой пользовательский компонент, применяя более новую функцию JS, называемую *классом*. Классы JS предоставляют отличный и понятный способ показать, как работает наш пользовательский элемент, а также как он наследует от `HTMLElement`.

Давайте начнем с совсем пустого класса, который наследует от `HTMLElement`. Чтобы сделать что-то простое, что станет более полезным по ходу чтения книги, мы начнем с ползунка. Ползунок прост в использовании,

и его также просто создать. По окончании пользователь сможет перетаскивать рукоятку ползунка, чтобы выбрать значение:

```
class Slider extends HTMLElement {}
```

Если подумать над названием элемента, то наиболее очевидным выбором будет `silder`, но нам ведь нужно пространство имен! Поскольку эта книга называется «Веб-компоненты в действии», а ползунок должен представлять собой общий компонент пользовательского интерфейса, который можно использовать где угодно, назовем наш ползунок `wcia-slider`. Теперь, когда ваше новое определение элемента сопоставлено с другими распространенными элементами, как видно на рис. 2.4, вы можете создать что-то особенное:

```
customElements.define('wcia-slider', Slider);
```

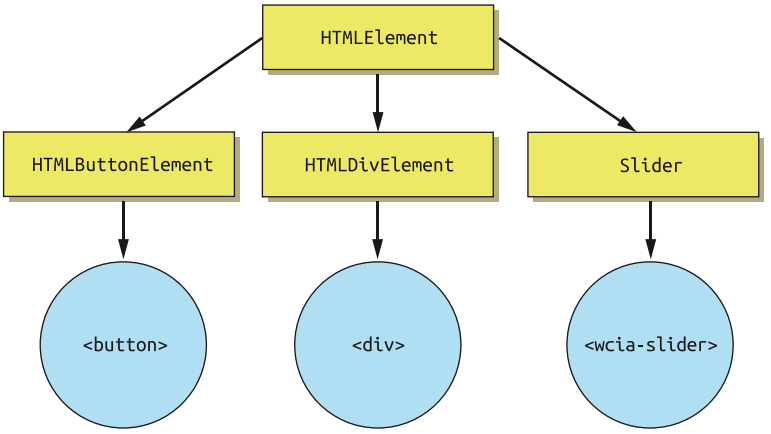



**Рис. 2.4 Диаграмма наследования `HTMLElement` изменена, чтобы включить ваши пользовательские элементы на том же уровне, что и собственные**

Конечно, тут нет никакой пользовательской логики, потому что наш элемент основан на пустом классе. На данный момент он будет действовать так же, как и `HTMLElement`, но мы можем это исправить, используя в этом классе метод `connectedCallback`.

Данный метод является первым из нескольких методов жизненного цикла, предлагаемых API пользовательского элемента и запускаемых при добавлении компонента в DOM.

Давайте изменим наш класс, как показано в приведенном ниже листинге, чтобы дать как-то указать на то, что мы влияем на него. Вот фрагмент, который мы можем включить в нашу страницу, чтобы определить свой элемент.

**Листинг 2.1 Предоставление собственной логики нашему пользовательскому тегу**

```
<script>
    class Slider extends HTMLElement {
        connectedCallback() {
            alert('hi from Slider');
        }
    }
    if (!customElements.get('wcia-slider')) {
        customElements.define('wcia-slider', Slider);
    }
</script>
```

Предупреждение, что компонент находится на странице и работает

Чтобы увидеть это в действии, просто добавьте собственный тег в тело вашего HTML-кода:

```
<body>
    <wcia-slider></wcia-slider>
</body>
```

Когда вы это сделаете, фактически вы не увидите ничего на своей странице, кроме всплывающего диалогового окна с предупреждением. Теперь, когда мы убедились, что можем внедрить логику в наш новый веб-компонент, давайте создадим наш веб-компонент, чтобы сделать его более заметным.

Для этого я должен упомянуть об области видимости и о том, чем она может быть полезна для нас в нашем веб-компоненте. В типичном контексте JS можно легко упустить, на какую область видимости ссылается `this`.

В случае с веб-компонентами и классами мы можем использовать `this` несколькими простыми и понятными способами. За некоторыми заметными исключениями, такими как события обратного вызова и таймеры, `this` в вашем компоненте будет ссылаться на сам элемент. Сюда входят пользовательские методы и свойства, которые вы вводите для элемента, а также любые методы или свойства, которые уже есть у элемента. Другими словами, любой метод или свойство, которое вы можете использовать из обычного, непользовательского элемента, может применяться в этой области, и на него может ссылаться `this`. Ключевое слово `this` – область видимости нашего нового пользовательского элемента.

Примеры того, что вы можете вызвать из `this` внутри своего класса пользовательского элемента, включают в себя все, что наследуется от `HTMLElement`, например получение CSS элемента с помощью `this.style`, получение высоты элемента с помощью `this.offsetHeight` или добавление слушателя событий, когда пользователь кликает на ваш компонент с помощью `this.addEventListener('click', callback)`.

Чтобы дать нашему элементу некое содержимое, в частности фон и рукоятку, давайте начнем со свойства `innerHTML`. Опять-таки, `innerHTML` можно использовать для любого элемента, и он служит для установки содержимого HTML внутри элемента. Аналогично его можно использовать и здесь:

```
this.innerHTML = '<div class="bg-overlay"></div><div class="thumb"></div>';
```

В качестве демонстрации подойдет и такой код. Позднее мы определенно улучшим его.

Давайте также добавим в компонент-ползунок стили. При добавлении стилей и определении работы компонента важно подумать о том, как он будет использоваться и для чего мы его создаем. Учитывая, что у нас уже есть ползунок, изначально предоставленный браузером, как показано на рис. 2.5, и созданный с помощью элемента `<input typ ="range">`, наш ползунок должен служить несколько иной цели.

**Рис. 2.5 Уже готовый ползунок, предоставляемый элементом `input`**

Наш ползунок будет работать так же, по нему будет перетаскиваться рукоятка. Тем не менее мы сделаем нашу дорожку намного больше. Фактически это будет скорее фон, нежели дорожка. Причина создания более крупной дорожки состоит в том, что мы можем лучше визуализировать то, что пользователь будет перетаскивать. Этот ползунок будет использоваться для изменения прозрачности определенного цвета. На рис. 2.6 показан ползунок (слайдер) прозрачности, который мы создадим.

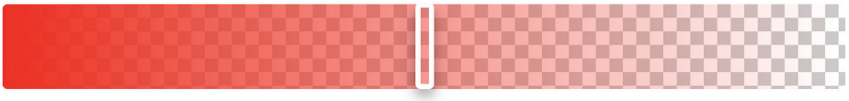


**Рис. 2.6 Наш новый компонент слайдер прозрачности цвета**

Теперь, когда мы определились с тем, как будет выглядеть ползунок, можно приступить к добавлению стилей! Как упоминалось ранее, можно использовать ключевое слово `this` в качестве области видимости нашего компонента, которое мы затем можем подключить к свойству `style`, как и любой другой элемент:

```
this.style.display = 'inline-block';
this.style.position = 'relative';
this.style.width = '500px';
this.style.height = '50px';
```

Помимо обращения к свойству `style` в области видимости компонента, мы можем использовать еще одно свойство `HTMLElement: querySelector`. Обычно мы могли бы использовать `query- Selector` для нашего документа (`document`), чтобы найти дочерний элемент внутри. Например, если бы нам нужно было найти где-нибудь элемент с классом `myelement` на странице, можно было бы сделать следующее:

```
let myElement = document.querySelector('.myelement');
```

Функция `querySelector` не должна выполнять поиск по ширине документа. Вместо этого он может быть ограничен любым обычным элемен-

том для запроса и выбора его дочерних элементов. Учитывая, что наш компонент является обычным элементом, мы можем запросить и выбрать его дочерние элементы и также применить к ним стили. В приведенном ниже листинге показано, как можно получить доступ к веб-компоненту, используя область видимости this для изменения стиля дочерних элементов компонента.

**Листинг 2.2 Запрос и выбор внутренних компонентов и настройка их стиля**

```
this.querySelector('.bg-overlay').style.width = '100%';
this.querySelector('.bg-overlay').style.height = '100%';
this.querySelector('.bg-overlay').style.position = 'absolute';
this.querySelector('.bg-overlay').style.backgroundColor = 'red';

this.querySelector('.thumb').style.marginLeft =
    '100px';
this.querySelector('.thumb').style.width = '5px';
this.querySelector('.thumb').style.height = 'calc(100% - 5px)';
this.querySelector('.thumb').style.position = 'absolute';
this.querySelector('.thumb').style.border = '3px solid white';
this.querySelector('.thumb').style.borderRadius = '3px';
```

Добавление стиля к элементу наложения фона внутри компонента

Добавление стиля к элементу рукоятки внутри компонента

Собрав все воедино, у нас получается то, что показано в этом листинге.

**Листинг 2.3 Полный, но простой пример веб-компонента**

```
<html>
<head>
    <title>Slider</title>

<script>
    class Slider extends HTMLElement {
        connectedCallback() {
            this.innerHTML =
             '<div class="bg-overlay"></div><div class="thumb"></div>';

            this.style.display =
             'inline-block';
            this.style.position = 'relative';
            this.style.width = '500px';
            this.style.height = '50px';

            this.querySelector('.bg-overlay').style.width = '100%';
            this.querySelector('.bg-overlay').style.height = '100%';
            this.querySelector('.bg-overlay').style.position = 'absolute';
            this.querySelector('.bg-overlay').style.backgroundColor = 'red';

            this.querySelector('.thumb').style.marginLeft = '100px';
            this.querySelector('.thumb').style.width = '5px';
            this.querySelector('.thumb').style.height = 'calc(100% - 5px)';
            this.querySelector('.thumb').style.position = 'absolute';
            this.querySelector('.thumb').style.border = '3px solid white';
            this.querySelector('.thumb').style.borderRadius = '3px';
        }
```

Настройка HTML-содержимого нашего веб-компонента

Настройка общего стиля нашего веб-компонента

```
    }
    if (!customElements.get('wcia-slider')) {
        customElements.define('wcia-slider', Slider);
    }
</script>
</head>
<body>
    <wcia-slider></wcia-slider>
</body>
</html>
```

Конечно, теперь вместо простого предупреждения мы видим, что у нашего компонента на странице есть надлежащее содержимое!

## 2.5 Использование вашего пользовательского элемента на практике

На этом этапе, если вы продолжаете, у вас есть собственный пользовательский элемент, работающий на вашей странице. В дополнение к правилу, которое гласит, что в вашем пользовательском теге должно быть тире, *ранее существовало* дополнительное правило, касающееся использования тега. Пользовательские элементы подпадают под тип элемента, который не может быть выражен как пустой или самозакрывающийся тег. Другими словами, приведенные ниже варианты до недавнего времени не работали:

```
<wcia-slider /> или <wcia-slider>
```

Однако теперь в последних версиях браузеров работают даже эти варианты. Таким образом, кроме требования тире, ваш элемент может использоваться всеми способами, которыми могут использоваться и другие элементы. Когда вы будете иметь дело с более сложными компонентами, логика кликов, вероятно, будет находиться внутри вашего компонента, но мы, безусловно, можем связать событие клика с нашим веб-компонентом, как и с любым другим элементом:

```
<wcia-slider onclick="alert('clicked')"></wcia-slider>
```

Атрибуты также удобны в использовании, но, конечно, существует не так много ситуаций, когда в вашем компоненте было бы полезно использовать атрибут без логики. Давайте изменим способ визуализации нашего компонента с помощью нескольких разных атрибутов: цвета и значения. Эти атрибуты можно встроить в тег компонента:

```
<wcia-slider backgroundcolor="#0000ff" value="180"></wcia-slider>
```

Затем мы можем изменить цвет фонового элемента, поменяв «красный» цвет на значение атрибута:

```
this.querySelector('.bg-overlay').style.backgroundColor =
this.getAttribute('backgroundcolor');
```

Между тем атрибут value может изменить положение рукоятки ползунка:

```
this.querySelector('.thumb').style.marginLeft = this.getAttribute('value') + 'px';
```

Проделав эти небольшие изменения, мы можем теперь изменить цвет и положение ползунка на что угодно. К сожалению, компонент выглядит несколько неприглядно, как показано на рис. 2.7, и это не совсем то, что я показывал вначале.

**Рис. 2.7 Ползунок (теперь он синий, и рукоятка сдвинута влево), на который влияют атрибуты value и color**

Возможно, вы также заметили, что я использовал JS для настройки свойств стиля вместо того, что я должен был использовать: CSS. Как и в случае с любым другим элементом, мы можем поместить наш элемент и внутренние дочерние элементы в блок `<style>` и сделать все так, как я обещал изначально.

Кое-что из CSS, который мы используем, выглядит довольно сложным. По этой причине пока я удалил атрибуты – мы вернемся к этому в последующих главах, используя фактический код, чтобы сделать компонент функциональным как нечто, что вы действительно бы использовали.

Мы добавим позади компонента фон с клеточками с помощью немного полусумасшедших стилей (фоновое изображение, расположение и размер). Если честно, сам я не создавал его – я нашел его в интернете! Правила линейного градиента для цвета и свойство box-shadow для рукоятки также немного длинные, но эти мелкие детали в приведенном ниже листинге могут в дальнейшем украсить пользовательский интерфейс.

**Листинг 2.4 Извлекаем выгоду от встроенных стилей CSS**

```
<html>
<head>
    <title>Slider</title>

    <script>
        class Slider extends HTMLElement {
            connectedCallback() {
                this.innerHTML =
                  '<div class="bg-overlay"></div><div class="thumb"></div>';
            }
        }

        if (!customElements.get('wcia-slider')) {
            customElements.define('wcia-slider', Slider);
        }
```

```
    </script>

    <style>
        wcia-slider {
            display: inline-block;
            position: relative;
            border-radius: 3px;
            height: 50px;
            width: 500px;
            background-image: linear-gradient(45deg, #ccc 25%,
              transparent 25%),linear-gradient(-45deg, #ccc 25%,
              transparent 25%),linear-gradient(45deg, transparent 75%,
              #ccc 75%),linear-gradient(-45deg, transparent 75%, #ccc 75%);
            background-size: 16px 16px;
            background-position: 0 0, 0 8px, 8px -8px, -8px 0px;
        }

        .bg-overlay {
            width: 100%;
            height: 100%;
            position: absolute;
            border-radius: 3px;
            background: linear-gradient(to right, #ff0000 0%, #ff000000 100%);
        }

        .thumb {
            margin-top: -1px;
            left: 250px;
            width: 5px;
            height: calc(100% - 5px);
            position: absolute;
            border-style: solid;
            border-width: 3px;
            border-color: white;
            border-radius: 3px;
            pointer-events: none;
            box-shadow: 0 4px 8px 0 rgba(0, 0, 0, 0.2),
                        0 6px 20px 0 rgba(0, 0, 0, 0.19);
        }
    </style>
</head>
<body>
    <wcia-slider></wcia-slider>
</body>
</html>
```

**Добавление довольно сложного CSS в наш компонент, чтобы получить правильный стиль**

В этом случае мы можем легко настроить все стили прямо в нашем CSS. Вы, возможно, заметили, что мы действуем немного дезорганизованно, просто поместив блоки скриптов и стилей в HTML-код. Не волнуйтесь, мы разберемся с этим, чтобы все выглядело намного чище, по ходу книги. На данный момент, даже для простого компонента, это выглядит как вполне стилизованный ползунок (рис. 2.8), который может быть полезен для более глубокого изучения позже.

Рис. 2.8 Стилизованный слайдер с использованием CSS

## 2.6 Создание (полезного) первого компонента

Посмотрим правде в глаза: исходя из того, что мы пока узнали о пользовательских элементах и создании пользовательской логики в вашем первом компоненте, это был не очень полезный компонент (например, рукоятка не двигается). Не волнуйтесь! По мере продвижения в этой книге мы будем дорабатывать его, добавляя интерактивность, исследуя некоторые стандартные методы, позволяя ему работать с другими компонентами.

Однако сейчас пришло время взять то, что мы уже изучили, а также некоторые из наших предыдущих знаний в области веб-разработки и создать простой веб-компонент, который сразу же может быть полезен и значим как самостоятельный компонент.

Первое, что приходит на ум при простом случае использования, – это то, что создателям веб-сайтов всегда было нужно, и оно было сделано и переделано бесчисленное количество раз в jQuery. Его с пользой можно применять во всех веб-контекстах, начиная от блогов и заканчивая веб-приложениями. Я говорю о карусели фотографий или изображений.

Идея здесь состоит в том, чтобы создать компонент, который можно поместить в любое место на странице и который позволит нам указать название альбома и автора и пролистать альбом фотографий с помощью кнопок «Вперед» и «Назад». Для этого я выбрал несколько фотографий с популярного хостинга изображений imgur.com (я скопировал их в свой репозиторий на GitHub, чтобы они не исчезли со временем) в качестве мест, которые, как я думаю, было бы весело посетить и поместить их в альбом. Мой компонент «будущие фотографии из отпуска» в итоге выглядит, как показано на рис. 2.9.

Без сомнения, я мог бы потратить больше времени на стилизацию и создание графики, особенно для кнопок «Вперед» и «Назад», но здесь мы оставим простые кнопки. Давайте рассмотрим все более подробно и шаг за шагом создадим эту фотокарусель.

### 2.6.1 Настраиваем свой веб-сервер

Есть простые вещи, которые можно делать без использования веб-сервера. Одна только загрузка простого HTML-файла в вашей локальной файловой системе в выбранном вами браузере вам не поможет. При попытке загрузить настройки, файлы CSS или JS при отсутствии сервера эти файлы будут заблокированы. Что делать?


**Рис. 2.9 Результат следующей демонстрации, где мы создаем фотокарусель**

Лично мне нравится использовать интегрированную среду разработки (IDE), такую как WebStorm от компании JetBrains, которая автоматически создает сервер за вас, когда вы загружаете свой HTML-файл через ее пользовательский интерфейс. Многие веб-разработчики используют простой текстовый редактор и HTTP-сервер. Поскольку это бесплатный вариант, давайте так и сделаем! Node.js – отличный выбор, особенно потому, что он используется для множества других вещей, связанных с инструментами внешнего интерфейса. Если вы еще не установили Node.js, перейдите по ссылке https://nodejs.org, скачайте его и установите.

После установки мы можем использовать менеджер пакетов Node.js, NPM, для установки модулей по своему выбору. Обычно в случае с Node модули устанавливаются специально для своего проекта. На этот раз мы передадим флаг `-g`, чтобы установить модуль `http-server`, который можно использовать откуда угодно. Откройте терминал командной строки (не важно, в каком каталоге вы находитесь) и наберите это:

```
npm install http-server -g
```

Когда вы закончите и не будет никаких ошибок, у вас будет простой веб-сервер, который можно запускать из любого места на вашем компьютере. Теперь, когда у вас есть установленный инструментарий, вы можете создать папку проекта, где вам будет угодно. Я назову свой проект «photocarousel» и создам для него пустую папку на рабочем столе. Как только папка будет создана, я создам пустой HTML-файл с именем test.html, чтобы убедиться, что мой сервер работает и мой файл загружается. В своем любимом текстовом редакторе напишите HTML-код в следующем листинге (опять же, это просто для того, чтобы сделать что-то, что можно просмотреть в своем браузере).

**Листинг 2.5 Простая веб-страница для тестирования нашего сервера**

```
<html>
    <head>
        <title>Photo Carousel Demo</title>
    </head>
    <body>
        <h3>Hi, from your webserver</h3>
    </body>
</html>
```

Теперь в своем терминале перейдите к созданной вами папке проекта и наберите

```
http-server
```

Поскольку модуль `http-server` устанавливается глобально, везде, где вы выполняете эту команду, запускается веб-сервер. В случае успеха вы увидите такие результаты:

```
Starting up http-server, serving ./
Available on:
  http://127.0.0.1:8080
  http://10.0.0.17:8080
Hit CTRL-C to stop the server
```

Теперь в вашем браузере (давайте будем использовать Chrome или Safari) вы можете перейти по любому из этих адресов, добавив /test.html, и увидеть свой HTML-файл в действии.

Потрясающе! Если вы видите что-то вроде того, что изображено на рис. 2.10, это означает, что теперь у вас есть среда разработки!


**Рис. 2.10 Запуск нашей простой HTML-страницы с веб-сервера**

## 2.6.2 Пишем свой HTML-тег

Итак, мы собираемся написать свой пользовательский тег для карусели фотографий в теле нашей HTML-страницы. На самом деле это не сработает, но поможет нам подумать над тем, какие функции мы хотим реализовать, когда дело доходит до работы с веб-компонентами.

Я выберу для этого компонента пространство имен `wcia`, сокращение от *Web Components in Action*. Итак, имя моего тега – `wcia-photo-carousel`. Я мог бы просто добавить этот тег в код страницы:

```
<body>
    <wcia-photo-carousel></wcia-photo-carousel>
</body>
```

Теперь у нас есть возможность поразмыслить над разными вещами, которые мы могли бы изменить, когда речь заходит о нашем компоненте извне. Лично я думаю, что нам понадобится дать нашей карусели название альбома, которое будет отображаться над фотографией, а также имя автора, который создал фотоальбом. Однако наиболее важными являются фотографии, которые мы хотим показать в своем альбоме. Для этого мы передадим список разделенных запятыми URL-адресов. Это означает, что наш тег выглядит не так, как предыдущий пустой тег, а так, как показано в приведенном ниже листинге.

**Листинг 2.6 Наш компонент фотокарусели, используемый на веб-странице**

```
<body>
    <wcia-photo-carousel
            title="Future Vacation Photos"
            author="Ben Farrell"
            photos="images/fBmIASF.jpg,images/3zxD6rz.jpg,images/
            nKBgeLOr.jpg,images/yVjJZ1Yr.jpg"
    ></wcia-photo-carousel>
</body>
```

Атрибут title

Атрибут author

Еще один атрибут, содержащий разделенный запятыми список фотографий для отображения

Теперь, когда мы разобрались с входными данными нашего веб-компонента, можно начать думать о реализации.

## 2.6.3 Создаем свой класс

Как я уже говорил ранее в этой главе, есть более подходящие способы организовать свой код. Однако сейчас мы просто добавим тег `<script>` в наш заголовок HTML-кода, чтобы зарегистрировать наш компонент и запустить наш метод `connectedCallback`.

Сразу после тега `<title>` в нашем заголовке можно добавить блок скрипта, показанный в приведенном ниже листинге.

**Листинг 2.7 Добавляем блок скрипта с классом для определения нашего компонента**

```
<head>
    <title>Photo Carousel</title>

    <script>
        class PhotoCarousel extends HTMLElement {
            connectedCallback() {
            }


        if (!customElements.get(
          ➥'wcia-photo-carousel')) {
```

Класс для определения нашего компонента

Определяем тег для нашего компонента, если он еще не был определен

```
            customElements.define('wcia-photo-carousel', PhotoCarousel);
        }
    </script>
</head>
```

Здесь мы создали класс с именем `PhotoCarousel`, расширяющий `HTMLElement`. Мы создали пустой метод `connectedCallback`, который можно заполнить мгновенно. Далее ниже мы проверяем, определен ли наш тег `wcia-photo-carousel`, и если нет, определяем его как пользовательский элемент.

## 2.6.4 Добавляем содержимое в наш компонент

Теперь можно начать думать о том, какие элементы добавить в наш компонент, чтобы получить карусель, которая нам нужна. Лично я думаю, что подзаголовки title и author будут иметь смысл. Это могут быть теги заголовков `<h2>` и `<h4>` соответственно. Нам также понадобятся две кнопки – одна для перехода к следующей фотографии и одна для перехода к предыдущей. Наконец, нам нужен тег `<div>` для размещения наших фотографий.

Мы поговорим о шаблонных литералах позже, что поможет нам лучше построить HTML-код, а пока мы просто установим для свойства `innerHTML` значение в виде длинной строки, содержащей все упомянутые элементы. Мы сделаем это, когда наш компонент будет добавлен на страницу внутри нашего метода `connectCallback`, как показано в этом листинге.

**Листинг 2.8 Установка HTML-содержимого нашего компонента**

```
this.innerHTML = '<h2>'+ this.getAttribute('title') + '</h2>' +
    '<h4>by '+ this.getAttribute('author') + '</h4>' +
    '<div class="image-container"></div>' +
    '<button class="back">&lt</button>' +
    '<button class="forward">&gt</button>';
```

Обратите внимание, что мы используем атрибуты наших тегов `title` и `author` для отображения этой пользовательской информации. Как видно по рис. 2.11, у нас неплохое начало.


**Рис. 2.11 Наш компонент карусели с кнопками title, author и forward/back**

Видно, что здесь есть почти все, что мы добавили, – заголовок, подзаголовок, две кнопки, – но не контейнер изображений. Это связано с тем, что, хотя контейнер изображения и был добавлен, внутри контейнера ничего нет, и мы не указали его размер.

Поэтому хотя он и был добавлен в DOM, его просто не видно. Это подходящее время, чтобы приступить к стилизации нашего содержимого.

## 2.6.5 Добавляем стили

Сразу после нашего тега `<script>` в приведенном ниже листинге мы добавим блок стилей.

**Листинг 2.9 Добавление CSS для стилизации нашего компонента**

```
. . .
</script>
<style>
    wcia-photo-carousel {                          ← Стилизация всего компонента
        width: 500px;
        height: 300px;
        display: flex;
        padding-top: 10px;
        flex-direction: column;
        border-color: black;
        border-width: 1px;
        border-style: solid;
    }

    wcia-photo-carousel h2, h4 {                   ← Стилизация двух заголовков (title и author)
        margin-bottom: 0;
        margin-top: 0;
        margin-left: 10px;
    }

    wcia-photo-carousel .image-container {         ← Стилизация элемента div, содержащего наши изображения
        margin-top: 15px;
        flex: 1;
        background-color: black;
    }
</style>
```

Сначала мы устанавливаем общий стиль нашего контейнера компонентов для карусели фотографий. Я произвольно решил, что эти ширина и высота будут составлять 500 и 300 пикселей соответственно. Вы можете изменить эти параметры на свое усмотрение. Мне также нужен простой в использовании макет, поэтому я использовал модуль CSS Flexbox с направлением столбца, чтобы расположить свои элементы вертикально. Я также поместил рамку вокруг своего компонента и отступ сверху, чтобы у заголовка было пространство.

Затем я обнуляю величину отступа от нижнего и верхнего краев элемента для заголовков `h2` и `h4`. Заголовки обычно имеют довольно большое расстояние сверху и снизу, и здесь мне это не нужно. Я также сместил за-

головки на 10 пикселей влево, чтобы они не касались левой части моего компонента.

Наконец, я установил величину отступа от верхнего края для элемента `<div>`, контейнера изображения, равной 15 пикселям, чтобы между ним и заголовками было пространство, и сделал ему черный фон. Значение, равное 1, для `flex` здесь означает, что этот контейнер изображений займет все оставшееся пространство, которое я оставлю вокруг элементов, у которых уже есть высота, например кнопки и заголовки.

Теперь все начинает обретать форму! Наша ограниченная стилизация дает нам нечто похожее на то, что изображено на рис. 2.12.

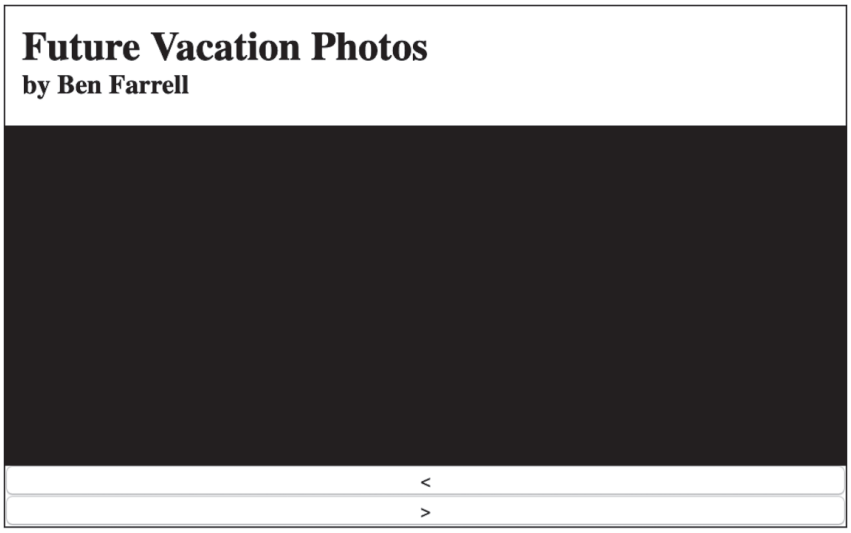



**Рис. 2.12 Прогресс, достигнутый после добавления стилей**

Здесь достаточно макета, чтобы теперь мы могли сосредоточиться на логике компонента.

## 2.6.6 Логика компонента

Раздумывая о том, что делать дальше, вы, возможно, помните, что мы еще не использовали список URL-адресов изображений, которые есть у нас в нашем теге в теле страницы. У нас также нет счетчика, значение которого начинается с 0 и увеличивается или уменьшается, и кнопок, которые используются в качестве указателя того, на какой фотографии мы находимся.

Давайте начнем. В методе `connectCallback`, перед настройкой `innerHTML`, как мы уже делали, добавим следующее:

```
connectedCallback() {
    this._photoIndex = 0;
    this._photos = this.getAttribute('photos').split(',');
    this.innerHTML = '<h2>'+ this.getAttribute('title') + '</h2>' + . . .
```

Здесь мы берем наш список фотографий и превращаем его в массив, используя запятые в качестве разделителей. Наряду с указателем фотографии, на которой мы находимся, для этого массива фотографий установлены внутренние свойства «экземпляра» нашего класса. Доступ к области видимости внутри каждого метода нашего класса можно получить с помощью ключевого слова `this`.

Давайте также создадим метод для отображения нашей фотографии в приведенном ниже листинге, а также вызовем его после того, как настроим наш `innerHTML`.

**Листинг 2.10 Добавление метода `showPhoto`**

```
connectedCallback() {
    this._photoIndex = 0;
    this._photos = this.getAttribute('photos').split(',');

    this.innerHTML = '<h2>'+ this.getAttribute('title') + '</h2>' +
        '<h4>by '+ this.getAttribute('author') + '</h4>' +
        '<div class="image-container"></div>' +
        '<button class="back">&lt</button>' +
        '<button class="forward">&gt</button>';

    this.showPhoto();
}

showPhoto() {
    this.querySelector('.image-container').style.backgroundImage =
        'url(' + this._photos[this._photoIndex] + ')';
}
```

Вызываем метод showPhoto, как только запустится компонент

Метод showPhoto, который устанавливает фоновое изображение элемента div

Наш метод `showPhoto` находит контейнер изображений путем запроса и выбора чего-либо с помощью класса `image-container`, но только в рамках нашего компонента, поскольку мы применяем `this.querySelector` вместо `document.querySelector`, который вы, возможно, используете обычно. Затем он устанавливает фоновое изображение для нашей текущей фотографии. Чтобы увидеть это в действии, убедитесь, что у вас есть папка с изображениями, названная так, как вы указали в исходном атрибуте `photos` для компонента. В репозитории на GitHub для этой книги эта папка уже настроена, чтобы вам было проще.

Однако тут есть проблема. Хотя технически это работает и показывается нужное фото, мои фотографии слишком большие! Все, что я вижу, – это голубое небо, а остальное не видно. В приведенном ниже листинге давайте добавим еще пару свойств стиля в наш контейнер изображений.

**Листинг 2.11 Добавляем стили для правильного отображения текущего изображения**

```
wcia-photo-carousel .image-container {
    margin-top: 15px;
    flex: 1;
    background-color: black;
    background-size: contain;
```

Делает так, чтобы наше изображение помещалось в содержащий элемент

```
    background-repeat: no-repeat;
    background-position: 50%;
}
```

Не повторять изображение и не заполнять контейнер

Центрируем изображение

Давайте более детально рассмотрим эти три правила, которые мы только что добавили. `background-size: contain;` означает, что мы устанавливаем размер так, чтобы изображение помещалось в контейнер, чтобы убедиться, что мы показываем всю фотографию целиком. `background-repeat: no-repeat;` переопределит поведение, при котором изображение повторяется снова и снова. Обычно при повторяющемся действии по умолчанию оно заполняет все оставшееся пространство, потому что изображение не точно соответствует размеру контейнера, который мы ему даем (пока нам не повезет). Здесь `no-repeat` деактивирует подобное поведение. Наконец, `background-position: 50%;` означает, что мы центрируем изображение как по вертикали, так и по горизонтали в нашем контейнере. После этого мы можем увидеть первую фотографию в альбоме, у нее отличный размер, и она располагается по центру, как показано на рис. 2.13.


**Рис. 2.13 Отображение текущего изображения**

## 2.6.7 Добавляем интерактивности

Думаю, что следующий очевидный шаг – заставить наши кнопки работать, чтобы они показывали следующую или предыдущую фотографию. Мы начнем с добавления двух строк в конец нашего метода `connectedCallback`:

**Листинг 2.12 Добавляем слушателей клика к нашим кнопкам**

```
. . .'<button class="back">&lt</button>' +
       '<button class="forward">&gt</button>';

this.showPhoto();

this.querySelector('button.back').addEventListener('click', event =>
    this.onBackButtonClick(event));

this.querySelector('button.forward').addEventListener('click', event =>
    this.onForwardButtonClick(event));
}
```

Слушает клики по предыдущей/задней кнопке

Слушает щелчки по кнопке вперед/назад

С помощью этих строк мы находим кнопки «назад» и «вперед» и добавляем к ним слушателя событий, чтобы при клике они вызывали методы onBack или onForward-ButtonClick.

Вы, наверное, заметили толстую стрелку: =>. Если вы никогда ее не видели, не волнуйтесь. Это более новая функция JS, описание которой приводится в приложении. Обычно вы бы сделали следующее:

```
this.querySelector('button.forward').addEventListener('click',
     this.onForwardButtonClick));
```

Толстая стрелка позволяет нам сохранять ту же область видимости, что и экземпляр нашего класса, когда вызывается функция. Мы можем получить доступ к свойствам и методам экземпляра класса (this) из обратного вызова, показанного в приведенном ниже листинге.

**Листинг 2.13 Обработка наших слушателей события click**

```
/**
 * Обработчик, когда пользователь нажимает кнопку "назад";
 * @param event
 */
onBackButtonClick(event) {
    this._photoIndex --;
    if (this._photoIndex < 0) {
        this._photoIndex = this._photos.length-1;
    }
    this.showPhoto();
}

/**
 * Обработчик, когда пользователь нажимает кнопку "вперед";
 * @param event
 */
onForwardButtonClick(event) {
    this._photoIndex ++;
    if (this._photoIndex >= this._photos.length) {
        this._photoIndex = 0;
    }
    this.showPhoto();
}
```

Обработчик для кнопки «назад»

Если мы находимся на первом изображении, перебирает фото до последнего изображения

Обработчик для кнопки «вперед»

Если мы находимся на последнем изображении, перебирает фото до первого изображения

Эти методы увеличивают или уменьшают текущий указатель нашей фотографии, а затем проверяют, находится ли указатель за пределами нашего массива. Если это так, мы будем перебирать фото, идя к началу или концу нашего массива. Наконец, мы вызываем наш предыдущий метод, чтобы показать текущую фотографию, учитывая новый `this._photoIndex`.

Несмотря на то что внешний вид нашего компонента не изменился, теперь мы можем кликать на эти кнопки, чтобы перейти к предыдущей фотографии или вернуться назад!

## 2.6.8 Последние штрихи

Все готово? Не совсем. Мне не нравятся кнопки «вперед» и «назад». Давайте разместим их по бокам, чтобы наш веб-компонент выглядел как настоящая карусель.

Сперва давайте добавим еще несколько стилей в приведенный ниже листинг, на этот раз для наших кнопок.

**Листинг 2.14 Добавляем стили для кнопок**

```
wcia-photo-carousel button {
    cursor: pointer;
    background: transparent;
    border: none;
    font-size: 48px;
    color: white;
    position: absolute;
    top: 50%;
}

wcia-photo-carousel button.back {
    left: 10px;
}

wcia-photo-carousel button.forward {
    right: 10px;
}
```

**Общие стили для обеих кнопок (кнопки пока не будут видны)**

**Стиль для кнопки «назад»**

**Стиль для кнопки «вперед»**

Назад или вперед, мы хотим, чтобы наши кнопки отображали курсор указателя при наведении его на них. Мы также хотим избавиться от внешнего вида кнопок браузера по умолчанию, поэтому удалим фон и рамку с наших кнопок. Далее мы сделаем очень большой размер для шрифта и цвет текста белым. Наконец, позволим кнопкам выйти за пределы столбца flex, чтобы они появлялись над изображением, установив для свойства `position` значение `absolute`. Мы также расположим их по центру по вертикали, установив для свойства `top` значение 50 %. Для каждой кнопки мы укажем расстояние от левого и правого края в 10 пикселей.

Если бы вы посмотрели на то, что сейчас у вас получилось, вы, вероятно, даже и не увидели бы свои кнопки, потому что если вы похожи на меня, окно вашего браузера открыто довольно большое, а кнопки при значении `absolute` для свойства `position` центрированы по всей странице,

а не на компоненте. Поскольку это белые кнопки на белой странице, вы их не увидите. Нам нужно добавить еще одно свойство CSS, чтобы расположить кнопки относительно компонента, а не страницы:

```
<style>
    wcia-photo-carousel {
        position: relative;
```

Здесь мы устанавливаем для расположения всего нашего компонента значение `relative`. Оно ничего не делает с нашим компонентом, но любой элемент внутри с `position: absolute` теперь относится к компоненту, а не к странице.

На случай, если вы где-то что-то пропустили, весь наш демонстрационный код можно увидеть в этом листинге.

**Листинг 2.15 Повторение демонстрационного кода**

```
<html>
<head>
    <title>Photo Carousel</title>

    <script>
        class PhotoCarousel extends HTMLElement {
            connectedCallback() {
                this._photoIndex = 0;
                this._photos = this.getAttribute('photos').split(',');

                this.innerHTML = '<h2>'+ this.getAttribute('title') + '</h2>' +
                    '<h4>by '+ this.getAttribute('author') + '</h4>' +
                    '<div class="image-container"></div>' +
                    '<button class="back">&lt</button>' +
                    '<button class="forward">
                       &gt
                     </button>';

                this.showPhoto();
                this.querySelector('button.back').addEventListener('click',
                  event => this.onBackButtonClick(event));

                this.querySelector('button.forward').addEventListener('click',
                  event =>
                this.onForwardButtonClick(event));
            }

            /**
             * обработчик, когда пользователь нажимает кнопку «Назад»
             * @param event
             */
            onBackButtonClick(event) {
                this._photoIndex --;
                if (this._photoIndex < 0) {
                    this._photoIndex = this._photos.length-1;
                }
                this.showPhoto();
            }
```

Класс, определяющий наш компонент

HTML-содержимое компонента

Слушатели событий нажатия кнопки

Обработчик нажатия кнопки «Вперед»

```
        /**
         * обработчик, когда пользователь нажимает кнопку «Вперед»
         * @param event
         */
        onForwardButtonClick(event) {
            this._photoIndex ++;
            if (this._photoIndex >= this._photos.length) {
                this._photoIndex = 0;
            }
            this.showPhoto();
        }

        showPhoto() {
            this.querySelector('.image-container').style.backgroundImage
                = 'url(' + this._photos[this._photoIndex] + ')';
        }
    }

    if (!customElements.get(
      ➥'wcia-photo-carousel')) {
        customElements.define('wcia-photo-carousel', PhotoCarousel);
    }

</script>

<style>
    wcia-photo-carousel {
        position: relative;
        width: 500px;
        height: 300px;
        display: flex;
        padding-top: 10px;
        flex-direction: column;
        border-color: black;
        border-width: 1px;
        border-style: solid;
    }

    wcia-photo-carousel h2, h4 {
        margin-bottom: 0;
        margin-top: 0;
        margin-left: 10px;
    }

    wcia-photo-carousel .image-container {
        margin-top: 15px;
        flex: 1;
        background-color: black;
        background-size: contain;
        background-repeat: no-repeat;
        background-position: 50%;
    }

    wcia-photo-carousel button {
        cursor: pointer;
```

**Обработчик нажатия кнопки «Вперед»**

**Показывает текущую фотографию, установив фоновое изображение элемента контейнера**

**Назначает класс компонента тегу**

**Стилизация компонентов с помощью стилей CSS**

```
            background: transparent;
            border: none;
            font-size: 48px;
            color: white;
            position: absolute;
            top: 50%;
        }

        wcia-photo-carousel button.back {
            left: 10px;
        }

        wcia-photo-carousel button.forward {
            right: 10px;
        }

    </style>
</head>
    <body>
        <wcia-photo-carousel
                title="Future Vacation Photos"
                author="Ben Farrell"
                photos="images/fBmIASF.jpg,images/3zxD6rz.jpg,images/
                nKBgeLOr.jpg,images/yVjJZ1Yr.jpg">
        </wcia-photo-carousel>
    </body>
</html>
```

**Фотокарусель на нашей HTML-странице**

Теперь мы обратимся к рис. 2.14, чтобы увидеть окончательный вид нашего компонента.

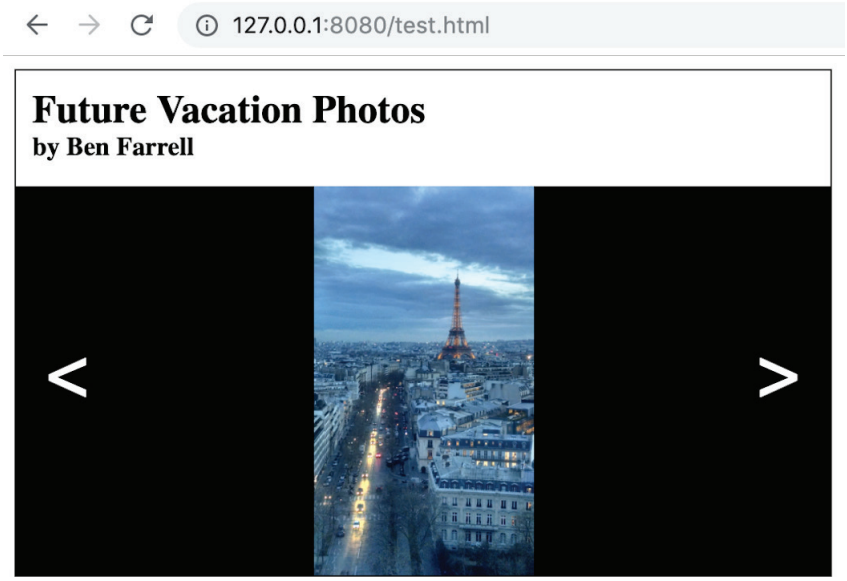



**Рис. 2.14 Готовый компонент фотокарусели**

### 2.6.9 Улучшение компонента

Несмотря на создание довольно полезного первого веб-компонента, есть множество способов, с помощью которых его можно улучшить. Самое главное – нам нужно упаковать нашу карусель как отдельный веб-компонент. В ее нынешнем виде использование ее в более крупном проекте вызвало бы путаницу с HTML, CSS и JS, встроенными прямо в основной HTML-код. В главе 5 мы подробно рассмотрим, как упаковать все это в один модуль JS.

Во-вторых, было бы здорово сделать наш компонент настраиваемым. Мы могли бы включать и выключать функции с помощью API или атрибутов в дополнение к нашему списку изображений. Мы рассмотрим их в главе 3.

Наконец, есть гораздо лучшие способы организовать наш HTML-код и таблицы стилей CSS, которые нужно включить в компонент, и даже защитить их от непреднамеренного изменения стилей и DOM. Шаблонные литералы и Shadow DOM будут рассмотрены позже.

## 2.7 Примечания относительно поддержки в браузерах

В начале этой главы я упоминал, что мы исключаем некоторые браузеры. Это происходит потому, что, хотя пользовательские элементы и поддерживаются в Chrome, Firefox и Safari, разработка Edge по-прежнему продолжается. Тем не менее вы можете использовать тестовую версию, предназначенную для разработчика. Я надеюсь, что скоро мы увидим финальную версию Edge.

Это просто-напросто оставляет IE без поддержки пользовательских элементов. К счастью, у нас есть для этого полифилы! Один из таких полифилов можно скачать здесь: https://unpkg.com/@webcomponents/custom-elements@1.2.4/custom-elements.min.js.

Кроме того, если у вас есть Node.js и вы можете использовать NPM, воспользуйтесь этим:

```
npm install @webcomponents/custom-elements
```

Какой бы метод вы ни выбрали, как только у вас появится полифил, просто включите этот файл сценария в свою страницу:

```
<script src="path/to/custom-elements.min.js"></script>
```

В дополнение к спецификации пользовательских элементов IE не поддерживает более новые функции языка JS, такие как классы. Не волнуйтесь, все это легко решаемо, но мы не будем вдаваться в подробности, пока не поговорим о процессах сборки. Если быть точным, я покажу вам способ перенести ваши версии ES2015/ES6 JS в ES5 JS, чтобы обеспечить поддержку в старых браузерах, или только в тех браузерах, которые еще не поддерживают их.

### Использование Custom Elements v1

В этой главе и на протяжении всей книги мы будем использовать спецификацию Custom Elements v1. Не волнуйтесь, это самая последняя и лучшая версия, и я сомневаюсь, что ее основы изменятся в ближайшие годы. Я упоминаю об этом, потому что версия 1 вышла довольно недавно, и поиск информации о веб-компонентах может сказать вам, что для создания пользовательского элемента следует использовать это:

```
document.registerElement('my-custom-tag', MyCustomTag);
```

Просто знайте, что в целом веб-компоненты претерпели недавние изменения и теперь в спецификации v1 они более устойчивы. Подробнее об этом и, в частности, о том, что изменилось, *см.* главу 4, в которой подробно описывается жизненный цикл компонента.

## Резюме

Из этой главе вы узнали:

- как теги, которые мы используем каждый день в базовой веб-разработке, наследуют от `HTMLElement` (даже если мы этого никогда не знали!);
- правила именования и использования вашего пользовательского элемента на странице (которые необходимы), а также стандартные правила (которым вы не обязаны следовать) для именования вашего элемента с помощью пространства имен;
- «попробовали на вкус» API веб-компонентов с помощью метода `connectedCallback`;
- как добавлять или настраивать `HTMLElement` с использованием распространенной методики объектно-ориентированного программирования, называемой *наследованием*, и создавать примеры, использующие наш новый пользовательский элемент.

# 3

# *Делаем так, чтобы ваш компонент можно было использовать повторно*

***Эта глава охватывает следующие темы:***

- использование методов получения и установки для работы с данными в вашем компоненте;
- применение метода `attributeChangedCallback` для прослушивания изменений в атрибуте;
- как определить, какие атрибуты прослушивать на предмет наличия изменений, используя метод `observedAttributes`;
- работа с атрибутами с использованием методов `hasAttribute()`, `getAttribute()` и `setAttribute()`.

В предыдущей главе мы подробно рассказали о простых способах создания вашего первого веб-компонента. В частности, мы рассмотрели создание вашего собственного пользовательского элемента и назначение ему некой минимальной пользовательской логики, чтобы ваш компонент работал определенным образом. Но что, если вы хотите, чтобы

ваш компонент работал по-разному в зависимости от того, какие параметры вы используете для его настройки? Что, если вы хотите, чтобы ваш компонент был адаптируемым? Обычно целью любой платформы, языка или фреймворка является создание повторно используемого кода, который можно просто настроить в соответствии с самым широким диапазоном вариантов использования.

Конечно, когда мы говорим, что хотим создавать повторно используемые и настраиваемые веб-компоненты, – это одно. Это почти бессмысленно, пока речь не будет идти о конкретном примере!

## 3.1 Реальный компонент

Один из моих недавних интересов – 3D для интернета. Меня особенно интересует, как виртуальная и дополненная реальность попадает в браузеры. Углубляться в изучение WebGL и Three.js или Babylon – это немного чересчур (и не по теме), но мы можем сделать что-нибудь простое, чтобы продемонстрировать возможности повторно используемых и настраиваемых компонентов.

### 3.1.1 Пример использования поиска в 3D

У 3D есть небольшая проблема с контентом. Мне нравится экспериментировать с решениями 3D для интернета, но я определенно не эксперт в создании 3D-моделей с использованием сложного программного обеспечения для 3D. В последнее время моя любимая вещь в виртуальной реальности – это стремительный рост инструментов для 3D-рисования и моделирования. Примечательно, что компания Google делает отличные вещи с помощью Blocks и TiltBrush, своих инструментов виртуальной реальности для моделирования и рисования в 3D. Более того, Google создала площадку, где авторы могут публиковать свои работы, под названием Poly.

Перейдя на сайт poly.google.com, вы можете просматривать и искать 3D-модели, выбирать любимые модели для использования их в своем приложении (многие из них можно свободно использовать и изменять). Что прекрасно подходит для наших целей, так это то, что у Poly есть API на базе REST, к которому мы можем подключиться и использовать для создания собственного веб-компонента трехмерного поиска! Опять же, подробно говорить только о 3D – это чересчур, особенно для книги по веб-компонентам, но все результаты, которые мы получаем, – это миниатюры изображений, поэтому нам не нужно усложнять процесс поиска и просмотра.

Как и во многих сервисах, таких как Poly, для доступа необходимо получить API-ключ. Если вы не хотите этого делать, вы по-прежнему можете изучать данный пример, потому что я предоставлю файл в формате JSON, который можно использовать вместо этого ключа, и вы можете запустить пример на собственном сервере.

Обо всем по порядку. Перейдите на страницу https://developers.google.com/poly/develop/web и следуйте инструкциям, чтобы получить ключ

API. Как только он у вас будет, поместите его в безопасное место, чтобы воспользоваться им позже.

## 3.1.2 Начнем с HTTP-запроса

Давайте теперь протестируем этот сервис и создадим HTTP-запрос в приведенном ниже листинге (в котором мы ищем попугая).

**Листинг 3.1 Создание HTTP-запроса к сервису Poly**

```
const url =
    'https://poly.googleapis.com/v1/assets?keywords=parrot&format=OBJ&key=
          <your_api_key>';
const request = new XMLHttpRequest();
request.open( 'GET', url, true );
request.addEventListener( 'load', (event) => {
   console.log(JSON.parse(
     event.target.response
   ));
});
request.send();
```

API для поиска в Poly (вставьте свой собственный API-ключ)

Создаем новый HTTP-запрос

Обратный вызов, где мы регистрируем ответ API

При выполнении этого кода вы должны увидеть все результаты, которые возвращаются прямо в консоли инструментов разработчика. Он также будет красиво отформатирован, учитывая, что мы преобразовали необработанный текст ответа обратно в формат JSON, как и предполагалось: `JSON.parse(event.target.response)`.

Когда мы посмотрим на вывод `console.log`, то увидим объект JSON, возвращенный сервисом. Конечно, со временем эти результаты изменятся, но тем не менее я вижу большое количество попугаев в результатах поиска! Это именно то, что мы указали в поиске по ключевому слову. Если мы расширим объект `assets` и посмотрим на возвращаемый массив 3D-моделей на рис. 3.1, то увидим, что у каждого ресурса есть объект миниатюры, который можно развернуть, чтобы посмотреть URL-адрес миниатюры. Этот URL-адрес – то, что мы и ищем!

Конечно, есть множество других данных, которые вы могли бы использовать, особенно если вы открыли массив «formats» для отображения фактических ссылок на 3D-объекты. Для наших целей мы просто будем использовать и отображать эти миниатюры.

## 3.1.3 Обертываем свою работу в пользовательский компонент

Давайте обернем HTTP-запрос, который мы только что сделали, в новый веб-компонент, который позволяет нам искать 3D-модели по ключевым словам и отображать результаты. Хотя усложнять не стоит. Нет необходимости перегружать каждый веб-компонент, чтобы делать слишком много, – мне нравится думать, что мы можем вести себя чрезвычайно детально с каждым компонентом, а для более крупных фрагментов функциональных возможностей мы можем объединить два или более компонентов. Вот почему мы не собираемся использовать ключевое сло-

во / поисковый ввод вне компонента. Наш веб-компонент будет отображать результаты поиска *только* на основе данных, которые мы передаем из ввода.

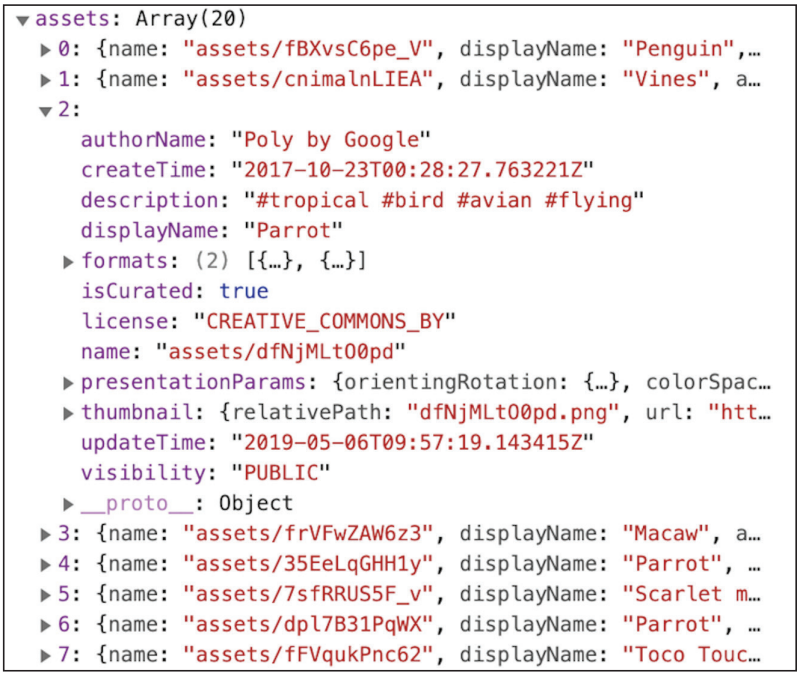



**Рис. 3.1 HTTP-ответ от Poly с 3D-моделями и детальной информацией о них**

Чтобы превратить наш фрагмент кода HTTP-запроса в веб-компонент, можно использовать то, что мы уже узнали о пользовательских элементах и методе `connectedCallback` из API веб-компонентов.

**Листинг 3.2 Создание веб-компонента из нашего HTTP-запроса**

```
<html>
<head>
    <meta charset="UTF-8">
    <title>Google Poly Search</title>
    <script>
        class PolySearch extends HTMLElement {
            connectedCallback() {
                this.doSearch();
            }

            doSearch() {
                const url =
      'https://poly.googleapis.com/v1/assets?keywords=parrot&format=OBJ&key=
            <your_api_key>';
```

Вызывает функцию поиска, когда добавляется компонент

```
                const request = new XMLHttpRequest();
                request.open( 'GET', url, true );
                request.addEventListener( 'load', (event) => {
                    console.log(JSON.parse( event.target.response ));
                });
                request.send();  <-- HTTP-запрос из последнего примера
            }
        }

        customElements.define(
'poly-search', PolySearch);  <-- Определяет наш компонент Poly search
    </script>
</head>

<body>
<poly-search></poly-search>  <-- Использует элемент Poly search на странице
</body>
</html>
```

Надеюсь, в этом листинге нет ничего из ряда вон выходящего. Я выделил фактический HTTP-запрос в метод `doSearch()`. Сейчас я вызываю его для метода `connectedCallback`, когда компонент добавляется в DOM. Поскольку у меня нет большого проекта, включающего множество компонентов в этом примере, я выбрал простое имя элемента, отражающее задачу, которую я выполняю: `poly-search`. Если бы я делал несколько компонентов для большого приложения, возможно, я бы назвал его как-нибудь вроде `<myappname-poly-search>`.

Вы, возможно, заметили, что сейчас наш компонент ищет только попугаев. Согласен, это не особо полезно. Однако сначала давайте покажем наши результаты. На рис. 3.2 показан наш компонент, обращающийся к API Google Poly и возвращающий список ресурсов, который затем отображает наш компонент.

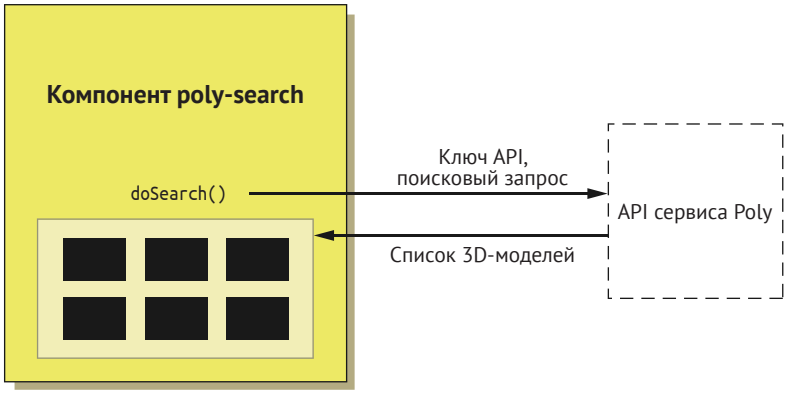



**Рис. 3.2 Наш пользовательский веб-компонент `poly-search` обращается к API Google Poly, используя API-ключ и поисковый запрос «parrot» (попугай). После чего мы возвращаемся к списку 3D-моделей и миниатюр**

### 3.1.4 Отображение результатов поиска

Мы можем начать с замены `console.log(JSON.parse(event.target.response));`, вызвав другой метод, который принимает все запрашиваемые нами 3D-модели:

```
this.renderResults(JSON.parse( event.target.response ).assets);
```

Затем, внутри нашего класса, мы добавим этот метод рендеринга для отображения всех миниатюр на нашей странице, как показано в приведенном ниже листинге.

**Листинг 3.3 Отображение результатов HTTP-запроса в нашем компоненте**

```
renderResults(assets) {
    let html = '';
    for (let c = 0; c < assets.length; c++) {
        html += '<img src="' + assets[c].thumbnail.url + '" width="200"
        height="150" />';
    }
    this.innerHTML = html;
}
```

Список результатов передается в нашу функцию рендеринга

Перебираем список результатов

Для каждого ресурса добавляем миниатюру изображения

После того как строка HTML собрана, добавляем все это в компонент

Все, что мы здесь делаем, – это перебираем наш массив ресурсов, захватываем URL-адреса миниатюр, превращаем его в источник элемента изображения и добавляем его в длинную строку HTML. Когда мы закончим, то добавим эту длинную HTML-строку в `innerHTML` нашего компонента.

Конечно, есть и другие способы сделать это, чтобы не создавать строк. Мы могли бы создавать новый элемент изображения с каждым циклом.

**Листинг 3.4 Альтернативный способ отображения результатов**

```
renderResults(assets) {
    for (let c = 0; c < assets.length; c++) {
        const img = document.createElement('img');
        img.src = assets[c].thumbnail.url;
        this.appendChild(img);
    }
}
```

Перебираем наш список результатов ресурсов таким же образом, как и прежде

Создаем элемент изображения каждый раз, вместо того чтобы использовать строку HTML

Добавляем каждый элемент в DOM, по одному за раз

В таких случаях мне лично больше нравится строковый подход. Вы можете создать большой кусок HTML-кода и одновременно заставить его обращаться к DOM, вместо того чтобы иметь по одному элементу на итерацию. Кроме того, HTML-код немного легче для чтения, особенно когда мы позже займемся шаблонными литералами. Большим недостатком создания каждого элемента по одному в цикле является то, что с каждым из них вы заставляете браузер повторно парсить и визуализировать весь этот блок. То же самое произошло бы, если бы вы добавляли каждое изображение по одному и каждый раз указывали бы значение для свой-

ства innerHTML. Вероятно, будет лучше придерживаться HTML-строки, которая формируется со временем, а затем установить все сразу в innerHTML.

### 3.1.5 Стилизация нашего компонента

Если вы запустите этот пример сейчас, то увидите несколько довольно больших изображений, расположенных вертикально в списке, как показано на рис. 3.3. Это не то, что нам непременно нужно для визуального отображения результатов, поэтому давайте уменьшим изображения и поместим их в красивые обертывающие строки, используя стили, как в приведенном ниже листинге.

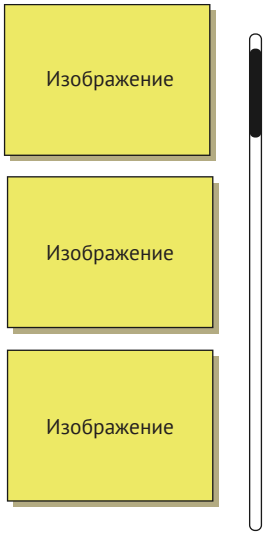



**Рис. 3.3 Наши результаты, полученные с сайта poly.google.com, до применения стилей. Изображения просто идут потоком вниз по странице, и мы вынуждены прокручивать ее, чтобы увидеть их все, потому что они слишком большие**

**Листинг 3.5 Стили для нашего компонента poly-search**

```
<style>
    poly-search {
        border-style: solid;
        border-width: 1px;
        border-color: #9a9a9a;
        padding: 10px;
        background-color: #fafafa;
        display: inline-block;
        text-align: center;
    }

    poly-search img {
        margin: 5px;
    }
</style>
```

Устанавливает красивую тонкую рамку вокруг всего нашего элемента

Разрыв между краями нашего элемента и внутренними результатами, которые мы отображаем

Цвет фона, который будет сочетаться с рамкой, отделяющей элемент от страницы

Позволяет элементам располагаться горизонтально и переходить на следующую строку, когда они выходят за пределы установленного пространства

Расстояние между изображениями

Для этого листинга я просто поместил стили в наш тег <head>, как вы обычно делаете при работе с CSS. Связывание стилей в рамках каждого

веб-компонента – это, безусловно, то, к чему мы придем позже, но сейчас пока не будем ничего усложнять.

Хотя мы уже нацеливаемся на наш элемент `poly-search` с помощью CSS-селектора, что совершенно допустимо! Когда вы создаете собственный пользовательский элемент, в действительности вы создаете элемент, который работает так же, как и любой другой элемент.

При запуске этого примера вы получите наиболее подходящее представление относительно того, что делает этот стиль, но рис. 3.4 – это визуальное приближение к тому, чего мы достигли, с последующим объяснением того, что мы сделали, используя стили.


**Рис. 3.4 Наша красиво стилизованная и центрированная сетка изображений. Изображения меньше по размеру, между ними правильное расстояние, и они располагаются на едва заметном белом фоне с сероватым оттенком в обрамлении серой рамки**

Это наш стилизованный пример в полном варианте:

**Листинг 3.6 Наш работающий веб-компонент со всеми стилями**

```
<html>
<head>
    <meta charset="UTF-8">
    <title>Google Poly Search</title>
    <script>
        class PolySearch extends HTMLElement {  <-- Определение веб-компонента
            connectedCallback() {
                this.doSearch();
            }

            doSearch() {  <-- Вызов функции поиска
                const url =
      'https://poly.googleapis.com/v1/assets?keywords=parrot&format=OBJ&key=
```

```
            <your_api_key>';
               const request = new XMLHttpRequest();
               request.open( 'GET', url, true );
               request.addEventListener( 'load', (event) => {
                   this.renderResults(JSON.parse
                     ( event.target.response ).assets);
               });
               request.send();
           }

           renderResults(assets) {  <-- Визуализируем результаты
               let html = '';
               for (let c = 0; c < assets.length; c++) {
                   html += '<img src="' + assets[c].thumbnail.url +
                   '" width="200" height="150" />';
               }
               this.innerHTML = html;
           }

       }
       customElements.define('poly-search', PolySearch);
   </script>

   <style>  <-- Стили компонента
       poly-search {
           border-style: solid;
           border-width: 1px;
           border-color: #9a9a9a;
           padding: 10px;
           background-color: #fafafa;
           display: inline-block;
           text-align: center;
       }

       poly-search img {
           margin: 5px;
       }
   </style>
</head>
<body>
<poly-search></poly-search>  <-- Использует компонент на странице
</body>
</html>
```

Теперь мы разобрались с основами, и у нас есть нечто, что работает визуально, но пока не очень полезно в качестве поискового компонента.

## 3.2 Делаем наш компонент настраиваемым

Давайте сейчас вернемся к нашей вопиющей проблеме и сути этой главы. Этот компонент нельзя использовать повторно. Во-первых, даже если бы я дал вам свой API-ключ, невозможно правильно установить его в компоненте. Во-вторых, мы все время ищем «попугаев». Невозможно пере-

дать этот поисковый запрос нашему компоненту, поэтому если кто-то из вашей команды использовал бы созданный вами компонент, он должен был бы зайти и напрямую изменить строку, содержащую URL-адрес:

```
const url =
    'https://poly.googleapis.com/v1/assets?keywords=parrot&format=OBJ&key=<y
    our_api_key>';
```

## 3.2.1 Создание API компонента с помощью устанавливающих методов

Давайте начнем с того, что разобьем эту строку. Мы сделаем это двумя различными способами, которые в конечном итоге будут дополнять друг друга. Первый метод, который мы рассмотрим, – это создание методов чтения (геттеров) и устанавливающих методов (сеттеров) API-ключа и поискового запроса.

Можно добавить этот листинг в наш класс.

**Листинг 3.7 Методы чтения и устанавливающие методы настраиваемых параметров нашего компонента**

```
set apiKey(value) {          <-- Сеттер для API-ключа
    this._apiKey = value;
    this.doSearch();
}

set searchTerm(value) {      <-- Сеттер для поискового запроса
    this._searchTerm = value;
    this.doSearch();
}
```

Без соответствующего метода чтения JS выдаст ошибку, если мы попытаемся прочитать или «получить» свойство. Тем не менее мы также можем легко создать его:

```
get searchTerm() {
   return this._searchTerm;
}
```

Пока, однако, геттеры на самом деле не нужны; нам просто нужно вставить поисковый запрос и переменные API-ключа *в* наш компонент, как показано на рис. 3.5.

Такое разбиение имеет смысл. Скорее всего, вам нужно будет настроить API-ключ только один раз, но поскольку пользователь продолжает искать разные вещи, поисковый запрос будет довольно часто обновляться.

## 3.2.2 Используя наш API извне

Имея код, приведенный в листинге 3.7, когда мы установим это свойство извне, будет запущена функция. В связи с этим, если бы вы не знали код этого класса, вы бы подумали, что работаете с простой переменной,

благодаря нашим устанавливающим методам. Вы также могли заметить, что я использую символ нижнего подчеркивания (_) в именах своих переменных. Это не означает ничего особенного, но поскольку в JavaScript нет понятия «закрытых» переменных (кроме потрясающей новой функции полей класса в последней версии Chrome) или переменных, к которым у вас нет доступа вне вашего класса, я использую нижнее подчеркивание, чтобы указать на то, что мы не планируем осуществлять доступ к этим переменным извне. Использование нижнего подчеркивания для кого-то может быть предметом спора и рассматривается как устаревшая практика. Если вы хотите глубже изучить эту концепцию, обратитесь к приложению. Независимо от этого в данном случае `_searchTerm` – наша внутренняя переменная, которую мы используем, а `searchTerm` – устанавливающий метод этой переменной.

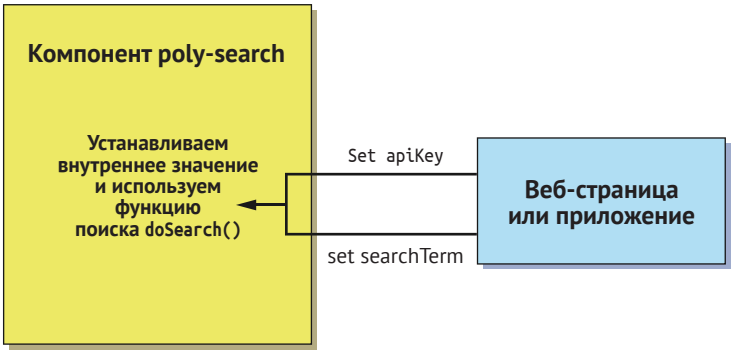



**Рис. 3.5 Использование сеттеров для нашего компонента извне позволяет нам выполнять логику и задавать значение, при этом API-интерфейс компонента будет оставаться простым**

Используя сеттер, мы не просто устанавливаем свойство `searchTerm`. Когда вы устанавливаете его вне нашего класса компонентов, это то, как он выглядит для пользователя API нашего компонента. Используя устанавливающий метод, мы вводим некую логику, чтобы установить это внутреннее свойство и запустить наш метод `doSearch()` для выполнения HTTP-запроса.

Теперь, если вы хотите написать какой-то код JS в теге скрипта вне класса компонента, можно написать следующее, чтобы сначала выбрать компонент, а затем установить каждое свойство (конечно, только после того как компонент будет правильно создан):

```
document.querySelector('poly-search').apiKey = '<your_api_key>';
document.querySelector('poly-search').searchTerm = 'parrot';
```

Безусловно, если бы мы запустили поиск без API-ключа или поискового запроса, то потерпели бы неудачу, поэтому в приведенном ниже листинге мы можем обернуть наш метод поиска в оператор `if`, чтобы убедиться, что обе переменные присутствуют, перед тем как начать поиск.

**Листинг 3.8 Оборачивание метода поиска с использованием оператора `if`**

```
doSearch() {
    if (this._apiKey && this._searchTerm) {
        const url = 'https://poly.googleapis.com/v1/assets?keywords=' +
          this._searchTerm + '&format=OBJ&key=' + this._apiKey;
        const request = new XMLHttpRequest();
        request.open( 'GET', url, true );
        request.addEventListener( 'load', (event) => {
            this.renderResults(JSON.parse( event.target.response ).assets);
        });
        request.send();
    }
}
```

Проверяем наличие API-ключа и поискового запроса

Предоставление нашим компонентам API-интерфейса, подобного этому, – неплохое упражнение, но для этого конкретного случая использования существует другой метод передачи данных: атрибуты. Мы постоянно применяем атрибуты в веб-разработке. Фактически атрибут `src` для установки URL-адреса миниатюр в каждом изображении является лишь одним из примеров. Даже обычная установка стилей элемента с помощью слова `class` или ссылки `href` для тега ссылки является примером атрибута.

## 3.3 Использование атрибутов для конфигурирования

Использование атрибутов в веб-компонентах настолько очевидно, что вы можете игнорировать их в пользу подхода, при котором используются методы чтения и устанавливающие методы. Мы так часто используем атрибуты, что, возможно, и не думаем о них как о чем-то, что можно применять для внутренней работы вашего веб-компонента.

### 3.3.1 Аргумент против API компонента

При использовании подхода с методами чтения и устанавливающими методами возникает некоторая сложность, которая на самом деле не нужна. С одной стороны, оборачивание метода поиска с помощью конструкции `if/then`, для того чтобы проверить, установлены ли `apiKey` и `searchTerm` -, – хорошая практика, когда разработчик забывает установить то одно, то другое, но было бы хорошо, если бы оба свойства были доступны сразу, когда компонент используется по назначению.

Еще одна неприятность заключается в применении JS для установки этих свойств. Если бы эти свойства были атрибутами в теге HTML, нам не пришлось бы устанавливать `apiKey` и `search-Term` в двух отдельных строках. В более сложных приложениях может быть трудно отследить, где вы установили их в коде. Кроме того, могут возникнуть проблемы с синхронизацией с вашим компонентом. Возможно, ваш компонент еще не создан должным образом, когда вы вызываете эти сеттеры. Если такое произойдет, ваши значения, возможно, будут просто потеряны!

Определенно, это решаемые проблемы, но сейчас давайте сосредоточимся на атрибутах.

## 3.3.2 Реализация атрибутов

Давайте немного изменим ситуацию. Во-первых, избавимся от наших сеттеров и кода JS, необходимого для того, чтобы использовать их. Нам они не нужны. Далее мы добавим наши атрибуты в тег пользовательского элемента:

```
<poly-search apiKey="<your_api_key>"
             searchTerm="parrot">
</poly-search>
```

Теперь заменим кое-какой код JS, чтобы использовать атрибуты вместо наших переменных. Давайте сохраним проверку с использованием конструкции `if/then` в приведенном ниже листинге на случай, если пользователь нашего компонента забудет использовать тот или иной атрибут.

**Листинг 3.9 Использование атрибутов для настраиваемых параметров в нашем методе поиска**

```
doSearch() {
    if (this.getAttribute('apiKey') && this.getAttribute('searchTerm')) {
        const url = 'https://poly.googleapis.com/v1/assets?keywords=' +
      this.getAttribute('searchTerm') + '&format=OBJ&key=' +
      this.getAttribute('apiKey');
        const request = new XMLHttpRequest();
        request.open( 'GET', url, true );
        request.addEventListener( 'load', (event) => {
            this.renderResults(JSON.parse( event.target.response ).assets);
        });
        request.send();
    }
}
```

Используем атрибуты вместо свойств для параметров конфигурации

Наконец, поскольку атрибуты становятся доступны сразу после создания элемента, мы можем сразу же выполнить начальный поиск, когда наш компонент добавлен в DOM, используя метод `connected-Callback`:

```
connectedCallback() {
   this.doSearch();
}
```

Для краткости я опускаю стили, когда мы рассматриваем текущее состояние нашего компонента в следующем листинге.

**Листинг 3.10 Наш полный (за исключением стилей) пример компонента с использованием атрибутов**

```
<html>
<head>
    <title>Google Poly Search</title>
```

```
<script>
    class PolySearch extends HTMLElement {
        connectedCallback() {
            this.doSearch();
        }

        doSearch() {
             if (this.getAttribute('apiKey') &&
             this.getAttribute('searchTerm')) {
                const url =
  'https://poly.googleapis.com/v1/assets?keywords=' +
  this.getAttribute('searchTerm') + '&format=OBJ&key=' +
  this.getAttribute('apiKey');
                const request = new XMLHttpRequest();
                request.open( 'GET', url, true );
                request.addEventListener( 'load', (event) => {
                    this.renderResults(
                             JSON.parse( event.target.response ).assets);
                   });
                request.send();
            }
        }

        renderResults(assets) {
            let html = '';
            for (let c = 0; c < assets.length; c++) {
                html += '<img src="' + assets[c].thumbnail.url +
                '" width="200" height="150" />';
            }
            this.innerHTML = html;
        }
    }

    customElements.define('poly-search', PolySearch);
</script>
</head>

<body>
<poly-search apiKey="<your_api_key>"
searchTerm="parrot">
</poly-search>

</body>
</html>
```

После добавления компонента запускается функция поиска

Если заданы поисковый запрос и API-ключ, добавляем их в конечную точку поиска

Отправка HTTP-запроса

Добавляем элемент изображения в HTML-строку для каждого ресурса

Устанавливаем значение в виде сгенерированной строки

Объявляем компонент на странице с API-ключом и поисковым запросом

Теперь компонент выглядит довольно работоспособным, но того, что мы сделали, пока еще недостаточно. Этот поисковый запрос, вероятно, будет часто меняться; нам нужно будет отслеживать изменения.

## 3.3.3 Чувствительность к регистру символов

Обратите внимание, что хотя я использовал прописную букву «K» в `apiKey` и прописную букву «T» в `search-Term`, сами атрибуты не чувствительны к регистру. Мы могли бы полностью переписать наш тег, как показано

ниже, и это не повлияло бы на вещи вообще (хотя есть веская причина сохранить все в нижнем регистре, о чем мы поговорим чуть позже):

```
<poly-search apikey="<your_api_key>"
             searchterm="parrot">
</poly-search>
```

## 3.4 Прослушивание изменений в атрибутах

Однако есть еще одна проблема, связанная с нашим вариантом использования. Это правда, что наш API-ключ, скорее всего, никогда не изменится в нашем веб-приложении, но нам нужно, чтобы пользователи вводили текст и что-то искали. Прежде чем приступить к решению этой проблемы, давайте создадим типичный текстовый ввод, который позволяет пользователю вводить поисковый запрос. Данный аспект находится за пределами нашего веб-компонента, поэтому это не урок по веб-компонентам как таковым, а просто нечто, что поможет нам продемонстрировать и решить нашу проблему с атрибутами.

### 3.4.1 Добавление поля ввода текста

Учитывая это, давайте изменим содержимое нашего тега `<body>`.

**Листинг 3.11 Поле для ввода текста для нашего компонента**

```
<body>
    <label>Enter search term: </label>
    <input type="text" onchange="updatePolySearch(event)" />
    <br /><br />

    <script>
       function updatePolySearch(event) {
           document.querySelector('poly-search').setAttribute('searchterm',
             event.target.value);
       }
    </script>

    <poly-search apikey="<your_api_key>" searchterm="parrot">
```

Сейчас мы добавили поле для ввода текста со слушателем событий `onchange`. Перед этим у нас идет тег `<label>`, просто чтобы объяснить в нашем пользовательском интерфейсе, что фактически делает это поле. Обычно я не использую в теге встроенный код JS, подобный этому, но для такой простой демонстрации проще показать это таким образом. Событие `onchange` происходит только тогда, когда пользователь «отправляет» текст, то есть когда он нажимает клавишу **Enter**.

Функция, которую оно вызывает, `updatePolySearch`, захватывает отправляемое событие, которое включает в себя *цель* или какой элемент отправил событие. Мы можем запросить `event.target.value`, чтобы получить новый поисковый запрос, введенный пользователем. Оттуда можем установить атрибут `searchterm` нашего веб-компонента.

Не стесняйтесь опробовать это прямо сейчас! Если вы откроете инструменты разработки в своем браузере, чтобы увидеть, как выглядят элементы на странице, то увидите, что наш атрибут `<poly-search> searchterm` меняется в реальном времени после того, как мы изменили наше поле для текстового ввода.

К сожалению, простое обновление атрибута не приводит к повторному запуску поиска и обновлению наших результатов. Мы должны сделать это сами, что приводит нас ко второму методу жизненного цикла веб-компонента: `attributeChangedCallback`. Нашим первым методом жизненного цикла был `connectCallback`, но теперь мы готовы пойти дальше.

## 3.4.2 Метод attributeChangedCallback

Метод `attributeChangedCallback` похож на любой другой метод жизненного цикла веб-компонента. Вы просто добавляете метод в свой класс, чтобы переопределить пустой метод `HTMLElement`, и он будет запущен при изменении атрибута.

Этот метод принимает три параметра: имя атрибута, который изменился, старое значение атрибута и его новое значение:

```
attributeChangedCallback(attrName, oldVal, newVal)
```

Давайте интегрируем это в наш веб-компонент и посмотрим, что произойдет. Здесь я буду немного зловредным, но предупреждаю вас заранее. Мы интегрируем этот метод, но он не сработает из-за одной недостающей детали, которую я объясню позже.

Первое, что нужно сделать, – это избавиться от метода `connectedCallback` в нашем классе. Мы делаем это потому, что в нашем конкретном случае наш метод `connectedCallback` инициирует поиск. Тем не менее теперь метод `attributeChangedCallback` фактически также будет делать это. Говоря технически, наш атрибут *действительно* меняется, превращаясь в нечто, когда запускается наш компонент, поэтому запускается `attributeChangedCallback`. Кроме того, у нас нет никакой логики, чтобы отменить наш HTTP-запрос перед его повторным запуском в нашем компоненте – для простоты и отсутствия ошибок, когда обе эти функции обратного вызова срабатывают практически одновременно, давайте просто избавимся от `connectedCallback`.

Далее добавим наш метод `attributeChangedCallback`.

**Листинг 3.12 Метод `attributeChangedCallback`, используемый для прослушивания изменений в `searchterm`**

```
attributeChangedCallback(name, oldval, newval) {
   if (name === 'searchterm') {
       this.doSearch();
   }
}
```

Здесь обратный вызов действительно выглядит просто. Если изменяемое имя атрибута – `searchterm`, снова запустите поиск. Этот аспект чувствителен к регистру. Имя будет всегда в нижнем регистре. Это может

немного сбивать с толку, если вы пишете свой атрибут в HTML в стиле camel case («верблюжий регистр»), а затем будете писать здесь имя таким же образом. Во избежание путаницы лучше всегда писать атрибуты в нижнем регистре.

Когда я писал эти строки, то случайным образом все усложнил, прежде чем обнаружил это. Изначально я написал следующий код:

```
attributeChangedCallback(name, oldval, newval) {
   if (name === 'searchterm' && oldval !== newval ) {
       this.doSearch();
   }
}
```

Я подумал, что мне нужно вызывать поиск, только если старое значение отличалось от нового. Нет смысла перезапускать поиск и тратить сетевой запрос, если значение не меняется, верно? Что же, если значение не изменилось, данный метод не нужно вызывать в первую очередь, поэтому выполнять этот дополнительный шаг излишне.

Теперь, когда мы зафиксировали изменения атрибутов и приняли меры, когда они меняются, все должно работать, верно? Еще нет! Тут я упустил одну маленькую деталь, касающуюся того, как работает этот метод. Прежде чем объясню, о чем идет речь, немного истории.

### 3.4.3 Атрибуты, за которыми ведется наблюдение

В начале этой главы я немного рассказал о том, как распространенные атрибуты связаны со всем, что мы делаем в HTML. У каждого элемента есть множество потенциальных атрибутов, которые он может использовать и которые на самом деле что-то значат. Как минимум у элементов, вероятно, всегда будет элемент `class` для стилей. И конечно же, мы можем придумать любой атрибут, который захотим. Когда эти потенциальные атрибуты повсюду, при вызове метода `attributeChangedCallback` каждый раз, когда что-то меняется, если нас не волнуют эти изменения, будет тратиться много времени на выполнение кода.

В версии 0 API веб-компонентов с методом `attributeChangedCallback` происходило именно это: он вызывался каждый раз, когда менялось что-то общее, как, например, атрибут `class`. Первые пользователи веб-компонентов считали это несколько раздражающим и нерациональным. Поэтому теперь, в версии 1 этого API, мы должны указать нашему компоненту, что конкретно нужно слушать.

**Листинг 3.13 Сообщаем нашему компоненту, за изменениями каких атрибутов следует наблюдать**

```
static get observedAttributes() {
   return ['searchterm'];
}
```

Если вы незнакомы с ключевым словом `static` для метода класса, обратитесь к приложению. Говоря кратко, это метод, который вызывается для определения класса, а не для созданного экземпляра.

В этом статическом методе мы установили для нашего геттера observedAttributes массив, содержащий searchterm. Если бы мы хотели, чтобы наблюдение велось за бóльшим количеством атрибутов, можно было бы просто добавить в массив больше элементов:

```
static get observedAttributes() {
   return ['searchterm', 'apikey', 'anotherthing', 'yetanotherthing' ];
}
```

После добавления этого последнего фрагмента в наш пример в листинге 3.14 наш код должен заработать. Этот новый код, используемый для наблюдения за нашим атрибутом searchTerm, изображен на рис. 3.6. Теперь мы автоматически загружаем результаты с первым поисковым запросом «попугай», но когда пользователь отправляет другие запросы, результаты обновляются.

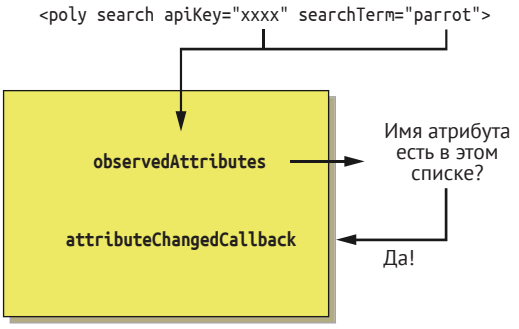



**Рис. 3.6 Прежде чем метод attributeChangedCallback будет запущен внутри вашего компонента в результате изменения атрибута в разметке вашего компонента, имя этого атрибута должно быть в списке observedAttributes**

**Листинг 3.14 Полный компонент с атрибутами**

```
<html>
<head>
    <title>Google Poly Search</title>
    <script>

        class PolySearch extends HTMLElement {          <-- Класс компонента
            static get observedAttributes() {
                return ['searchterm'];                  <-- Атрибут, находящийся под наблюдением
            }

            attributeChangedCallback(name, oldval, newval) {
                if (name === 'searchterm') {
                    this.doSearch();                    <-- Когда атрибут, находящийся под наблюдением,
                }                                           изменяется, запускается поисковый запрос
            }

            doSearch() {                                <-- Поисковый запрос, который
                if (this.getAttribute('apiKey') &&          использует API-ключ
      this.getAttribute('searchTerm')) {                    и поисковый запрос...
                    const url =
      'https://poly.googleapis.com/v1/assets?keywords=' +
      this.getAttribute('searchTerm') + '&format=OBJ&key=' +
```

```
      this.getAttribute('apiKey');
                  const request = new XMLHttpRequest();
                  request.open( 'GET', url, true );
                  request.addEventListener( 'load', (event) => {
                      this.renderResults(JSON.parse
                        ( event.target.response ).assets);
                  });
                  request.send();
              }
          }

          renderResults(assets) {   <-- Визуализирует все 3D-модели
              let html = '';
              for (let c = 0; c < assets.length; c++) {
                  html += '<img src="' + assets[c].thumbnail.url +
                   '" width="200" height="150" />';
              }
              this.innerHTML = html;
          }
      }

      customElements.define(
            'poly-search', PolySearch);   <-- Отображает имя тега в класс компонента
  </script>

  <style>   <-- Стили компонента
      poly-search {
          border-style: solid;
          border-width: 1px;
          border-color: #9a9a9a;
          padding: 10px;
          background-color: #fafafa;
          display: inline-block;
          text-align: center;
      }

      poly-search img {
          margin: 5px;
      }
      input {
          font-size: 18px;
      }
  </style>
</head>

<body>

<label>Enter search term: </label><input type="text"
       onchange="updatePolySearch(event)" />   <-- Поле ввода, чтобы пользователь мог ввести поисковый запрос
<br /><br />

<script>
   function updatePolySearch(event) {   <-- При изменении поля ввода устанавливает атрибут searchTerm для нашего компонента
       document.querySelector('poly-search').setAttribute('searchTerm',
         event.target.value);
   }
```

```
</script>
<poly-search apikey="<your_api_key>"
             searchterm="parrot">
</poly-search>

</body>
</html>
```

Компонент, добавленный на страницу с установленным API-ключом и начальным поисковым запросом

Благодаря этому мы позволили нашему компоненту реагировать на изменения. Для нас не имеет смысла реагировать на изменения API-ключа, потому что этот ключ, как правило, никогда не меняется. Поисковый запрос будет постоянно меняться, поэтому нам определенно нужен был способ отреагировать на это.

## 3.5 Делаем другие вещи еще более настраиваемыми

Теперь давайте поэкспериментируем с настройками! Мы можем применить ряд стилей, например установить размер изображения и цвет фона компонента.

### 3.5.1 Использование метода hasAttribute для проверки существования атрибута

В листинге 3.15 я веду себя как ленивый разработчик. Я не ожидаю, что размеры изображения или цвет фона нужно будет менять во время выполнения – а только тогда, когда мы изначально пишем HTML-код. Поэтому я не слушаю изменения в атрибутах; вместо этого просто устанавливаю эти свойства, когда компонент добавляется в DOM.

**Листинг 3.15 Добавление атрибутов размера и цвета фона**

```
connectedCallback() {
    if (this.hasAttribute('thumbheight')) {
        this._thumbheight = this.getAttribute('thumbheight');
        this._thumbwidth = (this.getAttribute('thumbheight') *
          1.3333 /*aspect ratio*/);
    } else {
        this._thumbheight = 150;
        this._thumbwidth = 200;
    }

    if (this.hasAttribute('backgroundcolor')) {
        this.style.backgroundColor = this.getAttribute('backgroundcolor');
    }
}
```

Если установлен атрибут thumbheight, он используется для определения размера изображения и вычисления ширины

В противном случае используются значения по умолчанию / жестко заданные значения

Если установлен атрибут backgroundcolor, сразу же настраивается стиль компонента

Я также не заставляю пользователя компонента иметь эти атрибуты. Вместо этого проверяю, использовал ли разработчик атрибут в своей

разметке с помощью метода `hasAttribute`, и, если это так, задаю эти свойства. Если нет, у нас есть альтернативные значения либо с использованием JS для размера, либо с применением уже существующего стиля для цвета фона.

Чтобы использовать свойства размера, я отредактировал метод рендеринга изображений, как показано в этом листинге.

**Листинг 3.16 Рендеринг наших миниатюр с настраиваемыми размерами**

```
renderResults(assets) {
    let html = '';
    for (let c = 0; c < assets.length; c++) {
        html += '<img src="' + assets[c].thumbnail.url + '" width="' +
         this._thumbwidth + '" height="' +
         this._thumbheight + '"/>';
    }
    this.innerHTML = html;
}
```

Используем свойства height и width для управления размером изображения

Поскольку мы добавили настройку с применением стилей, у вас, вероятно, разыгралось воображение! Конечно, мы могли бы настроить рамки, интервалы и т. д. Есть еще одна вещь, которую мы настроим, а именно конечная точка поиска.

## 3.5.2 Полная настройка URL-адреса HTTP-запроса для разработки

Это также тот момент, когда я порадую читателей, которые не захотели регистрироваться, чтобы получить API-ключ. В приведенном ниже листинге мы разобьем URL-адрес HTTP-запроса. Мы сделаем это, выделив основу URL-адреса, а также формат 3D-объекта.

**Листинг 3.17 Мы разбиваем наш URL-адрес HTTP-запроса, чтобы сделать его еще более настраиваемым**

```
doSearch() {
    if (this.getAttribute('apiKey') && this.getAttribute('searchTerm')) {
        const url = this.getAttribute('baseuri') +
'?keywords=' + this.getAttribute('searchTerm') + '&format=' +
this.getAttribute('format') + '&key=' + this.getAttribute('apiKey');
        const request = new XMLHttpRequest();
        request.open( 'GET', url, true );
        request.addEventListener( 'load', (event) => {
            this.renderResults(JSON.parse( event.target.response ).assets);
        });
        request.send();
    }
}
```

Добавляем базовый URI в качестве настраиваемого параметра, чтобы разрешить вызов другого сайта

С помощью приведенного ниже тега мы можем начать использовать все наши параметры настройки.

**Листинг 3.18 Добавляем атрибут `baseuri` в тег компонента**

```
<poly-search apikey="<your_api_key>"
             format="OBJ"
             thumbheight="50"
             backgroundcolor="red"
             baseuri=
               "https://poly.googleapis.com/v1/assets"
             searchterm="parrot">
</poly-search>
```

Определяет конечную точку поиска в атрибутах компонента

Теперь мы можем настроить атрибут `baseuri` на что-то другое. Конечно, у разных поисковых сервисов будут разные API и форматы результатов, но мы можем протестировать нашу настройку без Google, указав путь к файлу в формате JSON, который мы размещаем:

```
baseuri="http://localhost:8080/assets.json"
```

Конечно, будут отличия, в зависимости от того, как вы настроили свой сервер разработки (это может быть localhost либо что-то другое, а порт 8080 является общим, но он сильно отличается в зависимости от ваших настроек).

## 3.5.3 Руководство по передовым методикам

Поскольку теперь мы рассмотрели методы чтения, устанавливающие методы и атрибуты для работы с данными, встает вопрос: что из них следует использовать? На самом деле все зависит от вас, но существует ряд новых передовых методик. Пока слишком рано принимать эти передовые практики как обязательные к исполнению, но здесь есть несколько хороших идей, особенно если вы собираетесь использовать свои компоненты совместно с другими людьми. Один из ресурсов – это неполный рабочий проект: https://github.com/webcomponents/goldstandard/wiki. Google также опубликовала несколько передовых методик: https://developers.google.com/web/fundamentals/web-components/best-practices.

## 3.5.4 Избегайте использования атрибутов для расширенных данных

В руководстве по веб-компонентам Google есть несколько рекомендаций для атрибутов. Одна из таких рекомендаций – не использовать атрибуты для расширенных данных, таких как массивы и объекты.

Допустим, например, что у вас очень сложное приложение, а для некоторых ваших веб-компонентов установка безумно сложна. Возможно, у вас есть 50 или более свойств, которые можно использовать для конфигурирования, или же ваши данные конфигурации должны быть представлены в виде вложенной структуры:

```
{
    Tree: {
        Branches: [
```

```
            { branch: {
               leaves: [
                { leaf: "leaf"},
                { leaf: "leaf"},
                { leaf: "leaf"},
                  ]
                }
            }
        ]
    }}
```

В любом случае, выделение этих свойств для отдельных атрибутов было бы обременительно или невозможно.

На самом деле можно преобразовать объект JSON в строку и вставить ее в атрибут нашего тега:

```
<my-element data="{"Tree": {"Branches": [{"branch": {"leaves": [{"leaf":
    "leaf"},{"leaf": "leaf"},{ "leaf": "leaf"}]}}]}}
    " my-element>
```

Однако, вероятно, проще сделать это с помощью кода:

```
myElement.setAttribute('data', JSON.stringify(data));
```

Чтобы извлечь данные, вам нужно будет сериализовать эту строку в JSON:

```
JSON.parse(this.getAttribute('data'));
```

Однако в конце, когда у вас в DOM появится эта массивная, пугающего вида строка, ваши инструменты разработки станут намного сложнее для чтения, создавая препятствия для понимания структуры DOM. В этом случае, возможно, лучше использовать метод или сеттер, чтобы передать свои данные в компонент и избежать использования атрибутов для расширенных данных.

### 3.5.5 Отражение свойств и атрибутов

Еще одна рекомендация, предложенная Google, – выполнить то, что называется *отражением* ваших атрибутов и свойств. Отражение – это практика использования геттеров и сеттеров, а также и атрибутов ваших данных, при которой они всегда синхронизированы друг с другом. Особенно при передаче вашего компонента другим разработчикам или его совместного использования с другими людьми пользователи могут ожидать согласованного API-компонента.

С атрибутами, как правило, легче работать при написании HTML-кода, в то время как с помощью кода JS настройка свойств компонента более краткая и простая в использовании. Другими словами, разработчики JS предпочтут написать `yourcomponent.property = 'something';` и, вероятно, вряд ли напишут `yourcomponent.setAttribute('property', 'something');`. В то же время тот, кто пишет HTML-код, предпочел бы просто установить атрибут в разметке.

Если эти два метода не делают одно и то же или один из них поддерживается, а другой нет, это может привести в замешательство пользователя вашего компонента. Вот почему при установке свойства через JS соответствующий атрибут должен меняться в элементе, и наоборот. Когда атрибут изменяется, получение свойства после этого должно отражать новейшее значение.

Одна из ловушек, которую Google определила в своем руководстве, – это использование метода `attribute-ChangedCallback` для обновления сеттера. Google называет это *реентерабельностью*; вот как это реализуется.

**Листинг 3.19 Подводный камень для размышлений из руководства по передовым методам работы с веб-компонентами Google**

```
// Когда атрибут [checked] изменится, установите свойство checked для соответствия;
attributeChangedCallback(name, oldValue, newValue) {
  if (name === 'checked')
    this.checked = newValue;
}

set checked(value) {
  const isChecked = Boolean(value);
  if (isChecked)
    // Упс! Это приведет к запуску бесконечного цикла, потому что будет инициирован
    // метод attributeChangedCallback(), который затем снова устанавливает
    // это свойство.
    this.setAttribute('checked', '');
  else
    this.removeAttribute('checked');
}
```

Когда атрибут изменяется, вызывается сеттер

При вызове сеттера атрибут обновляется, приводя к появлению бесконечного цикла

В этом примере, который я взял непосредственно из документации Google для разработчиков, возникает бесконечный цикл. Тут используется сеттер. Он устанавливает атрибут, но это приводит к запуску `attributeChangedCallback`, который снова использует сеттер, который затем изменяет атрибут... Вы уловили суть – это бесконечный цикл, и поток можно увидеть на рис. 3.7.

Возможно, более подходящий способ – использовать атрибут в качестве так называемого «источника истины». Я добавил отражение к свойству `searchTerm` в нашем примере поиска по сайту Google Poly, с дополнительным геттером и сеттером, как показано в приведенном ниже листинге.

**Листинг 3.20 Добавление геттера и сеттера в дополнение к существующим атрибутам для отражения**

```
static get observedAttributes() {
    return ['searchterm'];
}

get searchTerm() {
    return this.getAttribute('searchTerm');
}

set searchTerm(val) {
```

Геттер просто вернет доступ и вернет атрибут

```
    this.setAttribute('searchTerm', val);
}

attributeChangedCallback(name, oldval, newval) {
    if (name === 'searchterm') {
        this.doSearch();
    }
}
```

Сеттер установит атрибут

При настройке запускается attributeChangeCallback и начинается поиск

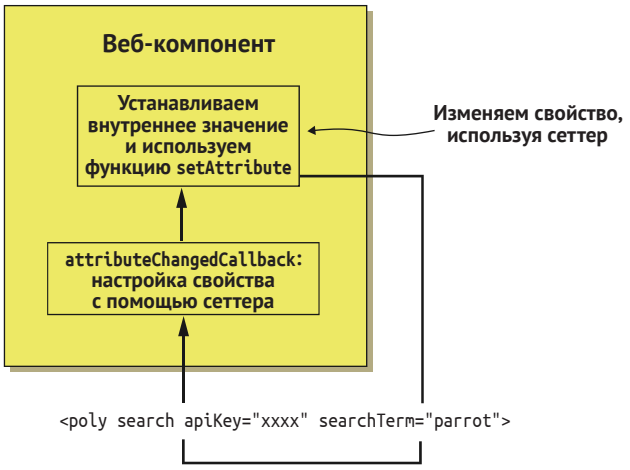



**Рис. 3.7 Реентерабельность – плохой способ для реализации отражения свойств или атрибутов. Установка атрибута при использовании геттера приводит к запуску `attributeChangedCallback`, который затем может установить это свойство снова, продолжая бесконечный цикл**

В этом примере наш метод получения просто возвращает текущий атрибут, а наш устанавливающий метод устанавливает его. Конечно, существуют дополнительные способы достижения рефлексии, но важный вывод заключается в том, что если вы хотите максимизировать опыт разработчика в случае с вашим компонентом, сделайте так, чтобы ваши атрибуты и свойства оставались согласованными и синхронизированными друг с другом!

## 3.6 Обновление компонента-ползунка

Теперь, когда мы понимаем, как работать с атрибутами, чтобы создать повторно используемый компонент, и знаем, как использовать отражение атрибутов в своих интересах, пришло время обновить компонент-ползунок из предыдущей главы, чтобы сделать его интерактивным и реагирующим на атрибуты, которые мы ему предоставляем, или свойства JS, которые мы для него устанавливаем. Сейчас наш класс компонента довольно тонкий, особенно после того, как мы переместили все стили в тег `<style>`. Все, что он делает, – это отображает HTML-код (два тега `<div>`); в приведенном ниже листинге показан ползунок без длинного списка стилей CSS.

**Листинг 3.21 Компонент-ползунок (без стилей)**

```
<html>
<head>
    <title>Slider</title>

    <script>
        class Slider extends HTMLElement {
            connectedCallback() {
                this.innerHTML = '<div class="bg-overlay"></div>
                <div class="thumb"></div>';
            }
        }

        if (!customElements.get('wcia-slider')) {
            customElements.define('wcia-slider', Slider);
        }
    </script>

    <style><!-- CSS was here --></style>
</head>
<body>
    <wcia-slider></wcia-slider>
</body>
</html>
```

Напомним, что мы временно использовали два свойства для управления некоторыми функциями компонента или, другими словами, его API. Давайте оформим этот API и перечислим здесь эти свойства:

- `value` – текущее процентное значение ползунка от 0 до 100;
- `backgroundcolor` – шестнадцатеричный код цвета самого верхнего фонового слоя.

Теперь, когда мы определили эти свойства, мы можем сделать две вещи. Во-первых, прослушать эти атрибуты на предмет наличия изменений. Мы будем добавлять все эти функции прямо в класс `Slider`.

**Листинг 3.22 Прослушивание изменений в атрибутах**

```
static get observedAttributes() {
    return ['value', 'backgroundcolor'];
}
attributeChangedCallback(name, oldVal, newValue) {
    switch (name) {
        case 'value':
            this.refreshSlider(newValue);
            break;

        case 'backgroundcolor':
            this.setColor(newValue);
            break;
    }
}
```

Прослушиваем изменения в атрибутах value и backgroundcolor

Реагируем на изменения в значении ползунка, если оно установлено снаружи компонента

Реагируем на изменение цвета фона

Второе, что нужно сделать, – связать эти атрибуты с надлежащим JS API, используя рефлексию, с которой мы только что познакомились. Когда одно из этих свойств задается с помощью сеттера JS, атрибут обновляется в компоненте. Аналогично, когда атрибут установлен для тега, это значение можно получить с помощью соответствующего геттера. В приведенном ниже листинге показана рефлексия в нашем компоненте для двух этих атрибутов.

**Листинг 3.23 Геттеры и сеттеры для свойств `backgroundcolor` и `value`**

```
set value(val) {
    this.setAttribute('value', val);
}

get value() {
    return this.getAttribute('value');
}

set backgroundcolor(val) {
    this.setAttribute('backgroundcolor', val);
}

get backgroundcolor() {
    return this.getAttribute('backgroundcolor');
}
```

Помните, что при использовании рефлексии наши атрибуты являются «источником истины», поэтому эти методы просто устанавливают или получают атрибут напрямую.

Мы и вправду почти готовы продемонстрировать работу ползунка! Возвращаясь к листингу 3.22, в котором содержится определение класса компонента, вспомните о `attributeChangedCallback`. У нас есть два метода, которые еще не существуют. При получении нового значения ползунка мы видим это:

```
case 'value':
   this.refreshSlider(newValue);
   break;
```

Аналогично, при получении нового значения цвета фона у нас есть это:

```
case 'backgroundcolor':
   this.setColor(newValue);
   break;
```

Чтобы мы могли начать видеть результаты нашей работы, нужно создать эти функции в классе компонентов.

**Листинг 3.24 Функции для установки цвета фона и значения ползунка**

Устанавливает цвет фона (плавный переход от непрозрачного сплошного цвета к тому самому прозрачному цвету)

```
setColor(color) {
    if (this.querySelector('.bg-overlay')) {
```

```
            this.querySelector('.bg-overlay').style.background =
              `linear-gradient(to right, ${color} 0%, ${color}00 100%)`;
    }
}
refreshSlider(value) {
    if (this.querySelector('.thumb')) {
            this.querySelector('.thumb').style.left = (value/100 *
              this.offsetWidth - this.querySelector('.thumb').offsetWidth/2)
              + 'px';
    }
}
```

Устанавливает текущее местоположение рукоятки ползунка на основе ее значения

Обе функции, вероятно, нуждаются в небольшом объяснении, даже если они совсем маленькие. Вначале проверяем, существует ли элемент DOM, который мы меняем. У `attributeChangedCallback` есть некоторая проблема с синхронизацией: он будет запускаться первым еще до `connectedCallback`, если в самом начале есть атрибуты компонента. Таким образом, этих элементов DOM, возможно, еще не существует. Как только мы обновим данный компонент для использования Shadow DOM позже, данной проблемы больше не будет. По этой же причине нам нужно добавить пару строк к `connectedCallback`, чтобы убедиться, что работаем с начальными атрибутами:

```
this.setColor(this.backgroundcolor);
     this.refreshSlider(this.value);
```

Затем, при настройке цвета, получаемое нами значение цвета является шестнадцатеричным (со знаком решетки вначале). Вначале, или на позиции 0 %, мы можем использовать это значение цвета как обычно. В нашем примере это красный цвет, или # ff0000. Второй узел градиента, на позиции 100 %, должен быть того же цвета, но полностью прозрачным. За исключением Edge, каждый современный браузер поддерживает добавление двух дополнительных нулей «00» в конце для указания прозрачности, чтобы дополнить двузначные значения красного, зеленого и синего цветов в более крупном шестнадцатеричном коде. Мы будем беспокоиться об Edge позже!

Функция `refreshSlider` – это довольно простая математика. Мы рассчитываем горизонтальное расположение рукоятки, взяв долю (процент, деленный на 100) от общей ширины компонента. Небольшая хитрость состоит в том, что на самом деле нам не нужен крайний левый край. Расположение рукоятки ровно посередине должно указывать значение. Чтобы отцентрировать ее, нам нужно отнять половину от ширины.

После этих последних обновлений, даже если у нас и нет интерактивности, по крайней мере, наши атрибуты приводят к обновлению компонента. Теперь мы можем загрузить HTML-файл и увидеть нечто похожее на то, что изображено на рис. 3.8.

Круто то, что даже если у нас еще нет интерактивности, атрибуты в нашем примере можно изменить. Когда страница обновится, вы увидите новый цвет и другое число в процентах. Как насчет синего фона на позиции 70 %?

```
<wcia-slider backgroundcolor="#0000ff" value="70"></wcia-slider>
```

**Рис. 3.8. На данный момент компонент-ползунок выглядит так**

Мы почти закончили! Следующий шаг – сделать так, чтобы рукоятку можно было перетаскивать.

Давайте закончим наш компонент, добавив несколько слушателей событий мыши к компонентам. Этих трех слушателей можно увидеть в приведенном ниже листинге.

**Листинг 3.25 Добавление трех слушателей событий для управления перемещением мыши, когда кнопка мыши нажата и когда она отпущена**

```
connectedCallback() {
    this.innerHTML = '<div class="bg-overlay"></div><div
     class="thumb"></div>';

    document.addEventListener('mousemove',
            e => this.eventHandler(e));

    document.addEventListener('mouseup', e => this.eventHandler(e));
    this.addEventListener('mousedown', e => this.eventHandler(e));
    this.refreshSlider(this.value);
    this.setColor(this.backgroundcolor);
}
```

Слушатели событий мыши, чтобы активировать перетаскивание ползунка

Из-за проблем с синхронизацией у attributeChangedCallback, когда он запускается первым, обновляем ползунок и раскрашиваем его

Когда речь идет о событиях мыши, при которых кнопка мыши нажата, в действительности нам важно только, когда пользователь кликает на компонент. Даже когда он кликает за пределами рукоятки, должна быть привязка к горизонтальному положению в ползунке. События, при которых кнопка мыши отпущена, должны фиксироваться на всей веб-странице. Если пользователь щелкает внутри компонента, но затем мышь перетаскивается наружу, он по-прежнему должен иметь возможность отпустить кнопку мыши, отпуская рукоятку. Точно так же, когда речь идет о перемещениях мыши, даже когда мышь перетаскивается за пределы компонента, положение ползунка все равно должно обновиться (как можно лучше в пределах границ ползунка).

Осталось только добавить код для нашего нового метода `eventHandler`.

**Листинг 3.26 Функция для обработки событий и функция для обновления процентного показателя ползунка**

```
updateX(x) {
    let hPos =
    x - this.querySelector('.thumb') .offsetWidth/2;
    if (hPos > this.offsetWidth) {
        hPos = this.offsetWidth;
    }
```

Смещает горизонтальное положение, чтобы использовать центр рукоятки

Ограничивает горизонтальное положение границами компонентов

```
    if (hPos < 0) {
        hPos = 0;
    }
    this.value = (hPos / this.offsetWidth) * 100;
}

eventHandler(e) {
    const bounds = this.getBoundingClientRect();
    const x = e.clientX - bounds.left;

    switch (e.type) {
        case 'mousedown':
            this.isDragging = true;
            this.updateX(x);
            this.refreshSlider(this.value);
            break;

        case 'mouseup':
            this.isDragging = false;
            break;

        case 'mousemove':
            if (this.isDragging) {
                this.updateX(x);
                this.refreshSlider(this.value);
            }
            break;
    }
}
```

Вычисляет горизонтальное положение в процентах и устанавливает атрибут value через API сеттера

Вычисляет горизонтальное положение относительно левого края компонента

Когда кнопка мыши нажата, устанавливает логическое значение, указывающее на то, что пользователь перетаскивает рукоятку, обновляет атрибут «value» и положение ползунка

Когда кнопка мыши отпущена, устанавливает логическое значение false, чтобы указать, что пользователь больше не перетаскивает рукоятку

При перемещении мыши, если логическое значение указывает на то, что пользователь перетаскивает рукоятку, атрибут «value» и положение ползунка обновляются

Благодаря этому последнему дополнению наш компонент-ползунок полностью функционален! Мы можем даже открыть инструменты разработчика, как показано на рис. 3.9, чтобы наблюдать за изменениями атрибута value при перетаскивании рукоятки.


**Рис. 3.9 Использование компонента-ползунка и наблюдение за тем, как обновляется атрибут value в инструментах разработчика**

Однако наш компонент еще не готов! Им нельзя поделиться, если кто-то еще в вашей команде захочет использовать его. Для этого потребуется добавить в компонент соответствующие стили CSS (в виде реальных стилей CSS, а не настройки в стиле JS, как это было в предыдущей главе) и отделить эти визуальные проблемы от основного класса компонента.

## Резюме

В этой главе мы расширили наш репертуар методов API пользовательских элементов, используя методы `connectedCallback` и `attributeChangedCallback`. В следующей главе мы подробно рассмотрим остальную часть жизненного цикла веб-компонента и сравним его с аналогичными жизненными циклами компонентов как в сети, так и за ее пределами. Также из этой главы вы узнали:

- как использовать атрибуты для вызова конечной точки для службы поиска, учитывая мысли о том, какие атрибуты нужно отслеживать, а какие нет, в том числе о том, как на самом деле следить за атрибутами на практике с помощью API веб-компонентов;
- что такое отражение и как оно может сделать ваш компонент более устойчивым, чтобы его можно было использовать с помощью тега, а также через пользовательский API JS, и как избежать проблемы реентерабельности;
- стратегии того, когда использовать атрибуты по сравнению с настраиваемым API, а когда использовать и то, и другое, чтобы обеспечить более подходящий опыт разработчика для пользователей вашего компонента.

# 4

# Жизненный цикл компонента

***Эта глава охватывает следующие темы:***

- использование метода `connectedCallCallback` для прослушивания, когда ваш компонент добавляется в DOM;
- как узнать, когда и как использовать метод конструктора, особенно потому, что это происходит до того, как компонент получает доступ к DOM;
- применение метода `desconnectedCallback`, чтобы прибрать за своим компонентом;
- редко используемый метод `adoptedCallback`.

## 4.1 API веб-компонентов

До сих пор мы исследовали несколько различных методов из API веб-компонентов, но на самом деле мы не говорили об API в целом. Эти методы являются основными строительными блоками для построения всего, от пользовательских компонентов до целых приложений. Поэтому неплохо было бы рассмотреть их все подробно. В последней главе мы

рассмотрели метод `attributeChangedCallback` и статический сеттер `observedAttributes`. В этой главе мы рассмотрим все остальное так же детально.

Кроме того, мы должны учитывать, что теперь, когда веб-компоненты поставляются в браузерах, эта спецификация должна считаться постоянной частью рабочего процесса веб-разработки на долгие годы. Учитывая это, мы должны быть уверены, что веб-компоненты могут использоваться в различных ситуациях.

Наиболее очевидный сценарий использования веб-компонентов пересекается с теми сценариями, на которые ориентированы большие фреймворки, такие как Angular, React и Vue. Вообще говоря, этот вариант использования представляет собой ориентированное на данные веб-приложение, которое может взаимодействовать с API на базе REST. На другой стороне спектра, когда мы видим более интенсивное использование графики для сети, например игры, 3D, видео и т. д., мы должны знать, что API веб-компонентов может справиться и с ними.

Чтобы иметь такую уверенность, я хочу подробно рассмотреть весь API, а также сравнить его с парой различных жизненных циклов компонентов. В случае с более традиционными веб-приложениями мы можем рассмотреть типичный жизненный цикл компонента React. Что касается более интенсивных графических приложений, можно взглянуть на жизненный цикл компонента чрезвычайно успешного (не сетевого) игрового движка с поддержкой 3D под названием Unity.

## 4.2 Обработчик connectedCallback

Ранее мы использовали метод `connectedCallback` в примерах из последних двух глав. Давайте вернемся к нему. На этот раз, однако, добавим оповещение в общий компонент, которое будет точно предупреждать нас, когда наш компонент запускается.

**Листинг 4.1 Проверка, когда вызывается наш метод `connectedCallback`**

```
<script>
    class MyCustomTag extends HTMLElement {
        connectedCallback() {
            alert('hi from MyCustomTag');
            this.innerHTML = '<h2>'+ this.getAttribute('title') +
              '</h2><button>click me</button>';
        }
    }

    if (!customElements.get('my-custom-tag')) {
        customElements.define('my-custom-tag', MyCustomTag);
    }
</script>

<style>
    my-custom-tag {
        background-color: blue;
        padding: 20px;
```

Оповещение, добавленное в connectedCallback из предыдущего примера

```
        display: inline-block;
        color: white;
    }
</style>

<body>
<my-custom-tag title="Another title"></my-custom-tag>
</body>
```

Конечно, то, что мы должны увидеть при запуске этого кода в нашем браузере, даже еще более простое по сравнению с тем, что мы видели в последних двух главах: простой, уродливый веб-компонент с заголовком и кнопкой с надписью «щелкни по мне». После добавления оповещения вы также сразу увидите модальное окно с надписью «Привет от MyCustomTag».

Теперь вопрос, основанный на ограниченном количестве кода, который у нас есть, заключается в том, когда вызывается метод `connectedCallback`. Подсказка кроется в названии этого метода, но давайте исследуем его, удалив `<my-custom-tag title="Another title"></my-custom-tag>` из тела нашей страницы.

Теперь у нас полностью пустая страница, но мы по-прежнему делаем что-то на ней. Наш блок со скриптами все еще работает, поэтому мы все еще регистрируем этот пользовательский компонент в качестве того, что мы могли бы использовать. Просто пока мы не размещаем его на странице.

Учитывая это и удалив наш элемент из тела, давайте обновим страницу: нет элемента, и нет оповещения. Воспользуемся конструктором нашего компонента, чтобы взглянуть на это немного подробнее. Если вы помните, в главе 2 мы определили конструктор как функцию, которая запускается при создании экземпляра класса.

Обратите внимание, что поскольку мы используем конструктор в унаследованном классе, то должны вызвать `super();` в качестве первой строки. При этом также вызывается конструктор `HTMLElement`. Обычно при вызове унаследованного метода можно вызывать `super.myInheritedMethod()` в любой строке, но здесь, в конструкторе, это просто `super();` на первой строке в конструкторе.

**Листинг 4.2 Оповещение из нашего конструктора и нашего метода `connectedCallback`**

```
<script>
    class MyCustomTag extends HTMLElement {
        constructor() {
               super();
               alert('hi from MyCustomTags
               ➥ constructor');
        }
        connectedCallback() {
            alert('hi from MyCustomTag
            ➥connected callback');
```

Оповещение, добавленное в конструктор

Оповещение, остающееся в connectedCallback для сравнения синхронизации

```
            this.innerHTML = '<h2>'+ this.getAttribute('title') +
              '</h2><button>click me</button>';
        }
    }

    if (!customElements.get('my-custom-tag')) {
        customElements.define('my-custom-tag', MyCustomTag);
    }
</script>
```

Хорошо, итак, если мы обновим эту страницу... опять ничего не происходит. Обратите внимание, что хотя мы полностью определили наш элемент, мы еще не создали его экземпляр или не привели его в действие! Чтобы проверить нашу теорию относительно того, что конструктор вызывается при создании, а `connectedCallback` начинает действовать при добавлении в DOM, давайте немного поработаем с DOM вручную с помощью JS.

После загрузки пустой страницы мы откроем инструменты разработки браузера и консоль. В консоли введите следующее:

```
x = document.createElement('my-custom-tag');
```

Здорово! Оповещение нашего конструктора запускается, и мы видим сообщение «привет от конструктора MyCustomTags». Создавая элемент, мы неявно вызвали `new MyCustomTag();`, в результате вызывается конструктор. Однако в то же время метод `connectedCallback` не был вызван, потому что мы не добавили его в наш DOM. Давайте сделаем это сейчас! В той же консоли, теперь, когда наша переменная `x` установлена, выполните приведенный ниже код:

```
document.body.appendChild(x);
```

Как и ожидалось, вызывается оповещение от `connectedCallback`. Кроме того, теперь вы должны увидеть компонент в теле страницы. Этот поток, от создания элемента до `connectedCallback`, зафиксирован на рис. 4.1.

Что, если мы попробуем что-нибудь менее прямолинейное? То, что мы только что сделали, вызывает вопрос относительно того, был ли запущен метод `connectedCallback`, потому что мы добавляли его в *любой* элемент или это был вопрос добавления его в DOM нашей страницы. Давайте проверим это, обновив страницу, и снова создадим наш элемент в консоли:

```
myEl = document.createElement('my-custom-tag');
```

Конечно, оповещение конструктора все равно сработает и покажет нам сообщение. Далее создадим еще один элемент, который будет действовать как контейнер:

```
myContainer = document.createElement('div');
```

Теперь наступает момент истины. Предупредит ли нас `connectedCallback`, когда мы добавим `myEl` в `myContainer`? Давайте попробуем:

```
myContainer.appendChild(myEl);
```

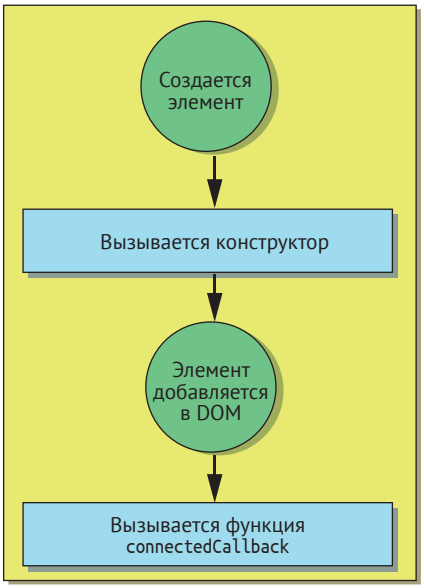



**Рис. 4.1 Начало жизненного цикла веб-компонента: сначала конструктор, а затем `connectedCallback` после добавления в DOM**

Ответ – нет! Добавление пользовательского компонента просто к любому элементу, еще не присоединенному к DOM, не будет инициировать метод `connectedCallback`. У нас есть изолированный узел в переменной `myContainer`. Вот как он выглядит:

```
<div>
      <my-custom-tag></my-custom-tag>
</div>
```

Хотя мы доказали, что наш метод `connectedCallback` не запускается, когда мы добавляем его в то, что не подключено к DOM, мы еще не доказали, что косвенное добавление в DOM инициирует этот метод. Давайте продолжим в консоли и попробуем это:

```
document.body.appendChild(myContainer);
```

Подтверждено! Вместо того чтобы добавлять наш пользовательский элемент непосредственно на страницу, сначала мы добавили его в другой контейнер (`<div>`). Затем добавили этот контейнер в DOM, и наш метод `connectedCallback` по-прежнему вызывается, доказывая, что функция обратного вызова вызывается, только когда она добавляется в страницу, и больше нигде, даже если она не добавлена туда напрямую.

Кроме того, если мы удаляем элемент, а затем повторно добавляем его, то видим, что метод `connectedCallback` вызывается каждый раз:

```
document.body.removeChild(myContainer);
document.body.appendChild(myContainer);
```

На самом деле это означает, что если вы добавляете, удаляете, а затем снова добавляете свой компонент, вы должны быть осторожны, чтобы выполнить любую разовую настройку только один раз.

На рис. 4.2 представлено наше объяснение с четырьмя сценариями. Компонент может находиться непосредственно на странице или даже внутри другого компонента. Если компонент, либо внешний компонент (а это может быть внешний, внешний, внешний компонент), находится на главной HTML-странице, будет вызываться метод `connectedCallback`.

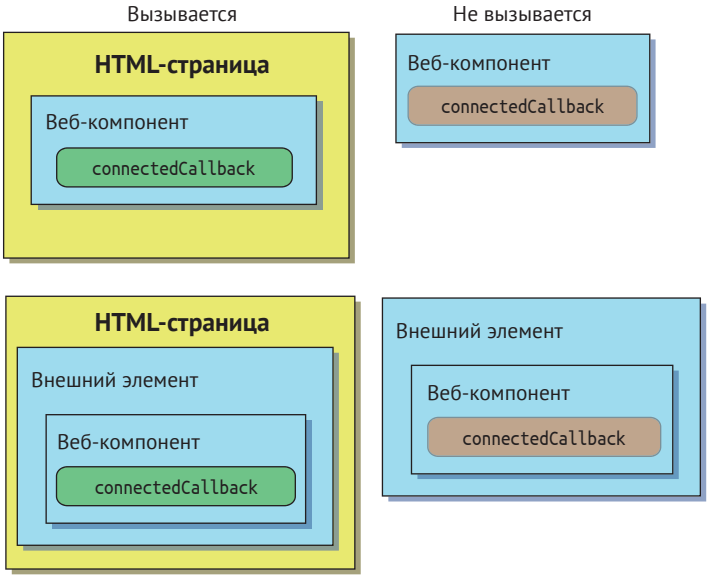



**Рис. 4.2 Четыре различных сценария создания вашего веб-компонента**

С другой стороны, даже если компонент добавляется внутри другого элемента, `connectedCallback` не будет работать, если внешний элемент на главной странице отсутствует. В целом, для того чтобы метод `connectedCallback` был инициирован, у компонента должен быть предок на главной HTML-странице.

## 4.2.1 Конструктор в сравнении с методом connectedCalback

Какое значение все это имеет для практических целей? Какой логике место в конструкторе по сравнению с методом `connectedCallback`? Было бы разумно подумать, что можно запихнуть все в конструктор и оставить метод `connectedCallback` пустым. К сожалению, нет – здесь есть некоторые нюансы.

Важным аспектом того, что вы хотите сделать при создании компонента, является настройка содержимого вашего элемента. Вы, вероятно, захотите установить `innerHTML` для разметки. Так, в нашем простом примере мы добавляем заголовок и кнопку. Также вы, возможно, захотите получить атрибут своего компонента. К сожалению, при запуске конструктора элемент еще не готов к взаимодействию таким образом.

Это можно продемонстрировать, переместив строку с `innerHTML` в конструктор:

**Листинг 4.3 (Неудачная) попытка установить innerHTML из конструктора**

```
class MyCustomTag extends HTMLElement {
    constructor() {
        super();
        this.innerHTML = '<h2>'+ this.getAttribute('title') +
          '</h2><button>click me</button>';
    }
    connectedCallback() {}
}
```

Когда наша страница перезагрузится, можно попытаться снова создать элемент с помощью функции `create-Element`, но в нашей консоли появляется ошибка:

```
DOMException: Failed to construct 'CustomElement': The result must not have
     Children
```

Браузер сообщает нам, что при первоначальном создании нашего пользовательского элемента ему запрещено иметь дочерние элементы. Кроме того, мы можем проверить наш атрибут `title`, который мы использовали для заполнения нашего тега заголовка, используя конструктор и метод `connectedCallback`.

**Листинг 4.4 Попытка доступа к атрибутам из конструктора и из метода connectedCallback**

```
class MyCustomTag extends HTMLElement {
    constructor() {
        super();
        console.log('From constructor',
                    this.getAttribute('title'));
    }
    connectedCallback() {
        console.log('From connectedCallback',
                    this.getAttribute('title'));
    }
}
```

Доступ к атрибуту этого компонента из конструктора (неудачно)

Доступ к атрибуту этого компонента из метода connectedCallback (успешно)

Когда мы перейдем к предыдущему листингу и перезагрузим нашу страницу, консоль укажет на то, что конструктор еще не знает о заголовке, регистрируя значение `null`. Хотя с нашим методом `connectedCallback` все в порядке.

Просто взглянув на это и увидев, что работает, а что нет, мы можем начать понимать, как нам организовать наш компонент. Метод `connectedCallback` должен содержать всю логику для визуального заполнения нашего элемента. Для типичного компонента большая часть логики, например добавление событий, взаимодействий и т. д., будет зависеть от наличия этих визуальных элементов. Из-за этого конструктор может остаться довольно пустым или лишенным значимого кода для многих ситуаций.

Однако в зависимости от вашего компонента в конструкторе могут существовать исключения. Одним из таких исключений является логика,

которая может возникнуть после инициализации вашего элемента, но до того, как он будет добавлен на страницу. Вы можете, например, создать элемент заранее и выполнить сетевой запрос на получение информации из интернета, прежде чем присоединить свой компонент к DOM. Таким образом, если у вашего компонента имеются все данные, которые он должен отобразить, он может сделать это мгновенно, находясь на странице. В этом случае, поскольку никаких зависимостей от визуальных элементов в вашем компоненте нет, конструктор может стать подходящим местом для этого кода.

**Листинг 4.5 Прекрасно отформатированный список свойств в конструкторе**

```
class MyCustomTag extends HTMLElement {
    constructor() {
          super();
    /**
     * URL-адрес, чтобы получить данные для заполнения нашего гипотетического списка;
     */
    this.serviceURL =
'http://company.com/service.json';

    /**
     * Внутренний счетчик для отслеживания чего-либо;
     */
    this.counter = 0;

    /**
     * Последнее сообщение об ошибке;
     */
    this.error;
    }
    connectedCallback() { . . . }
}
```

Метод конструктора

Добавляет удобочитаемые свойства в конструктор

Как я упоминал в начале главы, одним из замечательных применений конструктора может быть объявление свойств. Действительно удобно, когда в верхней части вашего класса есть конструктор и у вас есть возможность легко читать все свойства, которые вы используете, как видно из листинга 4.5. Я обнаружил, что даже если вы еще не настроили свои свойства, конструктор все равно отлично подходит для удобства чтения компонентов. Однако я должен еще раз упомянуть, что с появлением последней версии Chrome, которая поддерживает общедоступные и закрытые поля классов, мы можем объявлять наши свойства в самом классе, что гораздо приятнее и является более встраиваемым в любой другой язык, поддерживающий классы. Как только другие браузеры получат такую поддержку, описанный мною подход, скорее всего, станет плохой практикой.

Если вы используете Shadow DOM, возникает одно серьезное предостережение относительно применения конструктора в сравнении с методом `connectedCallback` для связанной с DOM логики, о чем пойдет речь

в главе 7. При использовании технологии Shadow DOM вы создаете отдельную мини-модель DOM, которая является внутренней для вашего компонента. В этом случае Shadow DOM доступен всякий раз, когда вы создаете его, – даже в конструкторе.

Вот почему вы будете становиться свидетелем того, что многие современные веб-компоненты используют конструктор практически для всего в компоненте, в то время как метод `connectedCallback` может вообще не использоваться.

Будете ли вы использовать теневую модель DOM? До недавнего времени я бы не рекомендовал это делать, но Firefox только что выпустил обновление с поддержкой для нее (наряду со всеми функциями веб-компонентов), и Edge должен вскоре выпустить соответствующий релиз.

Какой бы потрясающей ни была теневая модель DOM, вам нужно взвесить, нужна ли она вам и поддерживается ли она в выбранном вами браузере. Конечно, будут ситуации, когда использовать ее для своего проекта просто не имеет смысла – важно будет знать нюансы метода `connectedCallback` и конструктора.

## 4.3 Остальные методы жизненного цикла веб-компонента

Мы обсудили четыре из шести методов жизненного цикла нашего компонента (конструктор, `connectedCallback`, `attributeChangedCallback` и `observedAttributes`). Осталось два метода: `disconnectedCallback` и `adoptedCallback`.

### 4.3.1 Метод disconnectedCallback

Метод `disconnectedCallback` служит очень важной цели, которая состоит в том, чтобы дать компоненту возможность прибрать после себя. Этот метод запускается, когда компонент удаляется из DOM.

На то есть две причины. Во-первых, вы не хотите запускать случайный код, когда вам это не нужно. Второе – дать возможность запустить сбор мусора. Если вы незнакомы со сбором мусора, посмотрите на язык C++. Если вы храните данные в переменной, они никуда не исчезнут, или, если использовать правильную терминологию, переменные не будут *высвобождены.* Поскольку вы являетесь разработчиком, то должны надлежащим образом избавиться от них, когда закончите работу. Если вы не будете осторожны, все переменные, которые вы больше не используете, могут начать добавлять и использовать огромное количество памяти! К счастью, в случае с более современными языками, такими как JS, ваши неиспользуемые переменные будут «собираться как мусор». Время от времени, когда движок (в нашем случае движок JS) знает, что для очистки достаточно времени простоя, он войдет и высвободит переменные, которые вы не используете. Однако он не экстрасенс и не может угадать, что вам не нужно.

Вместо этого, если он обнаружит, что вы не ссылаетесь на что-либо в памяти, как показано на рис. 4.3, он высвободит это. Вот почему метод `disconnectedCallback` – хорошая возможность для сброса или обнуления всех переменных, которые могут быть связаны с другими объектами.

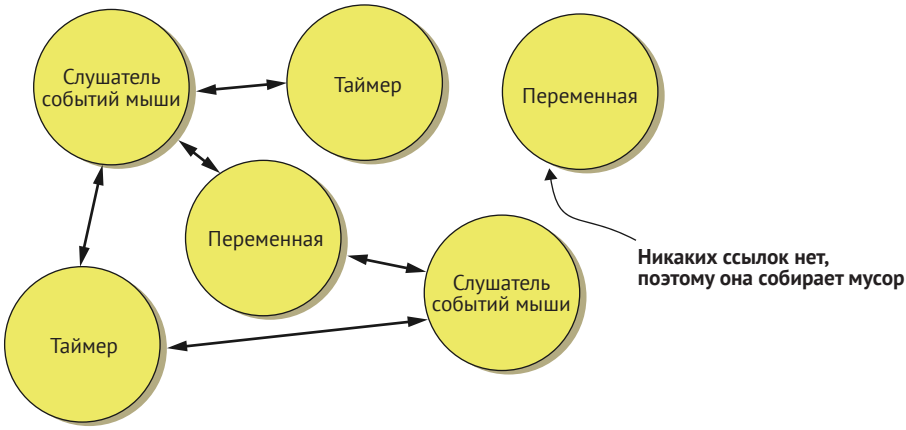



**Рис. 4.3 Ссылки на ячейки памяти внутри веб-компонента**

Беспокойство по поводу этих мелких деталей может быть непростой задачей, когда ваш компонент просто работает. Иногда, если мы точно знаем, как используем наш компонент, мы можем игнорировать некоторые из них. Например, если вы знаете, что ваше приложение никогда не будет удалено из DOM, вы можете игнорировать очистку. Конечно, контекст проектов может измениться, и от компонента, который вы никогда и не думали удалять, возможно, потребуется избавиться.

Чтобы привести пример столь необходимой очистки, рассмотрим ситуацию, когда вы запрашиваете сервер каждые 30 секунд, чтобы получить обновленные данные. Если вы используете метод `removeChild`, чтобы удалить дочерний элемент из родительского контейнера (`removeChild(вашэлемент);`), этот таймер по-прежнему будет запускаться, и по-прежнему будет выполняться запрос к серверу. Давайте попробуем провести упрощенный эксперимент на примере таймера обратного отсчета.

**Листинг 4.6 Демонстрация кода, который выполняется после удаления элемента**

```
<html>
<head>
    <meta charset="UTF-8">
    <title>Cleanup Component</title>
    <script>
        class CleanupComponent extends HTMLElement {
            connectedCallback() {
                this.counter = 100;
                setInterval( () =>
```

```
                this.update(), 1000);
            }

            update() {
                this.innerHTML = this.counter;
                this.counter --;
                console.log(this.counter);
            }
        }

        customElements.define('cleanup-component', CleanupComponent);
    </script>
</head>

<body>
    <cleanup-component></cleanup-component>
    <button onclick="document.body.removeChild(document.querySelector
                ('cleanup-component'))">remove</button>
</body>
</html>
```

Запускает таймер обратного отсчета

Консоль записывает текущее значение таймера (который все еще работает после удаления компонента!)

Кнопка для удаления компонента

В этом примере мы также регистрируем значение нашего счетчика:

```
console.log(this.counter);
```

Я также добавил кнопку с встроенным кодом JS. При нажатии на кнопку «Удалить» компонент таймера обратного отсчета удаляется из DOM.

При запуске этого примера таймер начинает обратный отсчет времени, как обычно. После нажатия на кнопку «Удалить» вы больше не видите таймер, но если вы откроете журнал консоли, то увидите, что он *по-прежнему* ведет обратный отсчет! Оставить этот таймер включенным – крайне плохая идея. Еще хуже, если мы будем засорять консольный журнал элементами, которые нам больше не нужны. Было бы еще хуже, если бы мы делали сетевые запросы, которые нам не нужны, или выполняли бы что-то затратное в вычислительном отношении для элемента, который нам не нужен.

Таким образом, мы можем использовать метод `disconnectedCallback` для очистки нашего таймера. Скорее всего, нам также понадобится очистить все добавленные слушатели событий, например события мыши. Давайте попробуем очистить наш таймер при удалении элемента в приведенном ниже листинге.

**Листинг 4.7 Использование метода `disconnectedCallback` для очистки таймера**

```
class CleanupComponent extends HTMLElement {
    connectedCallback() {
        this.counter = 100;
        this.timer = setInterval( () => this.update(), 1000);
    }

    update() {
        this.innerHTML = this.counter;
        this.counter --;
```

```
        console.log(this.counter);
    }

    disconnectedCallback() {
        clearInterval(this.timer);
    }
}
```

**Убирает таймер, когда компонент удаляется**

Теперь мы заключили наш таймер в переменную:

```
this.timer = setInterval( () => this.update(), 1000);
```

Таким образом, когда нам нужно выполнить очистку с использованием метода `disconnectedCallback`, мы можем сделать это, применив ту же переменную:

```
disconnectedCallback() {
    clearInterval(this.timer);
}
```

При повторной проверке журналов видно, что у нас больше нет сообщений, и наш элемент должен быть надлежащим образом убран на следующем этапе.

## 4.3.2 Метод adoptedCallback

Несмотря на то что даже мне нужно засучить рукава и чаще использовать метод `disnectedCallback` для написания более качественных и универсальных компонентов, в действительности я не видел, чтобы большинству людей он когда-либо был нужен. Метод жизненного цикла `adoptedCallback` запускается, когда ваш веб-компонент перемещается в другой документ.

Не волнуйтесь, если это не имеет смысла, потому что такого обычно не происходит. Обычно у вас будет только по одному документу на HTML-страницу. Исключением является применение плавающих (или встроенных) фреймов, которые в действительности перестали использоваться в большинстве случаев. По сути, с помощью такого фрейма у вас получается мини-HTML-страница во фрейме на вашей главной HTML-странице.

Можно взять элементы из плавающего фрейма и поместить их в окружающую страницу, или наоборот. Для этого вам нужно получить ссылку на элемент, а затем переместить ее в новый документ:

```
const frame = document.getElementsByTagName("iframe")[0]
const el = frame.contentWindow.document.getElementsByTagName(
       "my-custom-component")[0];
const adopted = document.adoptNode(el);
```

После этого будет запущен метод жизненного цикла `adoptedCallback`. Но опять же, я редко работаю с плавающими фреймами. Мне никогда не приходилось перемещать узлы из одного документа в другой. Возможно, вы найдете применение этому методу, и если у вас это получится, знайте, что ваш компонент может слушать!

## 4.4 Сравнение с жизненным циклом React

Давайте теперь поговорим о жизненном цикле веб-компонента в привязке к жизненному циклу React. В конце концов, при наличии всего лишь нескольких методов жизненного цикла может показаться, что веб-компоненты неполноценны. Учитывая популярность React и его широкую аудиторию разработчиков, он отлично подходит для сравнения веб-компонентов с целью выяснить, как они состыковываются.

React немного своенравен, как и все фреймворки и библиотеки. Он предлагает определенный жизненный цикл компонента, который подходит для разработчиков React и их вариантов использования. Конечно, в этом нет абсолютно ничего плохого, но дело в том, что мы рассматриваем жизненный цикл, который может или не может быть применим к тому, как вы хотите работать. Я хотел бы повторить, что это именно то, что мне нравится в работе с веб-компонентами – у них достаточно возможностей, чтобы покрыть необходимый минимум того, что вам нужно, а все, что сверх этого, можно создать с помощью собственного кода или существующих микрофреймворков либо библиотек.

В документации по React его методы жизненного цикла разбиты на четыре основные категории: монтирование, обновление, размонтирование и обработка ошибок. Метод обработки ошибок – метод, с которым мы еще не сталкивались, и действительно, в веб-компонентах нет ничего подобного.

Философия React (по крайней мере, начиная с версии 16) заключается в том, чтобы установить «границы ошибок», чтобы при возникновении ошибки в одном компоненте она не сводила бы на нет остальные компоненты или приложение.

Хотя и верно, что ошибка в JS потенциально может делать действительно плохие и неожиданные вещи где-нибудь в приложении на основе веб-компонентов, в случае с React ситуация была бы немного хуже. До выхода версии 16 ошибка могла бы размонтировать все ваше приложение! Ошибки в чистом JS обычно более покорные – неожиданные вещи будут случаться, но, как правило, ваше приложение не будет поставлено на колени. В результате этого в версии 16 были созданы границы ошибок, чтобы каждый компонент мог справиться с любой ошибкой и не влиять на остальное. Веб-компоненты немного более децентрализованы, поэтому проблемы React не так схожи.

В React понятие *монтирование* означает создание фрагмента HTML-кода, представляющего ваш компонент, а затем вставку этого кода в DOM. Для монтирования существует несколько соответствующих методов.

Как и веб-компоненты (и большинство всего остального), React позволяет переопределить конструктор. Это очень напоминает веб-компоненты, в том смысле, что вы, скорее всего, не захотите размещать здесь огромное количество логики компонентов и в идеале инициализировали бы вещи, которые будете использовать позже. Методы `componentWillMount` и `componentDid-Mount` позволяют вам работать до и после добавления компонента в DOM.

В то время как метод `componentDidMount` во многом схож с методом веб-компонентов `connectedCallback`, похоже, что для метода `componentWillMount` здесь не так много вариантов использования. Здесь нет ничего, чего нельзя было бы сделать с помощью конструктора. Фактически версия 16 уже показывает предупреждающие сообщения о том, что этот метод будет устаревшим в следующей основной версии.

До вызова метода `componentDidMount` (или когда компонент изменяется каким-либо образом) вы можете переопределить метод `render`. С помощью этого метода можно в основном возвращать HTML-код, чтобы представить внутреннюю разметку вашего компонента.

В случае с веб-компонентами метод `render` просто не требуется в качестве стандартного метода жизненного цикла, хотя LitElement и другие добавили его в свои веб-компоненты, чтобы сделать обновление HTML-кода более упорядоченным. Используя базовый жизненный цикл как он есть, мы можем контролировать `innerHTML` нашего компонента в любое время и не ограничены жизненным циклом нашего компонента в отношении того, когда устанавливать содержимое нашего компонента, или даже того, какие фрагменты обновляются. В этой связи лучше не быть связанными жесткими правилами, которые говорят, где и когда мы можем создавать внутреннюю работу своего компонента! В случае с LitElement и различными фреймворками вы ведетесь на шаблон проектирования и делаете выбор, чтобы быть связанными правилами, которые определяют, когда ваш компонент визуализируется. Если вы выбираете именно это, отлично, но в качестве стандарта, который должен соответствовать различным вариантам использования, я думаю, что гораздо лучше выбрать нечто вроде метода `render`.

Для обновления компонента в React также есть несколько методов: `componentWillReceiveProps`, `shouldComponentUpdate`, `componentWillUpdate`, `getSnapshotBeforeUpdate` и `componentDidUpdate`. Помимо метода `componentWillReceiveProps`, который скоро устареет, остальные могут быть полезны, когда в вашем компоненте что-то меняется и его нужно обновить. Они менее актуальны для веб-компонентов, потому что React как система отслеживает кучу всего, что выходит за рамки вашего реального HTML-элемента. Состояние, свойства и т. д. – это все, что изменяется и запускает изменение вашего компонента. На самом деле React рекомендуется использовать совершенно иначе.

Вы должны изменить состояние или свойства, а ваш компонент должен… «реагировать» на эти изменения.

С другой стороны, когда вы взаимодействуете с веб-компонентами, вы, скорее всего, будете поступать так же, как и с обычным элементом DOM: используя пользовательский API или атрибуты. При такой разнице необходимость в этих дополнительных методах отпадает. Некоторые могут утверждать, что способ, которым работает React, предлагает больше помощи, но при применении веб-компонентов у вас больше свободы и вы можете делать все, как вы хотите, как это нужно именно для вашего проекта.

## 4.5 Сравнение с жизненным циклом игрового движка

Говоря о свободе реализации того, что мы хотим в зависимости от проекта, не стоит рассматривать традиционные веб-приложения как единственный вариант использования для создания чего-либо в сети. Постоянно создается все больше и больше проектов с большим объемом графических операций. Неплохой вариант использования, на который стоит обратить внимание, – игровой движок. В связи с этим я думаю, что справедливо будет сравнить жизненный цикл веб-компонента с Unity. Unity 3D – один из самых популярных инструментов для создания 3D-графики в реальном времени для игр, приложений и даже дополненной или виртуальной реальности.

В Unity разработчик, как правило, работает с каким-либо 3D-объектом, к которому прикреплен базовый класс `Monobehavior`. Подобно тому, как наш веб-компонент расширяет `HTMLElement`, настраиваемое поведение в Unity расширяет поведение `Monobehavior`.

У `Monobehavior` есть два метода жизненного цикла, используемых для запуска поведения. `Awake` похож на наш конструктор веб-компонентов. Он вызывается при создании `Monobehavior`, независимо от того, активирован он или нет. В Unity поведения не обязательно активны и работают, если отключены.

Точно так же наш веб-компонент в действительности не «активирован», если он не был добавлен в DOM, потому что он не отображается на странице. В Unity есть методы `OnEnable` и `OnDisable`, чтобы следить за этим. Поведение можно активировать несколько раз, так же как и наш веб-компонент можно добавить в DOM несколько раз. Таким образом, здесь метод `OnEnable` очень напоминает метод нашего веб-компонента `connectedCallback`.

Метод Unity `Start` вызывается при первой активации поведения, в том числе если оно активируется при запуске приложения. У веб-компонентов нет аналогичного вызова, и, как я уже сказал, если мы добавляем один и тот же элемент в нашу модель DOM более одного раза, нам нужно защитить себя от повторной инициализации, если это навредит нашим компонентам. К счастью, это легко преодолеть – мы можем просто установить для переменной значение `true` в первый раз, пройдя наше `connectedCallback`, и не вызывать ту же самую инициализацию с помощью конструкции `if/then`.

Эти тонкие различия имеют значение, только если вы решите не использовать свой веб-компонент, просто написав разметку в HTML-коде, как при создании, добавлении и удалении элементов с помощью JS. Например, при создании прототипа или конкретного приложения вы, вероятно, будете точно знать, как должны использоваться ваши веб-компоненты, и сможете настроить их по мере необходимости. Если вы создаете библиотеку веб-компонентов, которой вы собираетесь поделиться с кем-то еще, можно рассмотреть все эти варианты использования.

Идем далее. В жизненном цикле `Monobehavior` у Unity 3D есть несколько методов, которые называются каждый кадр рендеринга. Это означает, что они вызываются много раз в секунду, чтобы дать разработчику возможность обновлять то, что отображается на экране при обновлении графики. Эти методы обрабатывают определенные вещи, такие как физика, различные проходы визуализации и т. д. В нашем случае я сведу их к методу Unity `update`, потому что они в действительности не будут применяться к веб-компонентам, только если речь не идет о WebGL или других конкретных случаях.

Хотя у веб-компонентов нет аналогичного метода обновления в качестве части API жизненного цикла или даже множества методов обновления, которые я описал ранее, нам, возможно, он и не нужен. Нам не обязательно делать игры или какие-то вещи с большим объемом графических операций, которые должны запускать каждый кадр с JS, поэтому в таких случаях нам это не нужно. В случае когда нам нужен метод обновления, можно воспользоваться несколькими способами.

Первое, что можно попробовать, – это таймер. Давайте возьмем тот таймер из предыдущего примера и начнем с него.

**Листинг 4.8 Компонент таймер обратного отсчета**

```
<html>
<head>
    <meta charset="UTF-8">
    <title>Countdown Timer</title>
    <script>
        class CountdownTimer extends HTMLElement {
            connectedCallback() {
                this.counter = 100;
                setInterval( () =>
                       this.update(), 1000);
            }

            update() {
                this.innerHTML = this.counter;
                this.counter --;
            }
        }

        customElements.define('countdown-timer', CountdownTimer);
    </script>
</head>

<body>
    <countdown-timer></countdown-timer>
</body>
</html>
```

Создает наш внутренний таймер (вызовы обновляются каждую секунду)

Отображает текущее значение таймера

Уменьшает каждое обновление таймера

В листинге 4.8 мы создали простой пример с компонентом таймер обратного отсчета (практически такой же, как ранее в этой главе). Когда наш компонент добавляется в DOM, мы используем метод `connectedCallback` для инициализации свойства `counter` и устанавливаем для него зна-

чение, равное 100. Также мы запускаем стандартный таймер JS и присоединяем его к внутреннему методу `update`:

```
setInterval( () => this.update(), 1000);
```

Если вы использовали таймер раньше, то знаете, что последний параметр 1000 заставляет таймер срабатывать каждые 1000 миллисекунд (или каждую секунду). В самом методе `Update` мы просто устанавливаем содержимое нашего компонента с помощью `innerHTML` и уменьшаем нашу переменную на единицу.

Когда вы запустите это в браузере, то увидите числовой дисплей, который начинается со 100 и ведет обратный отсчет, отсчитывая по единице каждую секунду. Метод `setInterval` отлично подходит для таких ситуаций, когда вам нужен обычный таймер; но для анимации или графики, которую необходимо менять каждую 1/30 секунды, например, более новая функция `requestAnimationFrame` от JS даст более плавные результаты, которые фактически связаны с циклом рендеринга браузера.

Давайте поменяем метод `setInterval` на `requestAnimationFrame` и сделаем что-нибудь более анимированное в приведенном ниже листинге.

**Листинг 4.9 Замена setInterval на `requestAnimationFrame`**

```
<html>
<head>
    <title>Visual Countdown Timer</title>
    <script>
        class VisualCountdownTimer extends HTMLElement {
            connectedCallback() {
                this.timer = 200;
                this.style.backgroundColor = 'green';
                this.style.display = 'inline-block';
                this.style.height = '50px';
                requestAnimationFrame ( () =>
                        this.update());
            }

            update() {
                this.timer --;
                if (this.timer <= 0) {
                    this.timer = 200;
                }
                this.style.width =
                       this.timer + 'px';
                requestAnimationFrame ( () =>
                       this.update());
        }

        customElements.define('countdown-timer', VisualCountdownTimer);
    </script>
</head>

<body>
```

Используем метод requestAnimationFrame вместо setInterval

Плавно анимируем ширину нашего компонента

Метод requestAnimationFrame вызывается при каждом вызове update

```
    <countdown-timer></countdown-timer>
</body>
</html>
```

За исключением того, что метод `requestAnimationFrame` встречается только один раз, что заставляет нас вызывать его при каждом вызове `update`, реализация здесь в основном та же, что и в случае с `setInterval`:

```
requestAnimationFrame ( () => this.update());
```

Опять же, у меня есть счетчик, но сейчас я называю его `timer`, потому что мы будем уменьшать наш компонент с каждым кадром анимации, чтобы имитировать таймер обратного отсчета. У меня также есть стили CSS для установки цвета фона, высоты и значения `inline-block` свойства `display`. Неудивительно, что здесь я устанавливаю стили, используя код, когда могу использовать CSS, но я не хочу усложнять этот пример:

```
this.style.backgroundColor = 'green';
this.style.display = 'inline-block';
this.style.height = '50px';
```

В методе `update` мы уменьшаем наш таймер и также проверяем, равен ли он 0 или меньше его. Если это так, то мы сбрасываем его до 200, просто чтобы сохранить наш компонент в бесконечном демоцикле. После всего этого мы устанавливаем высоту и ширину компонента в свойство `timer`. Наконец, мы вызываем следующий кадр анимации и снова запускаем наш метод `update`. В итоге получаем визуальный компонент зеленого цвета, который сжимает каждый кадр до тех пор, пока он не достигнет нуля, а затем снова сбрасывается до ширины в 200 пикселей.

В дополнение к методам `setInterval` и `requestAnimationFrame` другие фреймворки и библиотеки, которые мы можем использовать, могут иметь собственные способы вызова метода `update` по времени, подобные этому. Например, если вы применяете 3D-библиотеку, такую как Three.js или Babylon, у них есть свои приемы рендеринга, которые вы можете использовать, следовательно, ваш компонент будет реализован несколько иначе.

Дело в том, что жизненный цикл веб-компонента не поставляется с методом `update`, как многие другие жизненные циклы компонентов, которые вы, возможно, встречали. Поскольку веб-технологии могут использоваться для самых разных вещей, неразумно будет диктовать вам, как следует это делать.

Большую часть времени при моей работе мне никогда не бывает нужен метод `update`. Даже простую анимацию пользовательского интерфейса можно обработать с помощью стилей CSS. И конечно, когда я это делаю, мне нравится выбирать, какой метод использовать.

Возможно, в ваших личных случаях использования вам *всегда* нужен какой-то метод `update`, подобный тому, что есть в Unity. Безусловно, это имеет смысл, если вы занимаетесь разработкой игр или чем-то подобным и вам нужен метод `render` или `update` для управления игрой и анимацией.

Веб-компоненты поддерживают наследование, и мы можем пойти еще на один уровень дальше и надстроить существующий жизненный цикл компонента. Давайте возьмем код из нашего примера с анимацией таймера обратного отсчета и используем вызов `requestAnimationFrame`, чтобы активировать его.

**Листинг 4.10 Создание наследуемой базы компонентов для обновления каждого кадра**

```
<html>
<head>
    <script>
        class GameComponentBase
                extends HTMLElement {
            constructor() {
                super();
                this.onUpdate();
            }

            update() {}
            onUpdate() {
                this.update();
                requestAnimationFrame ( () => this.onUpdate());
            }
        }

        class VisualCountdownTimer
               extends GameComponentBase {
            connectedCallback() {
                this.timer = 200;
                this.style.backgroundColor = 'green';
                this.style.display = 'inline-block';
                this.style.height = '50px';
            }

            update() {
                this.timer --;
                if (this.timer <= 0) {
                    this.timer = 200;
                }
                this.style.width = this.timer + 'px';
            }
        }

        customElements.define('countdown-timer', VisualCountdownTimer);
    </script>
</head>

<body>
    <countdown-timer></countdown-timer>
</body>
</html>
```

Класс обеспечивает основу для создания компонентов в игровом стиле

Внутренний метод обновления, чтобы поддерживать работу функции requestAnimationframe

Метод update, который должен быть заполнен компонентом с использованием базового класса

Фактический класс компонента, который расширяет базовый компонент

Итак, в примере из листинга 4.10 мы по-прежнему выполняем ту же самую простую анимацию: уменьшаем графический индикатор обратного отсчета. Но мы извлекли логику, связанную с созданием события обновления каждого кадра в своем собственном классе. Обратите внимание, что я говорю *класс*, а не *компонент*, потому что мы сделали все, чтобы создать новый компонент, кроме определения пользовательского элемента и отображения его в тег.

Вместо этого мы создаем базовый класс `GameComponentBase`, от которого компоненты могут наследовать. На рис. 4.4 показана эта цепочка наследования. Источником всего является класс `HTMLElement`.

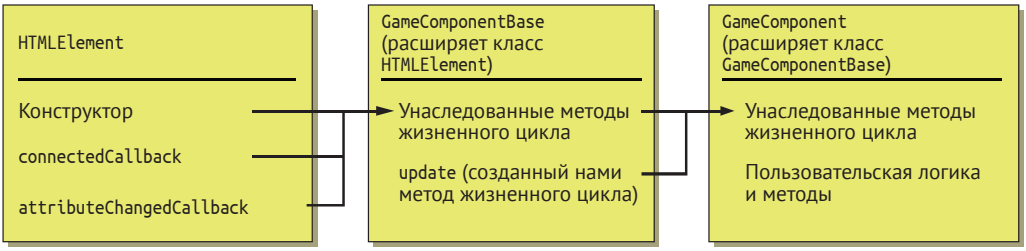



**Рис. 4.4 Использование наследования для создания подкласса `HTMLElement`, чтобы активировать обновления кадров, как в игровом движке**

Хотя я поступил по-хитрому. Вместо того чтобы вызывать метод `update` напрямую, у меня есть другой метод – `onUpdate`:

```
onUpdate() {
   this.update();
    requestAnimationFrame ( () => this.onUpdate());
}
```

Причину лучше всего объяснить так, как я бы не предложил сначала. Давайте не будем использовать оба метода, а только `update`.

**Листинг 4.11 Пример попроще с одним переопределяемым методом `update`**

```
class GameComponentBase extends HTMLElement {
    constructor() {
        super();
            this.update();
        }

        update() {
            requestAnimationFrame ( () =>
                this.update());  <-- Единичный метод update
        }
    }
```

Новый класс `GameComponentBase` по-прежнему хорош, и его можно использовать почти таким же образом, но давайте посмотрим, как это сделать.

**Листинг 4.12 Использование более простого базового класса**

```
class VisualCountdownTimer extends GameComponentBase {
    connectedCallback() {
        this.timer = 200;
        this.style.backgroundColor = 'green';
        this.style.display = 'inline-block';
        this.style.height = '50px';
    }

    update() {
        this.timer --;
        if (this.timer <= 0) {
            this.timer = 200;
        }
        this.style.width = this.timer + 'px';
        super.update();
    }
}
```

Теперь требуется вызвать super.update()

Обратите внимание, что мы слегка упростили класс `GameComponentBase`. Мы объединили два метода `update` в один, но в нашем компоненте `VisualCountdownTimer` теперь мы заставляем всех, кто использует класс `GameComponentBase`, вызывать каждый раз `super.update();`! Конечно, когда речь идет о наследовании, мы не будем вызывать метод `update` для нашего базового класса `GameComponentBase`, только если не используем `super.update()`. Не знаю, как вы, но я бы создал новый компонент и большую часть времени забывал бы вызывать `super.update()`. Такое небольшое предварительное планирование может сделать опыт разработчика более счастливым.

В Unity есть еще два метода жизненного цикла, `OnDisable` и `OnDestroy`, которые служат той же цели, что и метод `disconnectedCallback` в веб-компонентах: очистка после отключения или уничтожения компонента.

## 4.6 Жизненный цикл компонента v0

Теперь API веб-компонентов выглядит довольно солидно, не так ли? Мы сравнили и сопоставили его с жизненными циклами других компонентов, и я надеюсь, что у вас достаточно хорошее чувство, что он будет хорошо работать везде, где вы его используете. Я не жду, что вы будете знать на память каждый метод, особенно вначале. Все мы время от времени используем синтаксис Google. Одно из предостережений при использовании веб-компонентов состоит в том, что это относительно новый стандарт, и он уже прошел одну ревизию.

Это означает, что при просмотре синтаксиса вы можете случайно наткнуться на старые методы. В настоящее время мы используем API веб-компонентов версии 1. Предыдущая версия имела обозначение v0, и она не будет работать нигде, кроме того места, где она была реализована изначально: Chrome. Даже там, с течением времени, она будет становиться все более и более нестабильной.

**ВАЖНО** API веб-компонентов изменился!

На самом деле мало что изменилось (см. табл. 4.1), хотя первое, что нужно отметить, – это то, что, вместо того чтобы позволить вам использовать конструктор в версии 1, вы используете метод `createdCallback`.

**Таблица 4.1 Изменения API пользовательских элементов и веб-компонентов**

| Вызовы методов | Как он изменился |
|---|---|
| **Ранее:** `createdCallback`<br>**Теперь:** `constructor` | В версии 1 метод `createdCallback` заменен более стандартным конструктором |
| **Ранее:** `AttachedCallback`<br>**Теперь:** `connectedCallback` | В версии 1, чтобы прослушивать, когда ваш элемент был добавлен в DOM, вы используете метод `connectedCallback`; в версии 0 это был метод `attachedCallback` |
| **Ранее:** `detachedCallback`<br>**Теперь:** `disconnectedCallback` | Старый способ прослушивания, когда элемент удаляется из DOM, теперь в версии 1 используется метод `disconnectedCallback`, в версии 0 это был метод `disconnectedCallback` |
| **Ранее:**<br>`AttributechangedCallback`<br>**Теперь:**<br>`attributeChangedCallback` и `observedAttributes` | Последнее изменение – метод `attributeChangedCallback` в версии 1.<br>Название на самом деле не изменилось, а вот использование – да. Теперь необходимо убедиться, что вы задали наблюдаемые атрибуты (`observedAttribute`), как мы обсуждали в предыдущей главе, чтобы сообщить компоненту, какие атрибуты вы хотите прослушивать. Ранее эта функция обратного вызова просто слушала все |
| **Ранее:**<br>`document.registerElement`<br>**Теперь:**<br>`customElements.define` | Наконец, вне API жизненного цикла компонента способ регистрации вашего элемента тоже изменился. В настоящее время мы используем это:<br>`customElements.define('my-web-component', MyWebComponent);`<br>Раньше в версии 0 мы использовали это:<br>`document.registerElement('my-web-component', MyWebComponent);` |

## Резюме

Из этой главы вы узнали:

- как завершить изучение методов жизненного цикла, о которых вы уже узнали, с помощью оставшихся двух методов: `disconnectedCallback` и `acceptCallback`;
- о концепции сбора мусора, и почему нужно убирать за своим компонентом;
- как создать подкласс из веб-компонента и использовать его в качестве основы для предоставления общих функций, таких как покадровая анимация, другим компонентам;
- о различиях и сходствах с методами жизненного цикла React и игрового движка, а также что, несмотря на то что оба они располагают бóльшим числом методов для своих API, веб-компоненты не отстают.

# 5

# Реализация более качественного веб-приложения с помощью модулей

***Эта глава охватывает следующие темы:***

- модули ES2015 как альтернатива тегам `<script>` в вашем HTML-коде;
- создание самостоятельных веб-компонентов;
- использование веб-компонента для хранения всего своего приложения;
- управление областью видимости для функций обратного вызова при помощи толстой стрелки из ES2015.

До сих пор в упражнениях, приведенных в этой книге, мы помещали наши классы и определения компонентов в теги `<head>` на нашей главной HTML-странице. Как правило, вы ни за что не захотите делать это

в реальном проекте, и, возможно, вам захочется быть более организованным, когда речь идет о теге `<script>`, где указан путь к файлу JS для каждого имеющего у вас компонента. На первый взгляд это прекрасно. Если ваш проект использует веб-компоненты только в ограниченном виде, это работает! С таблицами стилей CSS похожая ситуация – у каждого компонента может иметься собственный CSS-файл, на который есть ссылка на главной странице. Однако когда вам нужно управлять большим количеством компонентов в своем проекте, можно потерять контроль. В этой главе мы рассмотрим модули ES2015 в качестве более подходящей альтернативы.

## 5.1 Использование тега <script> для загрузки ваших веб-компонентов

Чтобы объяснить, почему ссылки на несколько JS- или CSS-файлов на нашей главной HTML-странице могут стать проблемой, давайте вернемся к нашему веб-компоненту из главы 2. Если вы помните, этот компонент представлял собой карусель фотографий, которая позволила нам задать список фотографий для навигации по ним, а также метаданные для отображения, такие как заголовок и автор, как показано на рис. 5.1.


**Рис. 5.1 Возвращаясь к компоненту карусели для фотографий из главы 2**

В этом примере весь наш код JS и CSS находился в файле index.html без внешних ссылок. Конечно, все становится более удобным, если переместить этот код во внешние файлы, ссылки на которые мы можем указать. Это довольно типично для веб-проекта без излишеств. Когда мы делаем это, наш HTML-файл становится более управляемым и простым для чтения, как показано в приведенном ниже листинге.

**Листинг 5.1 Пример загрузки веб-компонентов с помощью тега <script>**

```
<html>
<head>
    <meta charset="UTF-8">
    <title>Script Source for Loading Web Components</title>
    <script src="photocarousel.js"></script>
    <link href="photocarousel.css"
          rel="stylesheet"
          type="text/css"/>
</head>
<body>
    <wcia-photo-carousel
            title="Future Vacation Photos"
            author="Ben Farrell"
            photos="https://i.imgur.com/fBmIASF.jpg,https://i.imgur.com/
            3zxD6rz.jpg,https://i.imgur.com/nKBgeLOr.jpg,https://
            i.imgur.com/yVjJZ1Yr.jpg">
    </wcia-photo-carousel>
</body>
</html>
```

Код JavaScript был помещен в связанный файл

Стили были помещены в связанный файл

Теперь, если у вас есть дополнительные веб-компоненты в этом проекте, вы можете добавлять все больше и больше тегов <script> и <link>. В этом нет ничего плохого. Когда мы разрабатываем большой проект, то неоднократно добавляем кучу библиотек, и в качестве последнего шага перед выпуском объединяем это в один файл для JS и один файл для CSS.

Часто, опираясь на ссылки на скрипты в моем HTML-коде, я поддерживаю два отдельных HTML-файла. Один для разработки, а другой для выпуска моего фактического проекта. В случае когда у нас есть много веб-компонентов, которые мы собираемся добавить, наш тег <head> может выглядеть так, как показано в приведенном ниже листинге. Здесь приводится пример со множеством гипотетических ссылок на JS- и CSS-файлы в более полнофункциональном приложении для фотоальбомов.

**Листинг 5.2 Пример файла index.html для разработки**

```
<head>
    <meta charset="UTF-8">
    <script src="photocarousel.js"></script>
    <link href="photocarousel.css" rel="stylesheet" type="text/css"/>

    <script src="photoalbumbrowser.js"></script>
    <link href=" photoalbumbrowser.css" rel="stylesheet" type="text/css"/>

    <script src="loginpanel.js"></script>
    <link href="loginpanel.css" rel="stylesheet" type="text/css"/>

    <script src="socialsharing.js"></script>
    <link href="socialsharing.css" rel="stylesheet" type="text/css"/>

    <script src="photouploader.js"></script>
    <link href=" photouploader.css" rel="stylesheet" type="text/css"/>
</head>
```

Компонент photocarousel (CSS/JS)

Гипотетический компонент photoalbumbrowser (CSS/JS)

Гипотетический компонент loginpanel (CSS/JS)

Гипотетический компонент socialsharing (CSS/JS)

Гипотетический компонент photouploader (CSS/JS)

В то же время наша цель – поместить как можно меньше зависимостей в наш готовый к работе HTML-файл. Мы могли бы запустить задачу с помощью Grunt, Gulp или даже просто NPM, чтобы объединить весь код JS и все стили, чтобы наш готовый к использованию тег `<head>` выглядел так:

```
<head>
    <meta charset="UTF-8">
    <script src="build.js"></script>
    <link href="build.css" rel="stylesheet" type="text/css"/>
</head>
```

Честно говоря, меня пока здесь не совсем все устраивает. Во-первых, мне нужно позаботиться об импорте для каждого веб-компонента, который я использую (CSS и JS). Во-вторых, здесь ничего не делается, чтобы добиться максимальной степени повторного использования кода. Да, я могу указать путь к внешним файлам, содержащим код для моих веб-компонентов, но что, если сами эти файлы должны указывать путь к внешним файлам? Например, в главе 4 мы рассмотрели расширение класса `HTMLElement` для создания игрового базового компонента, дающего нам метод `update`, который запускается с каждым кадром. Нам нужно как-то импортировать класс `GameComponentBase`.

Вы можете сказать: «Ну что же, импортировать этот класс очень просто: мы просто добавим его в список JS-файлов, на которые ссылаемся в нашем теге `<head>`». Опять же, в зависимости от вашего варианта использования этим можно управлять. Контраргумент здесь состоит в том, что вы принимаете вызов, чтобы отслеживать каждую зависимость в своем проекте. Если у вас есть только одна или две зависимости, отлично! Если у вас их 10, или 20, или еще больше, это может стать проблемой.

### 5.1.2 Крошечные сценарии более организованы, но усугубляют проблему со ссылками

Зависимости могут иметь множество форм. Наш класс `GameComponentBase` является основной зависимостью, но вы также можете рассмотреть менее крупные зависимости. Зависимости могут быть такими же небольшими, как вспомогательные методы для управления своим HTML-кодом, или даже очень крохотным конфигурационным объектом. Например, мы могли бы поддерживать модель данных всего проекта, которую мы импортируем в любой веб-компонент или файл JS, нуждающийся в ней:

```
appConfig = {
   rootURL: 'yourserver.com',
   apiVersion: 2,
   login: 'username'
}
```

Это всего лишь простой объект, содержащий данные о том, как мы хотим войти на наш сервер (если мы его используем), но потенциально его можно применять в любом веб-компоненте, который получает данные с этого сервера. Это кусок повторно используемого кода, который необ-

ходим нам везде. Это такой крошечный фрагмент кода JS, – который может быть связан с 20 или 50 другими крошечными кусочками кода, – что если вам все время нужно будет следить за тем, чтобы не забыть включить все это в тег <head>, это может стать проблемой.

### 5.1.3 Включение стилей CSS для самостоятельных компонентов

Прежде чем мы решим эту проблему, давайте сделаем наш класс веб-компонента еще более самостоятельным, заставив его управлять стилями. Мы просто исключим необходимость указывать путь к внешнему файлу CSS, вставляя правила <style> в свойство innerHTML вместе с разметкой HTML. Этот пример, как видно из приведенного ниже листинга, ничего не меняет в нашем компоненте, кроме как оставляет нам только один файл для ссылки при использовании компонента в нашем проекте.

**Листинг 5.3 Добавление стилей в innerHTML нашего компонента**

```
this.innerHTML = '<h2>' + \
   this.getAttribute('title') + '</h2> \
   <h4>by '+ this.getAttribute('author') + '</h4> \
   <div class="image-container"></div> \
   <button class="back">&lt</button> \
   <button class="forward">&gt</button> \
   <style> \
       wcia-photo-carousel { \
       width: 500px; \
       height: 300px; \
       display: flex; \
       padding-top: 10px; \
       flex-direction: column; \
       position: relative; \
       border-color: black; \
       border-width: 1px; \
       border-style: solid; \
   } \
   wcia-photo-carousel h2, h4 { \
       margin-bottom: 0; \
       margin-top: 0; \
       margin-left: 10px; \
   } \
   wcia-photo-carousel .image-container { \
       margin-top: 15px; \
       flex: 1; \
       background-color: black; \
       background-size: contain; \
       background-repeat: no-repeat; \
       background-position: 50%; \
   } \
   wcia-photo-carousel button { \
       cursor: pointer; \
```

HTML-разметка, которая была ранее в нашем компоненте

Стили CSS, добавленные в наш компонент, ранее во внешнем файле CSS

```
        background: transparent; \
        border: none; \
        font-size: 48px; \
        color: white; \
        position: absolute; \
        top: 50%; \
    } \
    wcia-photo-carousel button.back { \
        left: 10px; \
    } \
    wcia-photo-carousel button.forward { \
        right: 10px; \
    }\
</style>';
```

На данный момент у нас получилось нечто очень неплохое – полностью самостоятельный компонент, который нужно включать в код, используя только один тег `<script>`. Справедливости ради стоит отметить, что наш `inner-HTML` становится немного длиннее. В качестве критического замечания можно указать на то, что мы просто переместили некую сложность извне внутрь и сделали эту внутреннюю часть менее управляемой. Не волнуйтесь, мы обсудим это в следующей главе, расширив концепцию модулей, которую изучаем здесь.

**Листинг 5.4 Сокращение зависимостей без ссылок на стили**

```
<head>
    <meta charset="UTF-8">
    <script src="photocarousel.js"></script>
</head>

<body>
    <wcia-photo-carousel
            title="Future Vacation Photos"
            author="Ben Farrell"
            photos="https://i.imgur.com/fBmIASF.jpg,https://
           i.imgur.com/3zxD6rz.jpg,https://i.imgur.com/nKBgeLOr.jpg,https://
           i.imgur.com/yVjJZ1Yr.jpg">
    </wcia-photo-carousel>
</body>
```

**Число зависимостей компонентов сократилось. Вместо двух строк (CSS и JS) теперь только одна – JS**

Честно говоря, глядя на листинг 5.4, все выглядит довольно чисто. Вы никогда не узнаете о том, насколько сложен компонент внутри; вы просто используете его. При наличии всего лишь одного тега `<script>`, используемого для включения определения веб-компонента, здесь практически нечем управлять. Даже если бы тут был только отдельный CSS-файл, как это было раньше, тех, кто использует этот компонент, это могло бы немного сбить с толку. Они могут и не знать, что требуется CSS-файл, или даже где он находится или как называется.

Опять же, наличие одной этой зависимости для использования своего компонента и ее правильная работа, а также управление собственными зависимостями упрощают работу пользователям компонента.

### 5.1.4 Ад зависимостей

Теперь, если вы считаете, что наша цель, чтобы сделать наш веб-компонент полностью самостоятельным, – это добавить больше проблем, чем нужно, я бы не стал вас винить. Прежде чем я продемонстрирую следующую функцию, хочу сказать, что использование тегов `<script>` и ссылки на файлы CSS были именно тем, что я и сделал, и это довольно хорошо сработало.

Проблемы начинаются, когда вам нужно сделать свои веб-компоненты немного умнее и организованнее. Помимо тех небольших зависимостей JS, о которых я упоминал ранее, как лучше всего управлять HTML-кодом внутри своего компонента? Было бы идеально, особенно если кода много, оставлять его за пределами своего класса и вставлять его. Может быть, полезно сохранить разметку как отдельную задачу, не только чтобы она не загромождала ваш класс веб-компонента, но и чтобы несколько участников команды могли работать с одним компонентом, вместо того чтобы использовать один и тот же файл при работе с различными задачами, такими как разметка, логика контроллера или стиль.

Еще одна большая проблема возникает, когда у вас есть собственный веб-компонент внутри вашего веб-компонента. Как лучше всего справиться с этим? Подумайте, что произойдет, если вы создадите приложение, управляемое одним компонентом на вашей странице index.html. У этого компонента может быть восемь компонентов, а, в свою очередь, у каждого из них может иметься еще несколько компонентов.

В этой ситуации вам неожиданно придется:

- отслеживать каждый компонент, используемый в вашем проекте;
- убедиться, что у вас есть теги `<script>` для каждого компонента в вашем файле index.html;
- удалить ссылки на все компоненты, которые больше не используются;
- вести полный список всех компонентов зависимостей, включая их в файл index.html;
- управлять порядком загрузки каждого компонента и зависимости компонента, обеспечивая загрузку сценариев до того, как они понадобятся;
- находиться в тесном контакте с членами команды, учитывая, что все вы будете редактировать один и тот же файл index.html, чтобы управлять всем этим.

Есть множество причин, чтобы выбрать более подходящий способ, чем та неразбериха, которая показана на рис. 5.2, и, к счастью, во всех популярных современных браузерах можно использовать модули JS!

## 5.2 Использование модулей для решения проблем зависимости

Если вы незнакомы с концепцией модулей и хотите получить более подробную информацию, пожалуйста, обратитесь к приложению. Если го-

ворить кратко, мы собираемся отказаться от этого беспорядка с тегами `<script>`, который вынуждены поддерживать, и вместо этого загружать сценарии и компоненты, используя новое ключевое слово `import`. Благодаря этому мы можем уменьшить хрупкость включения JS в наш основной HTML-файл и сделать наши веб-компоненты ответственными за управление собственными зависимостями. Это чрезвычайно чистый и организованный способ работы с пользовательскими компонентами. Чтобы продемонстрировать его, давайте создадим простое приложение на основе веб-компонентов, содержащее несколько различных пользовательских компонентов, чтобы подчеркнуть этот сдвиг в стратегии.

В итоге у нас будет архитектура, подобная той, что представлена на рис. 5.3, управлять которой гораздо проще, чем тем, что показано на рис. 5.2, и которая решает многие наши проблемы.

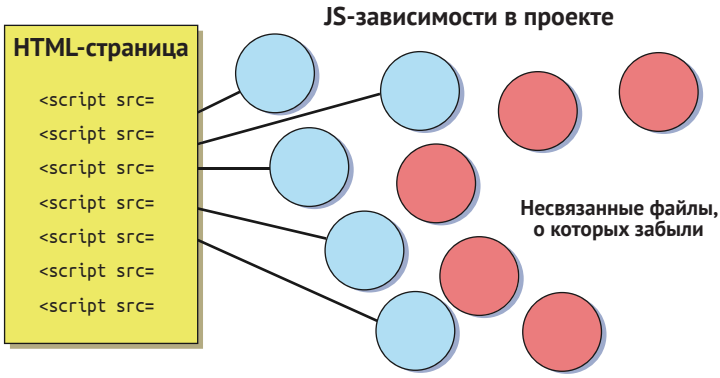



**Рис. 5.2 Использование тегов `<script>` на главной HTML-странице означает, что вы не должны забывать добавлять каждый JS-файл, который используете**

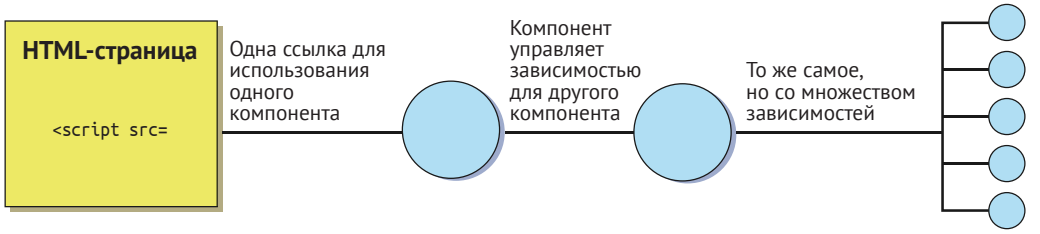



**Рис. 5.3 Страница index.html ссылается только на один компонент, что делает наш код проще, но по-прежнему допускает наличие множества зависимостей**

## 5.2.1 Создание музыкального инструмента с использованием веб-компонентов и модулей JS

В этом примере я хочу создать в нашем браузере струнный инструмент. Я назову его веб-арфой! Каждая струна создается веб-компонентом, и когда ее трогают, она вибрирует и издает звук. Чтобы ничего не услож-

нять, она будет выглядеть просто, как показано на рис. 5.4, но с функциональной точки зрения с ней должно быть весело поэкспериментировать.

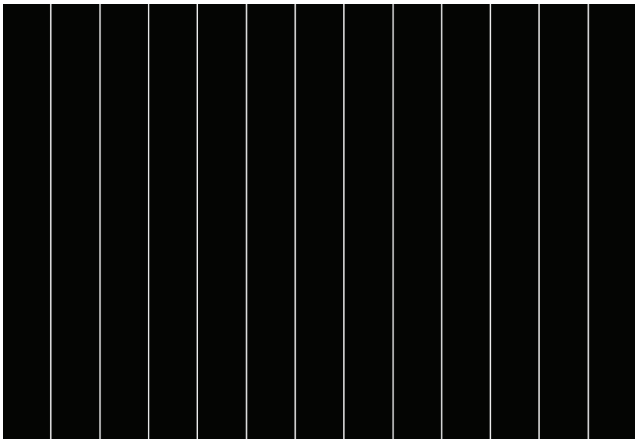


**Рис. 5.4 Вот как выглядит наша веб-арфа. Каждая белая линия – это струна, которая вибрирует и издает шум**

Мы будем использовать модули JS для управления всеми нашими зависимостями, так что загрузим только один файл JS в наш index.html, несмотря на тот факт, что мы используем несколько веб-компонентов. Однако мы не будем писать собственный аудиодвижок – вместо этого импортируем уже существующий прямо в веб-компонент, которому он необходим.

Еще одна замечательная вещь, касающаяся децентрализации наших зависимостей, заключается в том, что наша исходная HTML-страница очень проста. У нее имеется только один компонент, на который есть ссылки, как показано в приведенном ниже листинге.

**Листинг 5.5 Минимальный HTML-файл приложения для нашей веб-арфы**

```
<html>
   <head>
       <title>Web Harp</title>
       <script type="module"
               src="./components/app/app.js">
       </script>
       <link href="csshake.min.css"
             type="text/css" rel="stylesheet">
       <link href="main.css"
             type="text/css" rel="stylesheet">
   </head>
   <body>
       <webharp-app strings="8"></webharp-app>
   </body>
</html>
```

Наша единственная зависимость веб-компонента, которая отвечает за все другие зависимости компонента в приложении

Сторонняя CSS-библиотека csshake, используемая для того, чтобы придать нашим струнам эффект вибрации

CSS для управления стилями на нашей HTML-странице

Сейчас я немного сплутовал со стилями, потому что использую библиотеку для управления анимацией. Таким образом, в этом отношении компоненты данного проекта не самостоятельны, но во всех других аспектах они являются таковыми. HTML-код в каждом компоненте также очень прост. Поскольку мы просто создаем несколько вертикальных линий для обозначения струн в нашей арфе, разметки очень мало. Вся эта простота позволяет нам сосредоточиться на изучении JS-зависимостей с использованием модулей. В следующей главе мы рассмотрим использование модулей для управления HTML и CSS, чтобы сделать все намного чище, когда проекту требуется большее количество стилей и разметки, в отличие от этого.

В этом проекте мы будем управлять тремя компонентами:

- `<webharp-app>` будет содержать все наше приложение и управлять вводом с помощью мыши;
- `<webharp-strings>` будет содержать струны в нашем приложении;
- `<webharp-string>` будет каждой отдельной струной, по которой можно ударять.

Каждый из этих компонентов будет находиться в отдельной папке внутри главной папки компонентов. Это показано на рис. 5.5, где вы также заметите несколько дополнительных файлов, которые помогут нам управлять звуком и анимацией. Мы дойдем до этих дополнений по мере продвижения.

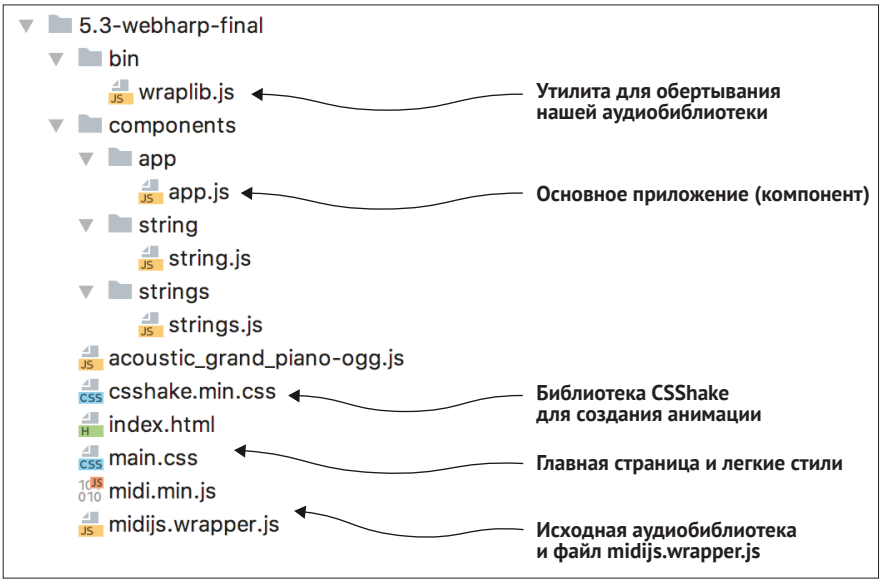



**Рис. 5.5 Файловая структура веб-арфы**

Также обратите внимание, что сначала мы создадим все эти компоненты, чтобы установить минималистичный визуальный макет нашего

приложения. После этого мы наконец поместим компонент `<webharp-app>` на страницу index.html, после чего сможем предварительно просмотреть приложение, прежде чем перейти к добавлению интерактивности, анимации и звука.

## 5.2.2 Начинаем с самого маленького компонента

Давайте начнем с самого малого и сосредоточимся на компоненте `<webharp-string>`. Это будет простая вертикальная линия, которую мы создадим с помощью тега `<div>`. Мы применим стили, чтобы ее высота совпадала с высотой контейнера, а ширина составляла 2 пикселя, и она была белого цвета, как показано на рис. 5.6. Наш компонент берет начало в приведенном ниже листинге.

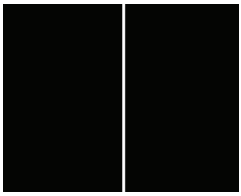


**Рис. 5.6 Отцентрированный компонент `<webharpstring>` на черном фоне**

**Листинг 5.6 Веб-компонент, определяющий отдельную струну нашего инструмента**

```
// file: components/string/string.js

export default class WebHarpString
extends HTMLElement {                          ← Экспортируем класс в качестве модуля
   strum(params) {}                            ← Пустые заполнители, которые будут заполнены позже

   stopStrum() {}

   connectedCallback() {
       this.innerHTML = '<div class="line"></div> \   ← innerHTML нашего компонента
                        <style>\
                            webharp-string > .line { \
                                background-color: white;\
                                height: 100%; \
                                width: 2px; \
                            }\
                        </style>';
   }
}

if (!customElements.get('webharp-string')) {   ← Регистрирует наш пользовательский элемент <webharp-string>
   customElements.define(
'webharp-string', WebHarpString);
}
```

Обратите внимание, что перед определением класса мы используем ключевые слова `export default`. Таким образом мы помечаем наш класс как модуль JS, который можно импортировать в другое место. Наш метод

connectedCallback, который появляется, когда наш компонент добавляется в DOM, не должен вас удивлять, учитывая то, что вы уже прочитали в других главах. Мы просто устанавливаем для innerHTML значение в виде тега  со стилями, о которых я упоминал выше.

Мы смутно знаем, что хотим иметь возможность ударить по этой струне. Если вы когда-либо видели гитарную струну, то, возможно, помните, что она немного вибрирует, но в конце концов останавливается. Итак, поскольку у нас нет реального плана, давайте просто остановимся на методе strum. Можно догадаться, что он принимает какие-то параметры в зависимости от того, какую ноту мы играем и насколько сильно ударяют по струне. Мы вернемся к этому позже, но мы также можем догадаться, что через некоторое время нам нужно перестать играть; следовательно, мы можем добавить пустой метод stopStrum.

## 5.2.3 Импорт и вложение веб-компонента в веб-компонент

Давайте перейдем к компоненту <webharp-strings> (листинг 5.7). Этот компонент будет служить контейнером макета для струн, которые мы планируем разместить горизонтально в приложении. Учитывая, что наш компонент <webharp-app> – это всего лишь тонкая обертка вокруг этого основного визуального компонента, компонент <webharp-strings> – это то, как будет выглядеть конечное приложение, как показано на рис. 5.4.

**Листинг 5.7 Веб-компонент, содержащий несколько струн веб-арфы**

```
// file: components/strings/strings.js

import WebHarpString from '../string/string.js';

export default class WebHarpStrings extends HTMLElement {
    connectedCallback() {
        let strings = '<div class="spacer"></div>';
        for (let c = 0; c < this.getAttribute('strings'); c++) {
                        strings +=
                               `<webharp-string></webharp-string>`;
        }

               strings += '<style>\
                webharp-strings { \
                    height: 100%; \
                    display: flex; \
               } \
               webharp-strings > webharp-string, div.spacer { \
                   flex: 1; \.
               } \
           </style>';

       this.innerHTML = strings;
       this.stringsElements =
               this.querySelectorAll('webharp-string');
   }
}
```

Импортируем отдельный компонент <webharp-string>

Перебираем и добавляем желаемое количество строк, указанное атрибутом strings

Каждая <webharp-string> устанавливается посредством flex-контейнера

Получаем список всех наших струн с помощью querySelector

```
if (!customElements.get('webharp-strings')) {
   customElements.define('webharp-strings', WebHarpStrings);
}
```

В нашем методе connectedCallback мы создадим компоненты <webharp-string> в цикле for, где количество циклов – это количество строк, которое нам нужно. Этот компонент принимает атрибут с именем strings, который передается в цикл for. В результате мы можем сделать арфу с необходимым нам количеством струн.

К счастью, у нас есть модуль CSS flexbox, позволяющий нам очень легко спланировать наш контейнер. Если дать каждой струне правило гибкости flex: 1, наши струны будут равномерно располагаться горизонтально по всему контейнеру, величина которого составляет 100 % от размера нашего приложения. Я также добавил разделитель ; в противном случае наша первая струна начиналась бы на самом краю контейнера и была бы практически не видна. Мы также используем querySelectorAll, чтобы поместить все элементы <webharp-string>, которые мы только что добавили в массив, который мы можем использовать позже, когда добавим интерактивности в наш компонент.

Самое главное, наша самая первая строка:

```
import WebHarpString from '../string/string.js';
```

Из этой книги мы узнали много интересного о веб-компонентах, но мне кажется, что идея импортировать еще один веб-компонент в уже существующий полностью с помощью JS, по сути, поднимает всю экосистему на один уровень вверх. С помощью этого импорта мы позволили нашему приложению узнать, что такое <webharp-string>, и когда мы добавляем его в innerHTML, наш пользовательский элемент действует ровно так, как и должен. Более того, нам не нужно ничего делать в нашем файле index.html, чтобы ссылаться на наш веб-компонент или каким-либо образом регистрировать его. Это просто работает как зависимость компонента, которому она нужна.

Несмотря на то что это простой пример, могут возникнуть ситуации, когда нам нужно использовать здесь <webharp-string>, также внутри другого компонента. При импорте, даже если один и тот же файл импортируется в несколько мест, запрос выполняется только один раз, а последующий запрос на импорт просто использует первый результат.

Кроме того, мы можем защитить наши веб-компоненты, не пытаясь зарегистрировать их снова, если они уже использовались где-то еще, например:

```
if (!customElements.get('webharp-string')) {
   customElements.define('webharp-string', WebHarpString);
}
```

Учитывая это, мы можем с легкостью импортировать наши веб-компоненты куда угодно.

Давайте теперь обернем наши <webharp-strings> конечным компонентом приложения, <webharp-app>, в котором будет находиться все наше

приложение и который будет единственным компонентом, включенным в нашу страницу index.html.

### 5.2.4 Использование веб-компонента для обертки всего веб-приложения

При создании такого веб-приложения можно легко разместить отдельные компоненты на главной HTML-странице. Возможно, вы могли бы поместить их в свой тег `<body>` и написать небольшой код в теге `<script>`, чтобы связать все это воедино.

Вероятно, ваше приложение будет увеличиваться в размерах, вы будете продолжать добавлять компоненты, и логика приложения будет расти. Когда это произойдет, ваш файл index.html с каждым разом будет все сложнее и сложнее поддерживать. Когда придет время извлечь основные элементы и повторно упаковать их в более мелкие компоненты, вероятно, будет иметь место рефакторинг.

Мне бы хотелось предложить что-то еще. Давайте создадим веб-компонент, представляющий все ваше приложение. У него будет та же структура и жизненный цикл, что и у остальных ваших компонентов, и он будет импортировать любые необходимые вам зависимости. Когда этот компонент начнет становиться слишком большим в процессе разработки вашего приложения, вы сможете легко разбить его на более мелкие веб-компоненты. Поскольку они будут иметь структуру, аналогичную остальным вашим компонентам, рефакторинг, вероятно, будет минимальным.

**Листинг 5.8 Веб-компонент приложения «веб-арфа»**

```
// file: components/app/app.js

import Strings from '../strings/strings.js';

export default class WebHarpApp extends HTMLElement {
   connectedCallback() {
       this.innerHTML = '<webharp-strings strings="' +
         this.getAttribute('strings') + '"></webharp-strings>';
   }
}

if (!customElements.get('webharp-app')) {
   customElements.define('webharp-app', WebHarpApp);
}
```

Наш прикладной компонент довольно прост. Как и в случае с нашим последним компонентом, мы импортируем любые дочерние компоненты, которые нам нужны. В этом случае мы импортируем компонент `<webharpstrings>`. Опять же, это контейнер, в котором содержатся все строки нашей веб-арфы. Подобно последнему компоненту, мы принимаем атрибут с именем `strings`, чтобы указать, сколько струн есть у нашей веб-арфы, и передать его компоненту `<webharp-strings>`. И снова мы используем ключевые слова `export default` перед определением нашего класса,

чтобы определить этот компонент как компонент, который можно импортировать. Между тем наш файл index.html безупречен и его легко читать, как показано в приведенном ниже листинге.

**Листинг 5.9 Как сейчас выглядит наш HTML-файл**

```
<html>
   <head>
        <title>Web Harp</title>
        <script type="module"                          ← Одна зависимость компонента приложения
                src="./components/app/app.js">
        </script>
        <link href="main.css" type="text/css" rel="stylesheet">
    </head>
    <body>
        <webharp-app strings="12"></webharp-app>       ← Компонент приложения
    </body>
</html>
```

Просматривая этот файл index.html, обратите внимание: единственное, что находится в теге `<script>`, – это наш компонент `<webharp-app>`. Все остальное является зависимостью нижестоящих компонентов, как видно на рис. 5.7, и, к счастью, здесь нам не нужно волноваться по их поводу. Еще раз важно отметить, что это возможно, потому что наш тег `<script>` имеет тип `module`, что позволяет загружать модули. Это, в свою очередь, дает возможность использовать ключевое слово `import` в рамках всего, что загружается в качестве результата.

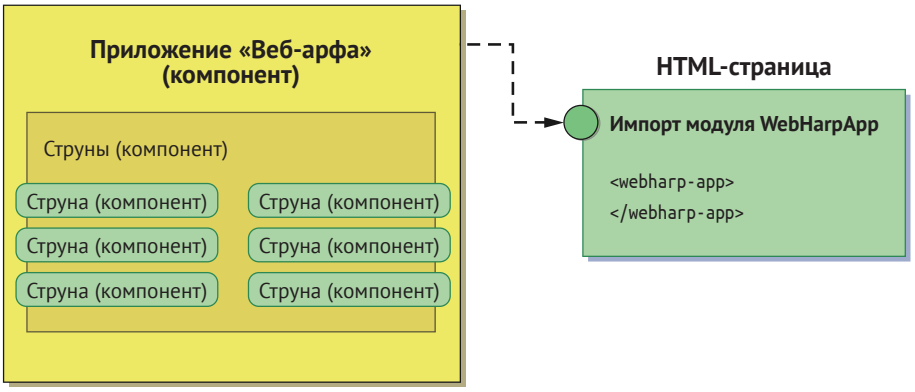



**Рис. 5.7 Вложенные веб-компоненты внутри веб-компонента приложения. Все они импортированы в виде модуля с нашей главной HTML-страницы**

В файле main.css нет ничего особенного: только код для установки размера и цвета нашего приложения и удаления полей, чтобы наше приложение работало по краям окна браузера, как показано в листинге 5.10, и отображалось в нашем браузере, как на рис. 5.8.

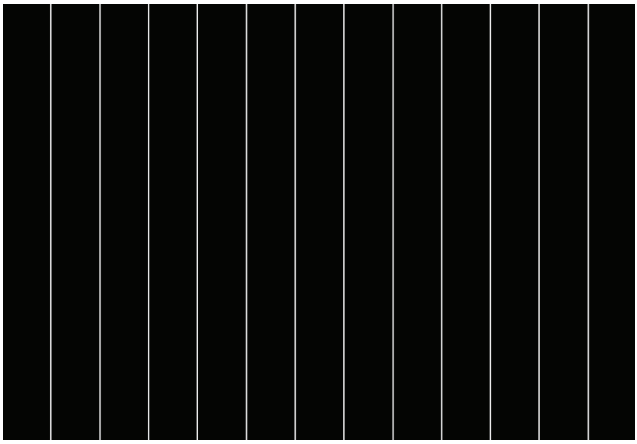


Рис. 5.8 Текущее состояние нашей веб-арфы

**Листинг 5.10 Простые стили для body и webharp-app**

```
body {
    background-color: black;
    margin: 0;
    padding: 0;
}

webharp-app {
    height: 100vh;
    width: 100vw;
}
```

На данный момент мы создали основную структуру нашего приложения. Необходимые компоненты на месте, поэтому мы можем просмотреть их в нашем браузере (не забудьте использовать какой-нибудь локальный веб-сервер, поскольку эти модули могут быть недоступны для загрузки, если вы просто применяете свою файловую систему). Ее внешний вид отныне не будет меняться, но нам нужно добавить немного функционала и интерактивности!

## 5.3 Добавляем интерактивности в наш компонент

Следующий шаг – заставить наше приложение работать! Забегая вперед, скажу, что наша цель – добавить анимацию и звук. Все это будет появляться, когда мы будем играть по струнам арфы с помощью мыши. Для этого мы будем слушать события ввода с использованием мыши, надстроив наш метод `connectedCallback` в компоненте `<webharpapp>`, определенном в components/app/app.js.

**Листинг 5.11 Добавление обработчика событий с помощью жирной стрелки**

```
// file: components/app/app.js

connectedCallback() {
    this.innerHTML = '<webharp-strings strings="' +
        this.getAttribute('strings') + '"></webharp-strings>';
    this.stringsElement =
        this.querySelector('webharp-strings');
    this.addEventListener('mousemove',
        e => this.onMouseMove(e));
}
```

Сохраняет ссылку на элемент <webharpstrings> для последующего использования

Добавляет слушателя события mousemove в наш компонент приложения

Чтобы использовать этот элемент позже, мы запрашиваем и сохраняем ссылку на наш компонент `<webharpstrings>` с помощью `this.querySelector('webharp-strings');`, как видно из листинга 5.11. Самое главное, что мы добавляем слушателя событий к самому компоненту (`this`) для прослушивания событий движения мыши:

```
this.addEventListener('mousemove', e => this.onMouseMove(e));
```

Слушатель, которого мы добавили, использует жирную стрелку, чтобы сохранить область видимости экземпляра класса в `onMouseMove`.

## 5.3.1 Прослушивание событий движения мыши

Конечно, функция, на которую мы указываем, еще не существует. Нам нужно добавить в наш класс `onMouseMove`, чтобы перехватить это событие:

```
onMouseMove(event) {
   this.stringsElement.points = {
       last: this.lastPoint,
       current: { x: event.pageX, y: event.pageY } };
   this.lastPoint = { x: event.pageX, y: event.pageY };
}
```

Здесь мы одновременно фиксируем текущие координаты мыши в переменной и перед этим отправляем текущую и последнюю координаты мыши в наш элемент `<webharp-strings>`. Отправка обеих этих координат позволяет нам получить расстояние, пройденное между нашими движениями, которое мы затем можем использовать, чтобы определить, насколько сильно или быстро мышь бьет по струнам.

## 5.3.2 Передача данных в дочерние компоненты

Обратите внимание, что мы отправляем эти точки с помощью метода чтения из нашего компонента `<webharpstrings>`, поэтому давайте заполним сеттер из файла component/strings.js кодом из этого листинга.

**Листинг 5.12 Отправка точек в компонент <webharpstrings>**

```
// file: components/strings/strings.js

set points(pts) {
    if (!this.stringsElements) { return; }
    if (!pts.last || !pts.current) { return; }
    let magnitude =
        Math.abs(pts.current.x - pts.last.x);
    let xMin
        Math.min(pts.current.x, pts.last.x);
    let xMax = Math.max(pts.current.x, pts.last.x);

    for (let d = 0;
             d < this.stringsElements.length; d++) {
        if (xMin <= this.stringsElements[d].offsetLeft && xMax >=
            this.stringsElements[d].offsetLeft) {
            let strum = {
                power: magnitude,
                string: d
            };
            this.stringsElements[d].strum(strum);
        }
    }
}
```

Проверяет, существует ли stringsElements

Проверяет, что текущие и последние координаты заполнены

Фиксирует скорость удара

Фиксирует самые низкие и самые высокие значения текущей и последней точек

Ударяет по соответствующим струнам, перебирая их

Итак, этот листинг немного сложный, но его можно объяснить. Во-первых, вы, наверное, помните из первых шагов данного примера, что мы посмотрели каждый из наших компонентов <webharp-string> или каждую визуальную струну и сохранили их все в массив, который можно будет использовать позже. Вот теперь мы их используем.

Во-первых, нам, вероятно, следует признать, что теоретически мы могли бы получить событие мыши, поступающее до того, как все настроено, поэтому сначала мы проверим, заполнен ли наш массив, и выйдем из функции, если это не так:

```
if (!this.stringsElements) { return; }
```

Мы также проверим, заполнены ли текущие и последние координаты, особенно потому, что во время первого события mouse-move у нас не будет этой последней координаты:

```
if (!pts.last || !pts.current) { return; }
```

Далее мы будем фиксировать скорость удара по струнам, получая расстояние между двумя *x*, или горизонтальными координатами мыши, также фиксируя самые низкие и самые высокие значения наших текущих и последних координат:

```
let magnitude = Math.abs(pts.current.x - pts.last.x);

let xMin = Math.min(pts.current.x, pts.last.x);
let xMax = Math.max(pts.current.x, pts.last.x);
```

Используя три этих полезных значения, мы можем перебрать элементы массива компонентов <webharp-string>. Если крайний левый край нашей струны попадает между последней и текущей координатами *x*, мы знаем, что нужно ударить по этой конкретной струне. Мы можем отправить числовой индекс того, по какой струне ударили, а также *магнитуду*, или то, насколько сильно это было сделано:

```
for (let d = 0; d < this.stringsElements.length; d++) {
    if (xMin <= this.stringsElements[d].offsetLeft && xMax >=
      this.stringsElements[d].offsetLeft) {
        let strum = {
            power: magnitude,
            string: d
        };
        this.stringsElements[d].strum(strum);
    }
}
```

Теперь у нас есть какая-то интерактивность! К сожалению, наша струна на самом деле пока еще ничего не делает, когда по ней ударяют. Однако мы можем проверить, что все работает, добавив `console.log` в файл components/string/string.js:

```
strum(params) {
   if (this.timer) { clearTimeout(this.timer); }
   this.timer = setTimeout( () => this.stopStrum(), 1000);
   console.log(params);
}
```

Теперь если вы запустите код и откроете свой журнал консоли, то сможете увидеть, по какой именно струне мы ударяем, а также насколько сильно она звучит, непосредственно в консоли.

### 5.3.3 Заставляем наши компоненты вибрировать с помощью CSS

Как и следовало ожидать, нужно добавить две последние вещи: визуальную и звуковую обратную связь (в конце концов, это музыкальный инструмент). Сначала давайте добавим визуальный элемент с оговоркой о том, что на самом деле это не урок по веб-компонентам или модулям JS, а просто нечто, что мы хотим добавить, чтобы заставить этот пример работать. Для этого мы задействуем CSS-проект под названием CSShake, который можно найти по адресу http://elrumordelaluz.github.io/csshake/.

Цель CSShake – создать эффект тряски, что я постарался изобразить на рис. 5.9. Есть множество разных способов, с помощью которых эта библиотека позволяет вам трясти что-либо. Это одна из тех хороших библиотек, которые вы и не думали использовать, но теперь, когда она нам нужна, просто здорово, насколько она продумана до мелочей! Для этой демонстрации мы просто дадим ссылку на CSS-файл и позволим стилям влиять на элементы в нашем компоненте, как обычно. В главе 7 мы из-

меним это понятие и защитим наш веб-компонент от проникновения стилей с помощью Shadow DOM.

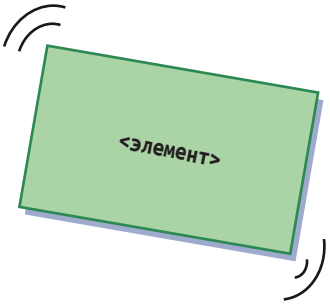



**Рис. 5.9 CSShake берет элемент на вашей странице и анимирует его разными способами**

Сперва давайте добавим в наш HTML-файл это:

```
<head>
   <title>Web Harp</title>
   <script type="module" src="./components/app/app.js"></script>
   <link href="main.css" type="text/css" rel="stylesheet">
   <link href="csshake.min.css" type="text/css" rel="stylesheet">
</head>
```

Чтобы использовать CSShake, мы просто добавляем классы и удаляем их из элементов, которые хотим потрясти:

**Листинг 5.13 Добавляем классы CSShake, чтобы создать эффект вибрирования для наших струн, когда по ним ударяют**

```
// file: components/string/string.js

strum(params) {
    if (this.timer) { clearTimeout(this.timer); }

    let dur = params.power * 10 + 250;
    this.classList.add(
        'shake',
        'shake-constant',
        'shake-horizontal');
    if (dur < 500) {
        this.classList.add('shake-little');
    }
    this.timer = setTimeout( () => this.stopStrum(), dur);
}

stopStrum() {
    this.classList.remove('shake', 'shake-constant', 'shake-horizontal',
      'shake-little');
}
```

Добавляем классы CSShake: базовый эффект «тряски», класс для запуска беспрерывной анимации и горизонтальный тип тряски

Если удар по струнам несильный, тряска будет слабой

Удаляем все классы после того, как по струнам перестают ударять

Как уже упоминалось, здесь мы начинаем с очистки таймера, если таковой существует. Мы также рассчитываем переменную `duration` в мил-

лисекундах, учитывая коэффициент мощности (или скорость, с которой ударяли по струне) и добавляя минимальную линию отсчета в 250 миллисекунд, или четверть секунды.

Для визуализации можно добавить несколько классов CSS, чтобы описать тряску струн.

Здесь используется базовый класс `shake`, а мы хотим, чтобы струна дрожала постоянно и горизонтально. Если удар не очень сильный, мы добавим стиль `shake-little`, чтобы сильный удар отличался от слабого.

Наш удар будет таким же долгим, как и рассчитанная нами продолжительность. Мы остановим удар, когда таймер остановится, и в этот момент удалим все классы, добавленные в компонент `<webharp-string>`.

## 5.4 Обертывание сторонних библиотек в виде модулей

Последнее, что нужно сделать, чтобы завершить наш эксперимент с вебарфой, – это звук! Web Audio API – сложная тема, и то же самое можно сказать о любой генерации аудио- и тональных сигналов в режиме реального времени. К счастью, у нас есть библиотеки JS, которые можно использовать, чтобы скрыть всю эту сложность. Одна из таких библиотек, с которой мне нравилось экспериментировать, – это MIDI.js (https://github.com/mudcube/MIDI.js/). Если вы знакомы со стандартом MIDI, то знаете, что в основном он используется для подключения музыкальных устройств, а не для создания самого звука, но эта библиотека также предлагает генерацию тональных сигналов в реальном времени. Если посмотреть на историю фиксаций, то можно заметить, что последняя фиксация была в 2015 году. Определенно нет ничего плохого в том, чтобы создавать хорошую библиотеку, подобную этой, а затем переходить к другим вещам, после того как она станет достаточно хорошей. Недостатком является то, что в этом проекте не используются новейшие функции языка JS, такие как модули, поэтому мы не можем импортировать эту библиотеку в наш веб-компонент.

### 5.4.1 Инструменты пользовательского интерфейса для обертывания модуля с помощью Node.js

Или все-таки можем? Хотя само по себе это не похоже на подходящий проект, в 2017 году Оуэн Денсмор опубликовал на сайте medium.com статью, в которой обсуждается обертывание JS-зависимостей в виде модулей. В одном из его проектов есть скрипт wraplib.js (https://github.com/backspaces/as-app3d/blob/master/bin/wraplib.js). Как видно на рис. 5.10, этот скрипт берет стороннюю библиотеку и оборачивает ее как модуль, который можно импортировать в ваш проект.

Я вытащил этот скрипт в папку проекта bin. Нам также требуется актуальная библиотека MIDI.js. Обычно мы бы установили MIDI.js из npm, что вы, безусловно, можете сделать, запустив это:

```
npm install midi.js
```

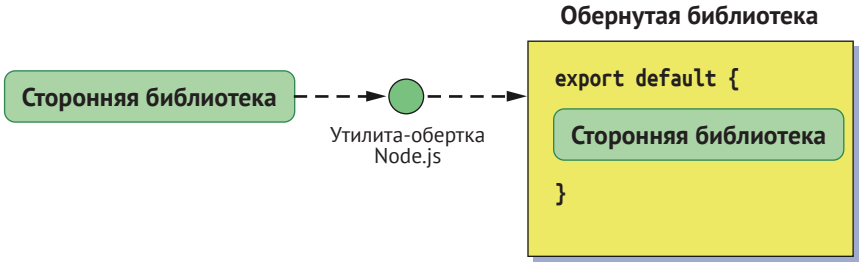



**Рис. 5.10 Использование утилиты Node.js для обертывания сторонней библиотеки в качестве импортируемого модуля**

Однако чтобы было удобнее, я поместил минифицированный файл MIDI.js в папку проекта в репозитории GitHub для этой книги, и мы можем использовать его непосредственно оттуда. Предполагая, что у вас уже установлен Node.js, перейдите в каталог вашего проекта в терминале и наберите это:

```
node ./bin/wraplib.js midi.min.js MIDI > midijs.wrapper.js
```

Файл wraplib.js, содержащий всего 33 строки кода, довольно прост, если вы его откроете. По сути, с помощью первого аргумента вы сообщаете ему, какой файл вы хотите обернуть; второй аргумент – это имя глобальной переменной, в которой хранится библиотека, а затем передается в выходной файл.

## 5.4.2 Не идеально, но работает

Возможно, вы не поверили своим глазам, когда я описал второй параметр. Wraplib – это что-то вроде приема, и распространенным аргументом является то, что он не должен загромождать глобальное пространство имен, как он это делает. В нашем примере, как только мы запустим библиотеку, если вы откроете инструменты разработчика и `window.MIDI`, то увидите библиотеку, которую мы оборачиваем. Такая схема размещения вещей в глобальном пространстве имен немного беспорядочная; но, с другой стороны, это прием, который позволяет нам импортировать библиотеку, не обновлявшуюся в течение нескольких лет. И конечно, данный узловой скрипт для обертывания библиотеки может легко превратить это в процесс сборки пользовательского интерфейса с помощью Gulp, Grunt или даже простого запуска npm. Мы рассмотрим это в главе 12.

## 5.4.3 Использование обернутого модуля для воспроизведения нот

Обернув файл midi.js в качестве модуля, давайте импортируем его и используем! В файле /string/string.js мы инициализируем и загружаем звуковой шрифт пианино в дополнение к нашей предыдущей разметке.

**Листинг 5.14 Инициализация MIDI.js и подготовка к игре на фортепиано**

```
// file: components/string/string.js

connectedCallback() {
    MIDI.loadPlugin({
        soundfontUrl: './',
        instrument: 'acoustic_grand_piano',
        onsuccess: () => this.onLoaded()
    });

    this.innerHTML = '<div class="line"></div> \
                      <style>\
                          webharp-string > .line { \
                              background-color: white;\
                              height: 100%; \
                              width: 2px; \
                          }\
                      </style>';
}
onLoaded() {
    this._ready = true;
}
```

Инициализируем плагин MIDI с помощью инструмента acoustic_grand_piano

Устанавливаем флаг, чтобы указать, что мы готовы, когда плагин будет инициализирован

Как и midi.js, я включил его в корень проекта. Кроме того, вы можете найти его и скопировать из исходного репозитория: https://github.com/mudcube/MIDI.js/tree/master/examples/soundfont. В этом же файле мы добавим функцию `playSound` и запустим ее из нашего метода `strum`.

**Листинг 5.15 Добавляем воспроизведения нот из функции `strum`**

```
// file: components/string/string.js

strum(params) {
    if (this.timer) { clearTimeout(this.timer); }

    let dur = params.power * 10 + 250;
    this.classList.add('shake', 'shake-constant', 'shake-horizontal');
    if (dur < 500) {
        this.classList.add('shake-little');
    }
    this.timer = setTimeout( () => this.stopStrum(), dur);
    this.playSound(params);
}

playSound(params) {
    if (!this._ready) { return; }

    let note = 60 + params.string * 5;
    MIDI.setVolume(0, 127);
    MIDI.noteOn(0, note, params.power, 0);
    MIDI.noteOff(0, note, 0.75);
}
```

Вызываем функцию playSound во время удара по струнам

Возвращаемся, если сторонняя библиотека еще не готова

Устанавливает ноту, которую мы хотим сыграть, в зависимости от выбранной струны

Начинает воспроизведение ноты с той же силой, с какой пользователь ударил по струне

Устанавливает продолжительность воспроизведения на ¾ секунды

Здесь есть некоторые мелкие детали, касающиеся нот, которые мы играем, а именно: мы начнем с ноты си в четвертой октаве и будем подниматься на пять с половиной шагов для каждого индекса струны, по которой ударили. Поскольку мы немного углубляемся в теорию музыки, если вам это не понятно, не страшно, но не стесняйтесь поэкспериментировать с числами.

Кроме того, при воспроизведении ноты мы используем силу удара по струне в качестве скорости (представьте себе клавишу пианино и то, как сильно вы ударяете по ней). Наконец, мы установим длительность ноты в 0,75 секунды (или как долго она будет звучать после нажатия). Я использую здесь постоянное число, потому что звук пианино не имеет большой изменчивости в длине, прежде чем упасть.

## 5.4.4 Больше никакого автовоспроизведения аудио

К сожалению, после того как я изначально написал эту главу, Chrome начал поставлять версии, в которых фоновое аудио не могло воспроизводиться, пока пользователь не предпринимал какие-либо действия, например кликал мышью. Хотя я, конечно, могу понять, насколько раздражает воспроизведение звука, когда об этом не просят, в таких забавных экспериментах, как этот, это несколько угнетает.

Тем не менее нам придется заняться этим, чтобы наша веб-арфа работала! Для этого мы просто заставим пользователя кликнуть по экрану с тегом , занимающим начальную страницу. Наш измененный файл index.html показан в приведенном ниже листинге.

**Листинг 5.16 Файл index.html**

```
<html>
    <head>
        <title>Web Harp</title>
        <script type="module" src="./components/app/app.js"></script>
        <link href="csshake.min.css" type="text/css" rel="stylesheet">
        <link href="main.css" type="text/css" rel="stylesheet">

        <script>
            function clicktostart() {
                document.querySelector('.audio-fix').style.display = 'none';
                document.querySelector('webharp-app').style.display =
                 'inline-block';
            }
        </script>
    </head>
    <body>
        <webharp-app strings="12"></webharp-app>
        <div class="audio-fix"
            onclick="clicktostart()">
            Click Me To Start
        </div>
    </body>
</html>
```

**При щелчке мышью приложение веб-арфы отображается нормально**

**Добавляет элемент div, который занимает всю страницу, заставляя пользователя сделать щелчок мышью**

И наконец, нам просто нужно применить стили к этому тегу , а также позволить ему и приложению накладываться друг на друга с помощью абсолютного позиционирования. В этом листинге показаны стили, которые мы добавили к тому, что у нас было ранее.

**Листинг 5.17 Новые стили CSS**

```
body {
    background-color: black;
    margin: 0;
    padding: 0;
}
.audio-fix {
    position: absolute;
    width: 100vw;
    height: 100vh;
    background-color: #2a2a2a;
    color: white;
    font-size: xx-large;
    display: flex;
    justify-content: center;
    align-items: center;
}

webharp-app {
    height: 100vh;
    width: 100vw;
    display: none;
    position: absolute;
}
```

Стилизуем и позиционируем элемент

Изначально делает так, чтобы веб-арфа не отображалась

Используем абсолютное позиционирование

Теперь пользователь увидит то, что изображено на рис. 5.11, прежде чем сможет запустить веб-арфу.


**Рис. 5.11 Пользователь должен кликнуть мышью для запуска веб-арфы, чтобы включить звук и не быть заблокированным Chrome**

### 5.4.5 Игра на веб-арфе

После этого мы можем перезагрузить нашу веб-арфу, навести мышку на струны и сыграть на ней! Некоторые вещи, безусловно, можно улучшить с помощью этого примера. Наши встроенные HTML-код и стили CSS выглядят довольно уродливо. Кроме того, было бы лучше, если бы мы могли сосредоточиться на логике наших веб-компонентов в классе и поместить CSS и HTML отдельно где-нибудь еще. Все это, безусловно, сделает наш компонент более читабельным и организованным. В следующей главе мы рассмотрим еще одну концепцию ES2015 под названием шаблонные литералы, которая поможет нам разобраться с этим!

## Резюме

Из этой главы вы узнали:

- как веб-компоненты могут управлять своими зависимостями, в том числе и другими веб-компонентами, с помощью модулей, а также как можно избежать путаницы в отношении того, как включить компонент в вашу страницу при наличии одного импорта для использования вашего веб-компонента;
- что веб-компоненты могут быть немного более самостоятельными, помещая стили CSS внутри компонента и избегая необходимости управлять большим количеством CSS-файлов или правилами для множества компонентов в одном CSS-файле;
- как обертывать сторонние библиотеки в виде модуля с использованием Node.js, даже если первоначальный автор никогда не предполагал, что библиотека будет использоваться таким образом, избегая необходимости делать исключение для другого самостоятельного компонента;
- как создать музыкальный инструмент в браузере с использованием веб-компонентов, причем основным приложением является веб-компонент, содержащий дочерние элементы, при этом файл index.html будет оставаться очень маленьким и управляемым.

# Часть II

# Способы улучшить рабочий процесс вашего компонента

Создание собственного HTML-элемента с помощью API пользовательских элементов – довольно удивительная вещь. Снаружи это выглядит как любой другой тег на странице, но внутри он настолько сложен или прост, насколько вам нужно! Теперь пришло время сосредоточиться на работе и погрузиться в процесс создания мощного веб-компонента. Именно здесь мы выходим за пределы пользовательских элементов и рассмотрим остальную часть набора стандартов, образующих веб-компоненты.

Как и в случае с любой новой технологией, веб-компоненты столкнулись с некоторыми ошибками, как, например, ныне устаревшие HTML-импорты; но здесь мы воспользуемся этой ошибкой и разобьем ее на соответствующие части, с которыми вы можете идти дальше. Мы сравним одну из этих частей, шаблонный тег, с другими способами написания внутреннего HTML-кода и CSS-стилей для создания пользовательского интерфейса вашего компонента.

Наконец, в конце этой части мы поговорим о самой известной функции веб-компонента: Shadow DOM. Хотя она и не является обязательной частью веб-компонентов, это существенный сдвиг в отношении того, как мы работаем с DOM браузера. Создание отдельной мини-модели DOM только для вашего компонента чрезвычайно эффективно, поскольку избавляет от разочарований, которые веб-разработчики испытывали на протяжении многих лет, создавая защитный слой вокруг компонента, куда случайным образом не закрадываются стили, а ваши внутренние элементы не изменяются по причине неправильного JS-кода.

Поскольку Shadow DOM является таким мощным свойством и таким изменением по сравнению с тем, что мы делали раньше, стоит упомянуть о нескольких важных предостережениях. Эти предостережения включают в себя полифилинг во все более редкой ситуации, когда ваш браузер не поддерживает веб-компоненты, а также в ситуациях, когда вам действительно нужно проникновение стилей, например при использовании дизайн-системы. Бóльшая часть этого раздела посвящена Shadow DOM, потому что это поистине переломный момент.

# 6

# Управление разметкой

***Эта глава охватывает следующие темы:***

- многострочные строки и их синтаксис;
- шаблонные литералы ES2015 (с переменными);
- шаблонирование HTML/CSS с использованием логики и функций JS;
- шаблонирование с использованием lit-html;
- шаблоны с тегами.

В этой главе мы продолжим опираться на то, что узнали из предыдущих глав, особенно из последней, в которой познакомились с модулями. До сих пор нам удавалось создавать самостоятельные веб-компоненты, которые загружают собственные зависимости, включая другие веб-компоненты. При этом наш файл index.html минимален. Между тем, изучая использование атрибутов и создавая собственный API-компонент в главе 4, мы, по сути, упростили применение веб-компонентов извне.

Тем не менее внутри у нас получился неряшливого вида компонент. Вставка большого количества разметки и CSS-стилей в `innerHTML` компонента работает хорошо, но это не очень удобно для чтения, особенно в части работы с многострочными строками. В этой главе мы рассмотрим эту проблему, и в конце у нас будут безупречные и организованные компоненты как внутри, так и снаружи.

## 6.1 Строки. Теория

Строки являются одной из самых основных вещей в JS. Вы, несомненно, постоянно используете их в каждом аспекте веб-разработки. Зачем переходить к такой простой концепции? Все дело в том, что в ES2015 появилась новая функция JS, которая значительно очищает наши веб-компоненты.

Так в чем же дело? До появления ES2015 существовало несколько разных строковых синтаксисов, которые делали то же самое, – двойные и одинарные кавычки:

```
"Hi I am a string"
```

или

```
'Hi I am a string'
```

Если вы помните из предыдущих примеров, мы пытались засунуть весь наш HTML-код в строку, а затем с помощью этой строки настроить `innerHTML` нашего компонента. Когда HTML-кода немного, то все в порядке:

```
this.innerHTML = '<div class="какой-то класс"></div>';
```

### 6.1.1 Когда innerHTML становится уродливым

Проблема в том, что HTML-код, который вы хотите добавить, начинает увеличиваться. Даже таким кодом с трудом можно управлять:

```
this.innerHTML = '<div><input type="text"/><button>Submit</button></div>';
```

Однако в какой-то момент, если разместить все в одной строке, это станет нечитаемым и трудноуправляемым, поэтому мы начинаем расширять нашу строку, чтобы охватить несколько строк. Давайте рассмотрим форму ввода данных из документации Mozilla по MDN.

**Листинг 6.1 Пример формы ввода в строке JS**

```
this.innerHTML = '<form> \
            <div> \
                <label for="example">Let's submit some text</label> \
                   <input id="example" type="text" name="text"> \
             </div> \
             <div> \
                <input type="submit" value="Send"> \
             </div> \
       </form>';
```

Каждая строка имеет обратную косую черту для перехода к следующей

Альтернативный способ создания многострочных строк в этом листинге немного более многословен:

**Листинг 6.2 Альтернативный способ создания многострочных строк**

Каждая строка заключена в одинарные кавычки и сопровождается знаком + в качестве продолжения

```
this.innerHTML = '<form>' +
                   '<div>' +
                     '<label for="example">Let's submit some text</label>' +
                      '<input id="example" type="text" name="text">' +
                  '</div>' +
                     '<div>' +
                       '<input type="submit" value="Send">' +
                         '</div>' +
                     '</form>';
```

Каждый из этих примеров далек от идеала. Желательно, чтобы HTML-код выглядел как на реальной HTML-странице. Это означает наличие нескольких строк, отступов и, что наиболее важно, отсутствие дополнительных издержек при использовании чего-то особенного, например обратной косой черты или знака +, которые используются для обозначения перехода на следующую строку.

Позвольте мне познакомить вас с немного иным способом написания строки в приведенном ниже листинге: символом обратного штриха (`).

**Листинг 6.3 Заключение строк HTML в символы обратного штриха**

Символы обратного штриха допускают наличие многострочных строк без дополнительного форматирования

```
this.innerHTML = `<form>
                     <div>
                     <label for="example">Let's submit some text</label>
                        <input id="example" type="text" name="text">
                     </div>
                     <div>
                     <input type="submit" value="Send">
                     </div>
                </form>`;
```

Такой способ написания строк называется *шаблонными литералами*, в противоположность способу с использованием *строкового литерала*, который мы применяли раньше. В то время как предыдущий пример решает наши проблемы с читаемостью и рабочим процессом, шаблонные литералы делают гораздо больше, что обычная помощь! Если вы незнакомы с использованием шаблонных литералов или выражений внутри них, обратитесь к приложению.

## 6.2 Использование шаблонных литералов

Используя этот более подходящий способ написания строк, можно себе представить, что могут существовать отличные способы извлечения HTML-кода из разных источников. Возможно, у вас есть какой-то HTML-код, который вы написали в другом HTML-файле. Вы изменили разметку и стили, чтобы они выглядели именно так, как вы хотите, и теперь пришло время интегрировать их. Сейчас мы рассмотрим несколько способов вставки этого HTML-кода.

### 6.2.1 Приложение для создания визиток

Давайте попробуем выполнить небольшое упражнение и сделаем приложение для создания визиток на базе браузера. Идея состоит в том, что мы предоставим несколько различных параметров, которые пользователь может настроить; тогда теоретически они будут готовы и их можно будет печатать. Внутри самой визитки не будет никакой логики или интерактивности; мы просто хотим отобразить статическую визитку с несколькими значениями, такими как имя, должность, адрес электронной почты и т. д., которые можно менять в зависимости от того, какие переменные используются. В отличие от предыдущих упражнений, мы сосредоточимся на макете и стилях заранее. Как только мы закончим, у нас будут результаты, подобные тем, что показаны на рис. 6.1.


**Рис. 6.1 Итоговый результат: визитная карточка, которая позволяет нам настраивать такие значения, как имя, должность и т. д.**

Давайте подумаем о том, что мы делали с веб-компонентами до сих пор. Любая визуальная обработка выполняется путем помещения нашего HTML-кода в JS и настройки свойства `inner-HTML` нашего компонента. Это хорошо, если мы знаем HTML-код и стили CSS, которые хотим использовать, но если макет и стили имеют первостепенное значение, это не лучший способ создания разметки и итерации.

Нет, самый подходящий способ – просто вернуться к основам веб-разработки и создать что-то прямо в HTML-файле с разметкой и стилями CSS. Его легко просматривать и настраивать, не беспокоясь ни о каких веб-компонентах или сложностях, связанных с JS. С точки зрения значений, которые мы хотим заменить, мы можем использовать синтаксис шаблонных литералов прямо в HTML-коде, как показано в листинге 6.5 и в нашем браузере на рис. 6.2.

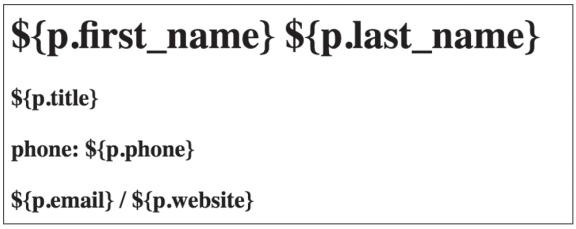



Рис. 6.2 Первоначальный макет визитки без стилей

**Листинг 6.4 Разметка для визитной карточки со встроенными выражениями**

```
<div class="biz-card">
   <div class="logo"></div>
   <div class="top-text">
       <h1>${first_name} ${last_name}</h1>   <-- Здесь идет имя и фамилия
       <h3>${title}</h3>                     <-- Здесь идет название должности
   </div>

   <div class="bottom-text">
       <h3>phone: ${phone}</h3>              <-- Здесь идут номера телефона
       <h3>${email} / ${website}</h3>        <-- Здесь идут адрес электронной почты и веб-сайт
   </div>
</div>
```

Это довольно простая разметка для HTML-файла, но постепенно она начинает усложнять ваш класс веб-компонента вместе со всем остальным. В нашей визитке есть контейнер `<div>` для всей карточки, состоящий из логотипа, за которым следуют имя и название должности. Текст в нижней части визитки содержит номер телефона, адрес электронной почты и веб-сайт.

Что действительно объединяет в одно целое, так это стили CSS. Правила стилей можно увидеть в листинге 6.5, а конечный результат изображен на рис. 6.3.


Рис. 6.3 Визитная карточка, созданная с использованием HTML и CSS перед интеграцией веб-компонентов

**Листинг 6.5 Стили для визитной карточки**

```
<style>
   .biz-card {                                  Основной стиль карточки
       font-size: 16px;
       font-family: sans-serif;
       color: white;
       width: 700px;
       height: 400px;
       display: inline-block;
       border-color: #9a9a9a;
       background-size: 5%;
       background-image:
     ➥url("background-pattern.png");           При копировании этого кода вставьте сюда
       box-shadow: 0 4px 8px 0 rgba(0, 0, 0, 0.2), 0 6px 20px 0 rgba
     ➥(0, 0, 0, 0.19);                         собственное фоновое изображение
   }

   .biz-card .logo {                            Стиль логотипа
       height: 100px;
       margin-top: 10%;
       text-align: center;
       background-image:
     ➥url("biz-card-logo.png");                При копировании этого кода вставьте сюда
       background-size: contain;                свое изображение логотипа
       background-position-x: center;
       background-repeat: no-repeat;
   }

   .biz-card .top-text {                        Стили для остальной части текста
       text-align: center;
   }

   .biz-card .top-text h1 {
       font-size: 2.5em;
       margin-bottom: 0;
   }

   .biz-card .top-text h3 {
       margin: 0;
   }

   .biz-card .bottom-text {
       text-align: center;
       margin-top: 10%;
   }

   .biz-card .bottom-text h3 {
       margin: 0;
   }
</style>
```

Конечно, я потратил какое-то время на итерацию и настройку разметки и стилей, чтобы получить окончательный результат, но в этом

и суть! Если мы будем держать наш визуальный дизайн подальше от веб-компонента и всего нашего проекта, то сможем сосредоточиться на его правильном проектировании и оформлении.

В нашем браузере мы видим сырой синтаксис шаблонных литералов, например `${first_name}`, в контексте эти выражения выглядят несколько уродливо. Несмотря на это, мы можем опробовать различные имена, адреса электронной почты и т. д., чтобы убедиться, что наш дизайн подходит для разных контекстов, прежде чем в конечном итоге поместить в него выражение заполнителя. Таким образом, мы сосредоточились на нашей разметке и стилях вне области видимости, гипотетического приложения и даже самого веб-компонента. При отсутствии JS в поле зрения мы могли бы даже передать это дизайнеру или frontend-разработчику, который, возможно, немного боится кода. Если мы довольны нашей разметкой и стилями, как можно использовать эту визитную карточку в нашем веб-компоненте?

## 6.3 Импорт шаблонов

Это тот момент, когда новые функции JS, которые мы изучали, прекрасным образом объединяют свои усилия. В частности, я говорю о комбинации шаблонных литералов с модулями JS.

Давайте начнем новый проект для размещения веб-компонента визитной карточки. Наш файл index.html, показанный в приведенном ниже листинге, снова будет предельно прост. Он служит лишь для размещения веб-компонента в нашей модели DOM и загрузки JS-модуля.

**Листинг 6.6 Новая страница, на которой размещен наш компонент визитной карточки**

```
<html>
   <head>
       <title>Business Card</title>
       <script
            type="module"
            src="components/bizcard/bizcard.js">
       </script>
   </head>

   <body>
       <biz-card></biz-card>
   </body>
</html>
```

Включает в себя модуль определения веб-компонента

Объявляет веб-компонент на странице

Когда мы закончим – и как только наш компонент выполнит свою работу по размещению HTML-кода и стилей, предоставив нам возможность указать значения, которые мы хотели бы поменять местами для наших выражений-заполнителей, – мы получим результат, изображенный на рис. 6.1 в начале этой главы.

### 6.3.1 Хранение разметки вне логики основного компонента

Далее, конечно же, стоит поработать над нашим классом определения компонента, но с небольшим изменением: мы не будем включать HTML-код или стили в приведенный ниже листинг.

**Листинг 6.7 Компонент «настройщика» визитной карточки**

```
import Template from './template.js';          ← Импортирует наш шаблонный модуль

class BizCard extends HTMLElement {
   connectedCallback() {
       this.innerHTML = Template.render({      ← Использует шаблон для визуализации
           first_name: 'Emmett',                 HTML и CSS в innerHTML компонента
           last_name: 'Brown',
           title: 'Student of all Sciences',
           phone: '555-4385',
           email: 'emmett@docbrown.flux',
           website: 'www.docbrown.flux'
       });
   }
}

if (!customElements.get('biz-card')) {
   customElements.define('biz-card', BizCard);
}
```

Поскольку сейчас у нас нет интерактивности и мы просто отображаем визитную карточку – `<biz-card>` с параметризованным текстом, нам просто нужно установить значения для свойства `innerHTML` нашего компонента.

До того, как приступить к изучению этой главы, мы просто устанавливали для `innerHTML` значения в виде уродливой строки в самом компоненте. Если бы у нас были переменные для вставки в строку, как сейчас, с именем, адресом электронной почты и т. д., было бы еще страшнее! Чтобы сделать наши проекты более четкими и организованными, давайте импортируем наш HTML-код посредством модуля JS.

Вы можете спросить себя: почему модуль JS? Учитывая, что наша цель здесь состоит в том, чтобы наш компонент отображал HTML-код, почему бы не импортировать этот код? К сожалению, JS является единственным допустимым типом модуля, поддерживаемым в настоящее время, но, возможно, в будущем мы сможем импортировать и другие типы. Фактически сейчас Chrome похоже намеревается реализовать модули CSS и HTML, но нужно немного подождать. В следующей главе я вкратце расскажу о теперь уже не существующих HTML-импортах как ранней попытке решить эту проблему, хотя они были импортированы только из другой директивы HTML, а не через JS, как мы пытаемся сделать сейчас.

### 6.3.2 Модуль для HTML и CSS

Все это говорит о том, что использование JS для хранения нашего HTML-кода – довольно сильная вещь и позволяет вставлять логику, когда нам

это нужно. Для начала давайте пойдем простым путем и создадим модуль, который содержит наш шаблон, как показано в этом листинге.

**Листинг 6.8 Определение нашего модуля шаблона**

```
export default {
   render(props) {
       return `${this.html(props)}
               ${this.css(props)}`;
   },
   html(p) { return ``; },
   css(p) { return `` }
}
```

Комбинированный HTML и CSS для визуализации

Функция для возврата будущего HTML-кода

Функция для возврата будущего CSS

Вы сразу заметите, что здесь у меня еще нет HTML-кода или стилей, потому что я хочу поговорить о структуре без разметки, которая бы мешала.

Во-первых, вы могли заметить, что это не класс, в отличие от любого другого модуля, который мы использовали. Конечно, можно свободно использовать здесь класс, если хотите, но на самом деле для этого нет причин, и он просто добавляет дополнительный этап создания экземпляра и его сохранения, если вам нужно использовать этот модуль несколько раз в классе.

Вместо этого, не делая его классом, мы можем сразу использовать его в нашем веб-компоненте, вызвав импорт и содержащуюся в нем функцию:

```
Template.render( . . .
```

Мой метод `render` объединяет как HTML, так и CSS из соответствующих методов.

Конечно, я мог бы просто собрать все разметки в одну; но я думаю, что их удобнее отделять и предлагать немного больше гибкости относительно того, как мы хотим вставлять и использовать любую из них, как показано на рис. 6.4.

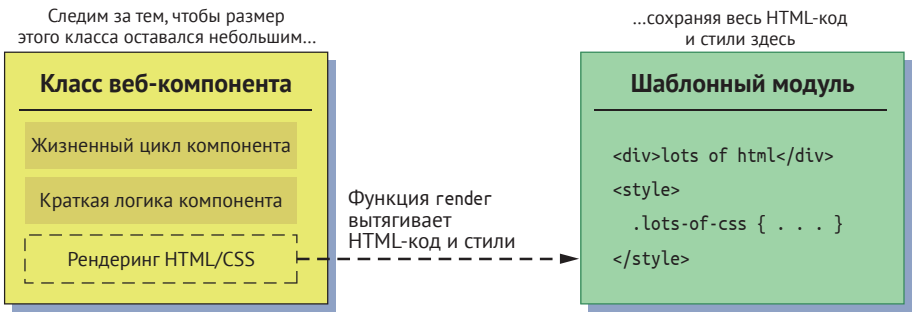



**Рис. 6.4 Сохраняем класс вашего веб-компонента маленьким с помощью еще одного модуля**

Теперь вопрос: как заполнить эти пустые шаблонные литералы содержимым? Очевидный ответ – открыть HTML-файл, который мы создали ранее в этой главе, просто скопировать то, что нам нужно, и вставить. Если вы работаете с одной или несколькими подобными разметками, скопировать и вставить их довольно просто. Но что, если бы вы работали с большой командой производственных ассистентов, которые не касались бы JS-кода или системы управления исходным кодом и производили бы десятки HTML- и CSS-шаблонов, постоянно взаимодействуя с командой проектировщиков? Некоторым это может показаться преувеличением, но я работал над проектами, в которых мы создавали оболочку приложения для размещения множества страниц. Это было что-то вроде опросника, где у всех страниц были достаточно разные макеты, чтобы иметь возможность использовать единый шаблон.

В таких случаях у вас может возникнуть желание автоматизировать процесс, когда вы берете HTML-код, который можно предварительно просматривать автономно в браузере вплоть до подготовки вашего модуля на основе JS. Именно это я и сделал в репозитории на GitHub для этого раздела. Там я создал автоматическую утилиту на базе Node.js, которая берет исходный HTML-файл и автоматически заполняет модуль шаблона, который мы будем использовать в нашей визитной карточке (рис. 6.5).

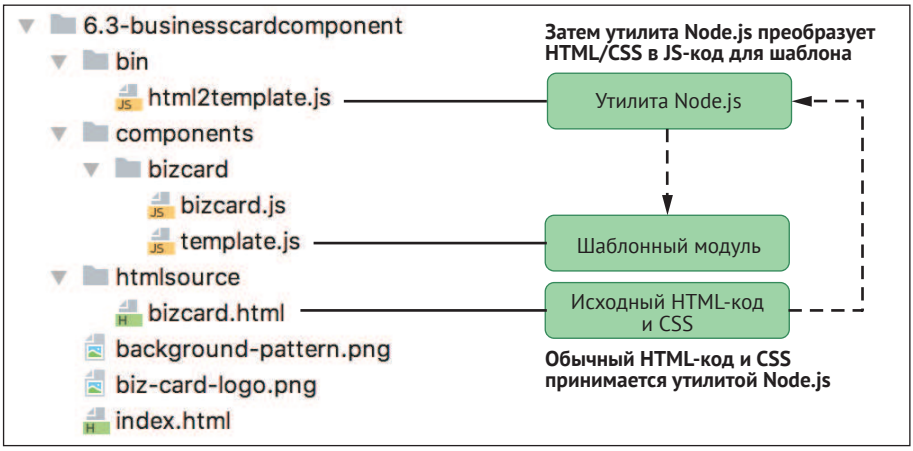



**Рис. 6.5 Пример утилиты на основе Node.js для автоматизации заполнения JS-модуля с помощью существующего HTML-файла**

Недостатком является то, что эти варианты использования, вероятно, настолько отличаются друг от друга, что мой пример служит лишь отправной точкой. Независимо от того, откуда взялся HTML/CSS, автоматизированная эта утилита или нет, наша визитная карточка выглядит как на рис. 6.6.


Рис. 6.6 Результаты интегрированного шаблона на данный момент

## 6.4 Логика шаблона

HTML и CSS под управлением JS обладают большим потенциалом, который может остаться незамеченным при использовании больших блоков разметки (будь то копирование и вставка или же все будет делаться автоматически). Чтобы понять, что я имею в виду, давайте немного поработаем с нашей визиткой. Мы дадим пользователю возможность выбирать нужный вариант из списка логотипов и мозаичного фона для персонализации своей карточки, как показано на рис. 6.7.

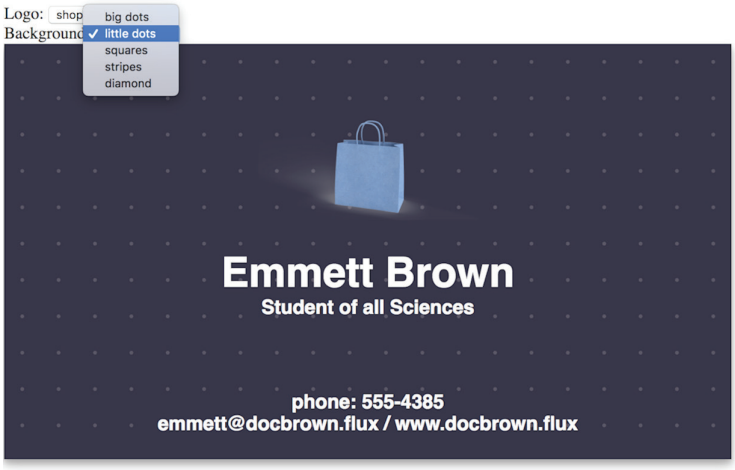



Рис. 6.7 Возможность настройки внешнего вида визитной карточки с помощью логотипа и фона

Для этого я хотел бы кратко коснуться подхода, который более соответствует стилю «сделай сам», а затем перейти к более позднему классу вариантов с большим потенциалом.

### 6.4.1 Создание меню из данных

Начнем с написания кода для генерации списков опций, показанных на рис. 6.8.

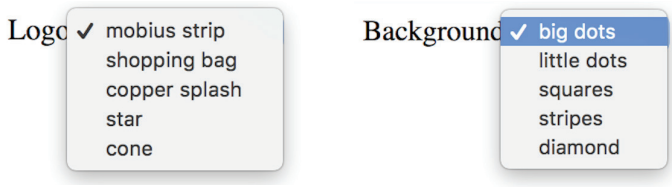



**Рис. 6.8 Два списка опций, которые мы добавим в наш компонент, чтобы настроить визитку**

Мы просто продолжим надстраивать наш компонент, не изменяя сгенерированный шаблон template.js. Для этого мы добавим дополнительные данные, чтобы передать их в метод `Template.render` в нашем определении веб-компонента bizcard.js, как показано в приведенном ниже листинге.

**Листинг 6.9 Передача параметров меню в шаблон**

```
this.innerHTML = Template.render({
   first_name: 'Emmett',
   last_name: 'Brown',
   title: 'Student of all Sciences',
   phone: '555-4385',
   email: 'emmett@docbrown.flux',
   website: 'www.docbrown.flux',

   backgroundChoices: [ <-- Варианты фона визитки
       { name: 'big dots', uri: './images/big-dot-pattern.png'},
       { name: 'little dots', uri: './images/tiny-dot-pattern.png'},
       { name: 'squares', uri: './images/square-pattern.png'},
       { name: 'stripes', uri: './images/stripes-pattern.png'},
       { name: 'diamond', uri: './images/diamond-pattern.png'},
   ],
   logoChoices: [ <-- Выбор логотипа
       { name: 'mobius strip', uri: './images/mobius-logo.png'},
       { name: 'shopping bag', uri: './images/bag-logo.png'},
       { name: 'copper splash', uri: './images/splash-logo.png'},
       { name: 'star', uri: './images/star-logo.png'},
       { name: 'cone', uri: './images/cone-logo.png'},
   ],
});
```

В частности, мы добавили два массива: один для мозаичного фона карточки и один для логотипа в центре. Мы будем использовать их, чтобы заполнить два раскрывающихся списка, обозначенных тегами `<select>` для настройки визитки.

Чтобы заполнить эти списки, мы добавим немного HTML-кода в наш модуль template.js, как показано в приведенном ниже листинге.

**Листинг 6.10 Вызов выражения на базе функции для отображения наших меню**

```
html(p) {
   return `
      <div class="logo-picker">
           Logo: ${this.options(p.logoChoices)}
      </div>
      <div class="background-picker">Background:
           ${this.options(p.backgroundChoices)}</div>

      <div class="biz-card">
       <div class="logo"></div>
       <div class="top-text">
           <h1>${p.first_name} ${p.last_name}</h1>
           <h3>${p.title}</h3>
       </div>

       <div class="bottom-text">
           <h3>phone: ${p.phone}</h3>
           <h3>${p.email} / ${p.website}</h3>
       </div>
   </div>`;
},
```

Заполнение HTML-кода вариантами логотипа

Заполнение HTML-кода вариантами фона

Обратите внимание, что хотя мы используем эти массивы, изначально определенные в определении веб-компонента, простая передача массива ничего не даст, кроме визуализации необработанного массива. Здесь вступает в дело пользовательский метод `options`.

## 6.4.2 Больше логики генерации и более жесткая автоматизация

С этими новыми меню с тегом `<select>` мы делаем нечто иное, используя шаблонные литералы.

Вместо того чтобы просто использовать переменную для заполнения, мы используем функцию из нашего шаблона с возвращаемым значением, содержащим строку с меню, как показано на рис. 6.9. Мало того, мы используем одну и ту же функцию для создания обоих меню, отличающихся только списком параметров, которые мы передаем, как показано в приведенном ниже листинге.

**Листинг 6.11 Функция для преобразования массива параметров в параметры меню**

```
options(list) {
   let choices = ``;
   for (let c = 0; c < list.length; c++) {
```

Перебираем список вариантов

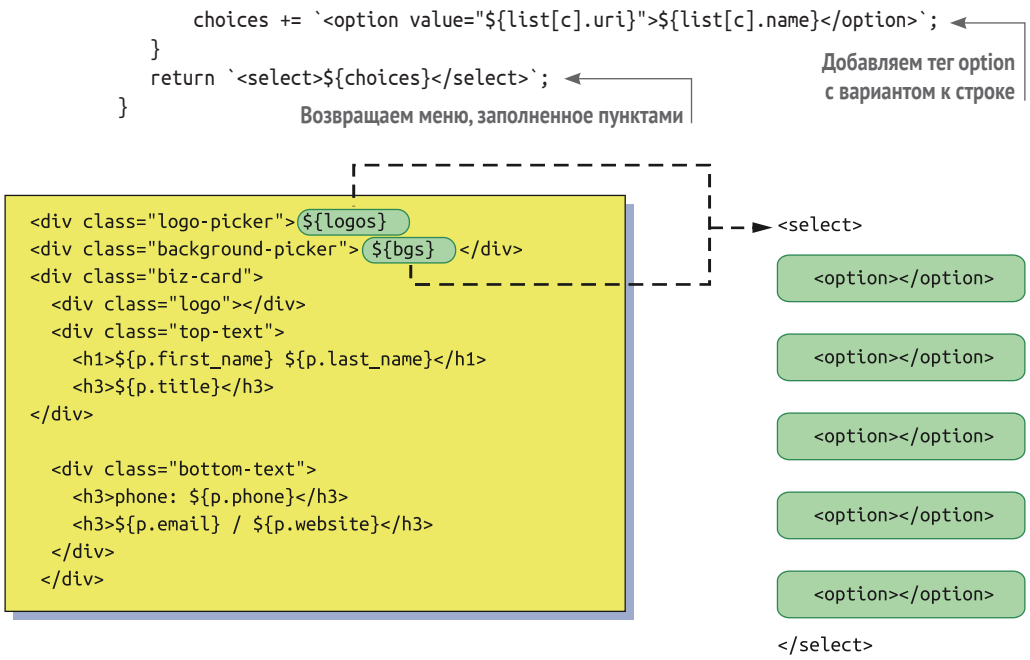



**Рис. 6.9 Вызов функции JS из HTML-шаблона для генерации пунктов меню из массива**

Далее нам нужно, чтобы компонент нашей визитной карточки реагировал на изменения в выпадающих меню. Это окольным путем приводит нас к последней вспомогательной функции, которую я хотел бы добавить в свои файлы template.js.

## 6.5 Кеширование элементов

Учтите, что нам нужно будет добавить слушателей событий в наши меню для прослушивания изменений в них. Для этого нам, естественно, понадобятся ссылки на них. Конечно, после настройки свойства `innerHTML` в функции `connectedCallback` из нашего определения веб-компонента достаточно просто сделать следующее:

**Листинг 6.12 Добавляем слушателей событий, чтобы реагировать на изменения в выпадающих меню**

```
this.querySelector('.logo-picker select').addEventListener( 'change', e =>
    this.updateGraphics() );
this.querySelector('.background-picker select').addEventListener(
    'change', e => this.updateGraphics() );
```

Добавляем слушателя событий для отслеживания изменений логотипа

Добавляем слушателя событий для наблюдения за изменениями фона

Однако этот метод далеко не идеален. Во-первых, выбор запроса отнимает время у ЦП.

Эти две строки вряд ли являются проблемой и появляются только один раз, чтобы добавить слушателя событий. С другой стороны, давайте посмотрим на функцию `updateGraphics` в классе веб-компонента bizcard.js, которая показана в приведенном ниже листинге.

**Листинг 6.13 При смене логотипа или фона выполняется повторная визуализация обоих**

```
updateGraphics() {
   this.querySelector('.biz-card')
       .style.backgroundImage = `url("${this.querySelector(
       '.background-picker select').value}")`;
   this.querySelector('.logo')
       .style.backgroundImage =`url("${this.querySelector(
       '.logo-picker select').value}")`;
}
```

Еще один метод querySelector для получения карточки

И еще один метод querySelector для получения логотипа

Эти две строки кода появляются при изменении любого из меню. Сначала мы выбираем элемент контейнера визитной карточки и присваиваем `backgroundImage` значение фонового меню, выбранного запросом. И еще раз мы делаем это для логотипа.

Да, мы выполняем дополнительную обработку, выбирая запрос четыре раза при каждом изменении меню. Если бы все было намного сложнее, это, вероятно, было бы серьезной проблемой. В частности, с этим примером нет никаких особых проблем, но при наличии ситуаций, когда вам нужна оптимизация, безусловно, стоит изучить эту дополнительную обработку!

## 6.5.1 Не заставляйте меня использовать метод querySelector в моем компоненте

Обратите внимание на отсутствие читабельности в этих операторах и помните, что разметка находится в модуле template.js, а не здесь, в классе веб-компонентов. Также учтите, что разметка неизбежно изменится, когда мы будем снова и снова возвращаться к нашему проекту, и, что еще хуже, выбор запросов может стать более сложным, когда наш HTML-код станет сложнее.

Из-за всех этих проблем мне нравится кешировать свои DOM-элементы, используя метод внутри самого модуля template.js. Когда функция находится рядом с функцией `html() { . . . }`, я могу легко ссылаться на разметку для создания своих селекторов. Простой пример с тегами `<form>` показывает это отображение на рис. 6.10.

В нашем примере с визиткой я могу поместить в модуль template.js это:

**Листинг 6.14 Используем метод `querySelector` один раз и сохраняем ссылки для последующего применения**

```
mapDOM(scope) {
   return {
```

Параметр scope – это ссылка на веб-компонент

```
        logoPicker: scope.querySelector(
                '.logo-picker select'),
        backgroundPicker: scope.querySelector('.background-picker select'),
        logo: scope.querySelector('.logo'),
        background: scope.querySelector('.biz-card')
    }
},
```

Используем метод querySelector один раз и сохраняем ссылки в объект

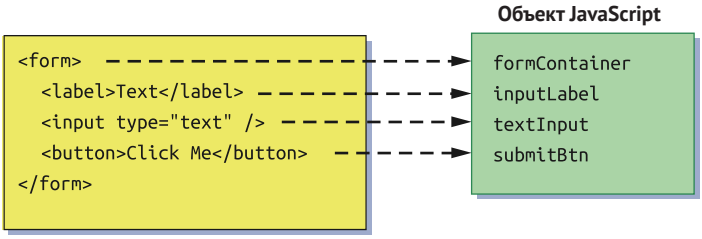



**Рис. 6.10 Отображение и кеширование элементов из метода `querySelector` в объект JS для удобства**

Здесь мы одновременно кешируем элементы и создаем простые ссылки на них.

Кроме того, эти простые ссылки могут быть настолько постоянными, насколько вам нужно! То есть, например, если мне нужно поменять селектор на `logoPicker`, я могу сделать это прямо здесь. Может быть, он поменяется на `scope.querySelector('.card-container > div.logo-chooser select')`. Мой селектор стал немного сложнее, но мой веб-компонент может и дальше ссылаться на свойство `logoPicker`.

Единственная небольшая сложность здесь заключается в том, чтобы перейти в `scope`. Поскольку функция `mapDOM` находится в другом модуле и не является экземпляром класса, у нее нет ссылки на класс веб-компонента. Чтобы решить эту проблему, можно просто передать ссылку на наш веб-компонент или `this` в функцию `mapDOM`, как это делается в приведенном ниже листинге.

**Листинг 6.15 Убираем метод `querySelector` из логики контроллера компонента**

```
import Template from './template.js';

class BizCard extends HTMLElement {
    connectedCallback() {
        this.innerHTML = Template.render({. . . });

        this.dom = Template.mapDOM(this);
        this.dom.backgroundPicker.addEventListener(
                'change', e => this.updateGraphics() );
        this.dom.logoPicker.addEventListener(
                'change', e => this.updateGraphics() );
        this.updateGraphics();
    }
```

Параметры не изменились и были опущены для краткости

Отображает наши элементы в объект JS

Добавляет слушателя к элементу logoPicker, на который ссылается наш объект элемента

```
    updateGraphics() {
        this.dom.background.style.backgroundImage =
                `url("${this.dom.backgroundPicker.value}")`;
        this.dom.logo.style.backgroundImage =
                `url("${this.dom.logoPicker.value}")`;
    }
}

if (!customElements.get('biz-card')) {
    customElements.define('biz-card', BizCard);
}
```

Устанавливает фоновое изображение элемента логотипа, на который ссылается наш объект элемента

Видно, что в нашей функции `connectedCallback` мы присваиваем `this.dom` объекту, где находятся наши кешированные элементы, и можем ссылаться на него где угодно в нашем классе. С помощью простых имен свойств, которые имеют смысл для нашего контекста, мы избегаем корявого кода, а также (минимального) снижения производительности при использовании метода `querySelector()`.

В последнее время я поддерживаю более автоматизированные подходы, которые используют атрибут, чтобы «пометить» каждый элемент, а затем используют скрипт для итерации и создания подобного сопоставления, не определяя его явно в вашем коде. С этим подходом можно познакомиться в моем репозитории на GitHub, посвященном этому разделу.

## 6.6 Умные шаблоны

Когда я пишу эту книгу, в проекте Polymer происходит нечто очень интересное. Напомню, что библиотека Polymer от компании Google, чьи версии выходили в период с 2013 по 2018 год, была разработана для работы с веб-компонентами. В то время веб-компоненты были настолько «неотесанными», и вам действительно нужна была библиотека или фреймворк, которые помогли бы вам идти в ногу с меняющимися достижениями и спецификациями.

Интересно, что после трех основных выпусков библиотека Polymer устарела и перешла в режим обслуживания. Этот проект в целом существует и очень активен, поскольку команда отделяет от проекта более мелкие и более целевые инструменты и библиотеки.

Два ярких примера тому – lit-html и LitElement. Оба они готовы к промышленной эксплуатации и имеют версию 1.0 (хотя технически версия LitElement была указана как 2.0, чтобы не вступать в конфликт с другим проектом LitElement, когда команда сменила название на NPM). Я не буду вдаваться в детали касательно LitElement, потому что, как ни крути, это тонкая обертка для всего, что мы узнали из данной книги! Итак, концепции в основном одни и те же. Команда Polymer добавила некоторые детали, такие как расширенный API жизненного цикла, а также автоматическая рефлексия (где свойства и атрибуты всегда синхронизированы).

Один из более сложных наборов функций в LitElement фактически выполняется через lit-html. Проект lit-html – это набор импортируемых модулей для управления вашим HTML-кодом и стилями CSS, как

те, что мы использовали начиная с пятой главы. Из-за этого его трудно назвать «библиотекой». Когда я думаю о JS-библиотеке, такой как React, или фреймворке, таком как Angular, обычно имею в виду большой монолитный файл, который мог бы взять на себя весь мой проект, и о том, что я должен был бы работать в стиле React или Angular.

Нет, и lit-html, и LitElement относятся к типу «opt-in per component». Это означает, что я могу использовать их для одного компонента, но, возможно, остальные мои компоненты в моем проекте их использовать не будут. В случае с lit-html, если есть функция, которой я не буду пользоваться, я просто не буду импортировать этот модуль, и он не будет увеличивать размер файла моего проекта.

Я думаю, что за этим подходом будущее веб-компонентов (а возможно, и интернета): легкие библиотеки типа «opt-in», которые легко можно заменить, в отличие от больших монолитных фреймворков или библиотек, заставляющих вас делать какие-то вещи, как удобно им. На самом деле подобных инструментов много; но с учетом того, насколько значимой является команда Polymer в пространстве веб-компонентов, мы, скорее всего, увидим какое-то серьезное применение для lit-html и LitElement со стороны разработчиков веб-компонентов, особенно потому, что они прокладывают путь к максимальной кросс-браузерной поддержке всех функций веб-компонентов, вплоть до IE11.

## 6.6.1 Использование lit-html

В случае с lit-html, как и любой другой JS-библиотекой, существует кривая обучения. У lit-html хорошо получается визуализировать HTML и CSS для вашего компонента, который вы определили в строке, как мы это делали до сих пор. Одним из преимуществ использования lit-html является то, что она заменяет только то, что изменилось при рендеринге, а это может привести к повышению производительности. Вспомните наш предыдущий пример настройки свойства `innerHTML` нашего компонента, где мы заменяем все содержимое. При наличии больших деревьев DOM это может привести к сбоям в производительности, если не проявить смекалку. Помимо простой визуализации ваших строк HTML-кода и стилей CSS, lit-html предлагает расширенные возможности для работы с шаблонами. Давайте кратко рассмотрим некоторые из этих функций.

Вначале вы обычно используете команду `npm install` для установки проекта:

```
npm install lit-html
```

Однако, чтобы было проще, я просто скопировал все это в каталог веб-компонента bizcard-lithml в репозитории на сайте GitHub для этой книги.

## 6.6.2 Модуль repeat

Первое, что нужно попробовать сделать, – это избавиться от нашей пользовательской JS-функции для создания наших меню с тегом `<select>`. Для этого мы будем использовать модуль lit-html `repeat` наряду со стандарт-

ным модулем html. При этом мы можем извлечь массив данных и повторно заполнить HTML-код, как показано на рис. 6.11. Мы сделаем это, добавив импорты lit-html в наш модуль template.js и изменив нашу разметку, чтобы включить в нее повторяющийся блок HTML, как показано в листинге 6.16.

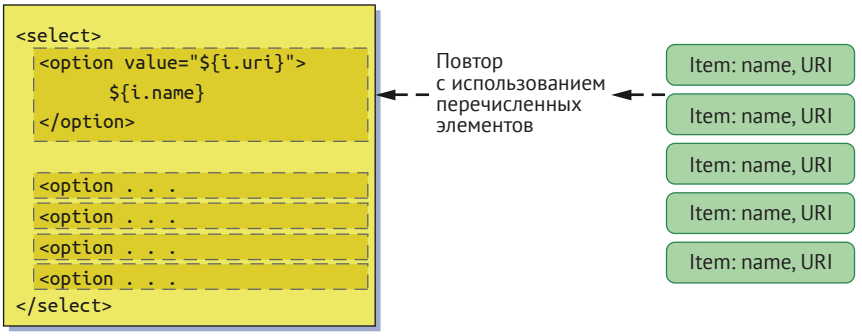



**Рис. 6.11 Используем массив элементов для повторения фрагмента HTML-кода, заполняя меню**

**Листинг 6.16 Использование lit-html для повтора HTML-кода меню**

```
import {html} from './lit-html/lit-html.js';
import {repeat} from './lit-html/directives/repeat.js';
export default {
   render(props) {
       return html`
               <div class="logo-picker">Logo:
                   <select>
                       ${repeat(
                           props.logoChoices,
                           (i) => i.id, (i, index) => html`
                             <option value="${i.uri}">${i.name}</option>`)}
                   </select>
               </div>
               <div class="background-picker">Background:
                   <select>
                       ${repeat(
                           props.backgroundChoices,
                           (i) => i.id, (i, index) => html`
                             <option value="${i.uri}">${i.name}</option>`)}
                   </select>
               </div>
               ${this.html(props)}
               ${this.css(props)}`;
    },
```

Повторяет пункты меню, чтобы создать меню для выбора логотипа

Повторяет пункты меню, чтобы создать меню для выбора фона

Обратите внимание, что наша структура практически не изменилась! Мы по-прежнему указываем на функции html() и css(), чтобы использовать нашу исходную разметку. Однако нам пришлось немного изменить

эти методы. Чтобы рассматривать разметку как HTML-код, а не как необработанный текст в lit-html, нам нужно использовать более продвинутую функцию шаблонных литералов под названием *шаблоны с тегами*. Эти шаблоны объединяют шаблонный литерал и функцию в кратком синтаксисе, который позволяет функции осуществлять парсинг литерала, как показано на рис. 6.12.


**Рис. 6.12 Элементы тегированной функции и принцип ее работы с шаблонным литералом**

В этом примере `html` – функция, предоставляемая lit-html, а наш шаблонный литерал – это наша разметка или стили CSS. На рис. 6.13 видно, что мы вкладываем эти тегированные шаблоны в нашу пользовательскую функцию `render`.

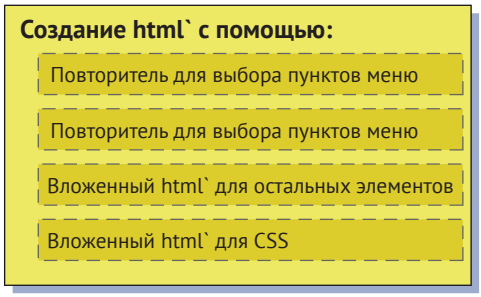



**Рис. 6.13 Наш шаблонный модуль, использующий lit-html**

## 6.6.3 Нужно ли вам использовать это?

Таким образом мы полностью убрали нашу пользовательскую функцию генерации меню! Вопрос, который нужно задать себе при использовании любого стороннего модуля или библиотеки, состоит в том, стоило ли оно того.

Теперь вы зависите от внешнего проекта, хотя он недавно и стал стабильным, учитывая, что он только что достиг версии 1.0. Кажется, что его синтаксис немного подвержен ошибкам, пока вы не поработаете с ним достаточно, и потенциально его сложновато отладить. Тем не менее у lit-html есть один большой плюс – это простая функция `render`. При использовании lit-html для визуализации HTML-кода обновляются только те фрагменты, которые были изменены. Сравните это с обычной настройкой `innerHTML`, которую мы делали раньше, – налицо более высокие затраты на производительность, для того чтобы перенести всю эту разметку в DOM, поэтому в случаях, когда необходимо обновить большое количество HTML-кода (особенно если это происходит часто или вы не уверены, что требуется обновление), lit-html может предоставить реальное преимущество.

Как и в случае с любым сторонним модулем или библиотекой, чем больше вы используете его, тем больше оно становится вашей второй натурой. Взятый отдельно, lit-html определенно не стоил бы того, чтобы создавать с его помощью наши меню в этом крошечном веб-компоненте. Что, если бы у нас были десятки или сотни веб-компонентов, у многих из которых были бы повторяющиеся элементы, созданные с помощью данных? Также что, если бы вы могли передать эти шаблоны разметки другому frontend-разработчику, который не хочет трогать логику вашего приложения или писать собственный JS-код для обработки этих сгенерированных данными элементов? Все это может быть хорошей причиной для использования только такой функциональности с «повтором».

## 6.6.4 Внедрение слушателей событий в разметку

Еще одна возможность, которую предлагает lit-html, – это добавление слушателей в свою разметку. Из нашего примера с визиткой вы, наверное, помните, что мы добавляем слушателей событий в наш класс веб-компонентов вручную:

```
this.dom.backgroundPicker.addEventListener( 'change', e =>
     this.updateGraphics() );
this.dom.logoPicker.addEventListener( 'change', e => this.updateGraphics() );
```

Если бы у нас был длинный список элементов, к которым нам нужно было бы добавить слушателей событий, это мог бы быть довольно большой кусок кода, который можно использовать только для настройки. Мы можем позволить lit-html помочь нам в этом, добавив слушателей событий прямо в шаблон и вызвав функцию в нашем веб-компоненте, как показано на рис. 6.14.

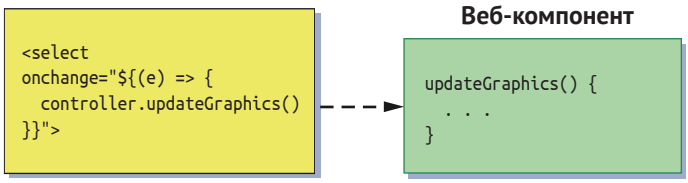



**Рис. 6.14 Встроенный слушатель событий, используемый для прослушивания изменений в меню и вызов функции в нашем веб-компоненте**

Для начала давайте настроим метод `connectedCallback` в нашем классе веб-компонентов, используя код из листинга 6.17.

**Листинг 6.17 Удаление слушателей событий в нашем компоненте для подготовки к lit-html**

```
connectedCallback() {
   render(Template.render(this, {
       first_name: 'Emmett',
       last_name: 'Brown',
```

Создаем шаблонный литерал для передачи его в функцию render

```
        title: 'Student of all Sciences',
        phone: '555-4385',
        email: 'emmett@docbrown.flux',
        website: 'www.docbrown.flux',

        backgroundChoices: [
            { name: 'big dots', uri: './images/big-dot-pattern.png'},
            { name: 'little dots', uri: './images/tiny-dot-pattern.png'},
            { name: 'squares', uri: './images/square-pattern.png'},
            { name: 'stripes', uri: './images/stripes-pattern.png'},
            { name: 'diamond', uri: './images/diamond-pattern.png'},
        ],
        logoChoices: [
            { name: 'mobius strip', uri: './images/mobius-logo.png'},
            { name: 'shopping bag', uri: './images/bag-logo.png'},
            { name: 'copper splash', uri: './images/splash-logo.png'},
            { name: 'star', uri: './images/star-logo.png'},
            { name: 'cone', uri: './images/cone-logo.png'},
        ],
    }), this);

    this.dom = Template.mapDOM(this);
    this.updateGraphics();
}
```

Передаем область видимости (this), чтобы lit-html знал, куда писать содержимое

Здесь есть только два изменения. Сначала мы удалили добавление слушателей событий. После добавления слушателей событий в нашу разметку с использованием lit-html, как показано в листинге 6.15, они нам больше не понадобятся. Во-вторых, мы хотим дать lit-html ссылку на наш веб-компонент, чтобы запустить нашу функцию `updateGraphics`, поэтому передаем `this` в качестве первого параметра в функцию `Template.render`, где вторым параметром являются все данные, которые мы передаем.

Теперь перейдем к магии lit-html. В листинге 6.18 мы хотим использовать стандартное событие `change` для прослушивания изменений в меню; но в сочетании с lit-html мы будем использовать выражение `@` из этой библиотеки для создания правильных привязок. Таким образом, мы можем вставить встроенную функцию, указывающую на наш веб-компонент, на который ссылается переменная с именем `controller`.

**Листинг 6.18 Внедрение слушателей событий в разметку с использованием lit-html**

```
render(controller, props) {
    return html`
            <div class="logo-picker">Logo:
                <select @change="${(e) => {
                  controller.updateGraphics()} }">
                    ${repeat(props.logoChoices, (i) => i.id, (i, index) =>
                     html`<option value="${i.uri}">${i.name}</option>
                `)}</select>
            </div>
            <div class="background-picker">Background:
```

Слушатель для меню с логотипом, добавленный с помощью lit-html

Слушатель для фонового меню, добавленный с помощью lit-html

```
            <select @change="${(e) => {
              controller.updateGraphics()} }">
                ${repeat(props.backgroundChoices, (i) => i.id, (i, index) =>
                 html`<option value="${i.uri}">${i.name}</option>
            `)}</select>
          </div>
          ${this.html(props)}
          ${this.css(props)}`;
},
```

Однако в нашем простом примере мы снова уменьшили код в нашем веб-компоненте только на две строки, когда удалили слушателей событий. Стоило ли оно того? Наверное, нет, но для более масштабных проектов с участием команды разработчиков это определенно стоило бы сделать!

Кроме того, по мере продвижения работы команды Polymer над проектом Polymer (www.polymer-project.org) можно увидеть, что lit-html вместе с LitElement становится довольно распространенным решением для веб-компонентов.

## 6.7 Обновление ползунка

Прошло немного времени с тех пор, как мы обновили компонент-ползунок, над которым начали работать еще во второй главе. Теперь, имея возможность импортировать шаблон и элементы кеша, мы можем немного улучшить его и сделать так, чтобы он был пригоден для совместного использования как настоящий полноценный компонент!

Мы можем начать с выделения нескольких файлов. До настоящего момента HTML-код, JS-скрипты и стили CSS ползунка были заключены в один HTML-файл. Наша цель – создать демонстрационный HTML-файл, чтобы продемонстрировать работу ползунка, JS-файл исходного класса компонента и, наконец, шаблонный модуль для хранения HTML-кода и стилей компонента. На рис. 6.15 показана файловая структура нового проекта.


**Рис. 6.15 Три файла для нашего компонента**

Вероятно, имеет смысл начать с модуля template.js, используя идеи, которые мы только что рассмотрели. Мы извлечем HTML-код компонента, который ранее был в классе компонента, и стили, которые ранее находились в комплексном HTML-файле ползунка. В приведенном ниже листинге этот модуль показан в полном объеме.

**Листинг 6.19 Шаблонный модуль компонента**

```
export default {
    render() {
        return `${this.css()}
```

```
            ${this.html()}`;
    },

    mapDOM(scope) {                                  Кеширует элементы компонента
        return {
            overlay: scope.querySelector('.bg-overlay'),
            thumb: scope.querySelector('.thumb'),
        }
    },

    html() {                                         Перемещает HTML-код из класса
        return `<div class="bg-overlay"></div>       компонента в этот модуль
                <div class="thumb"></div>`;
    },

    css() {                                          Перемещает стили из старого HTML-файла
        return `<style>                              в этот компонент/модуль
                    wcia-slider {
                        display: inline-block;
                        position: relative;
                        border-radius: 3px;
                    }

                    .bg-overlay {
                        width: 100%;
                        height: 100%;
                        position: absolute;
                        border-radius: 3px;
                    }

                    .thumb {
                        margin-top: -1px;
                        width: 5px;
                        height: calc(100% - 5px);
                        position: absolute;
                        border-style: solid;
                        border-width: 3px;
                        border-color: white;
                        border-radius: 3px;
                        pointer-events: none;
                        box-shadow: 0 4px 8px 0 rgba(0, 0, 0, 0.2), 0 6px
                        20px 0 rgba(0, 0, 0, 0.19);
                    }
                </style>`;
    }
}
```

Вы, возможно, не видели стили этого компонента начиная со второй главы, поэтому можете не заметить, что некоторые вещи отсутствуют. Здесь нет ширины и высоты компонента, а также (безумно сложных) стилей для создания клетчатого фона.

Причина состоит в том, что он делает компонент немного более настраиваемым извне. Если вы задумаетесь над этим, то вам понадобится, чтобы общий компонент пользовательского интерфейса, подобный это-

му, демонстрировал разные размеры в разных контекстах. Вы бы использовали стили, чтобы сделать это с любым другим элементом, и с этим компонентом не должно быть по-иному. Что касается клетчатого фона, я заранее предвижу, что нам понадобится использовать этот компонент в другом контексте, помимо настройки прозрачности. Установка стилей для фона снаружи позволяет нам легко и просто менять любой другой фон. В приведенном ниже листинге показана демонстрационная страница нашего ползунка со стилями для обозначения размера и фона, как уже обсуждалось.

**Листинг 6.20 Демостраница ползунка**

```
<html>
<head>
    <title>Slider Demo</title>

    <script                                    Модуль класса компонента
        type="module"
        src="slider.js">
    </script>
    <style>                                    Дополнительные стили, чтобы контролировать
        wcia-slider {                          размер и фон компонента
            height: 50px;
            width: 500px;
            background-image: linear-gradient(45deg, #ccc 25%,
             transparent 25%),linear-gradient(-45deg, #ccc 25%,
             transparent 25%),linear-gradient(45deg,
             transparent 75%, #ccc 75%),linear-gradient(-45deg,
             transparent 75%, #ccc 75%);
            background-size: 16px 16px;
            background-position: 0 0, 0 8px, 8px -8px, -8px 0px;
        }
    </style>
</head>
<body>                                         Компонент-ползунок на странице
    <wcia-slider                               с настройками по умолчанию
        backgroundcolor="#ff0000"
        value="50">
    </wcia-slider>
</body>
</html>
```

И последнее – это модуль slider.js. Да, теперь это модуль! Мы можем изменить `class Slider` на `export default class Slider`, чтобы его можно было импортировать. В приведенном ниже списке показан новый модуль. Он не включает детали, которые не были изменены.

**Листинг 6.21 Модуль Slider (slider.js)**

```
import Template from './template.js';          Импортируем файл template.js

export default class Slider extends HTMLElement {
    connectedCallback() {
```

```
        this.innerHTML = Template.render();
        this.dom = Template.mapDOM(this);

        document.addEventListener('mousemove', e => this.eventHandler(e));
        document.addEventListener('mouseup', e => this.eventHandler(e));
        this.addEventListener('mousedown', e => this.eventHandler(e));

        this.refreshSlider(this.getAttribute('value'));
        this.setColor(this.getAttribute('backgroundcolor'));
    }

    static get observedAttributes() { . . . unchanged . . . }
    attributeChangedCallback(name, oldVal, newValue) { . . . unchanged . . . }
    set value(val) { . . . unchanged . . . }
    get value() { . . . unchanged . . . }
    set backgroundcolor(val) { . . . unchanged . . . }
    get backgroundcolor() { . . . unchanged . . . }

    setColor(color) {
        if (this.dom) {
            this.dom.overlay.style.background = `linear-gradient(
            ➥to right, ${color} 0%, ${color}00 100%)`;
        }
    }

    refreshSlider(value) {
        if (this.dom) {
            this.dom.thumb.style.left = (value / 100 * this.offsetWidth -
            ➥this.dom.thumb.offsetWidth / 2) + 'px';
        }
    }

    updateX(x) {
        let hPos = x - this.dom.thumb.offsetWidth/2;
        . . . unchanged . . .
    }

    eventHandler(e) { . . . unchanged . . . }
}

if (!customElements.get('wcia-slider')) {
    customElements.define('wcia-slider', Slider);
}
```

Разгружаем HTML-код в шаблонный модуль

Кешируем элементы

Делаем класс импортируемым модулем

Использует кешированные элементы

Использует кешированные элементы

Использует кешированные элементы

Теперь наш компонент является совместно используемым фрагментом пользовательского интерфейса, который и правда можно использовать как часть любого другого проекта. В следующей главе мы сделаем последнюю вещь – заставим его использовать теневую модель DOM. Использование Shadow DOM не является непременным условием, но это потрясающая функция с точки зрения инкапсуляции компонентов. Я позволю вам быть судьей, когда вы прочитаете все об этом, и мы обновим ползунок в последний раз.

## *Резюме*

Из этой главы вы узнали:

- о новом способе записи строк с использованием символа `, что позволяет создавать шаблонные литералы. Это новая функция ES2015. Шаблонные литералы не только допускают использование многострочных строк без неудобного синтаксиса, но также позволяют вставлять переменные в шаблонную строку, что идеально подходит для вставки HTML-кода и стилей прямо в наш код JS без необходимости подстройки;
- как использовать кеширование элементов, а также разделение кода и разметки для лучшей читабельности и удобства сопровождения компонентов;
- о сгенерированных логикой шаблонах с использованием пользовательского JS, а также библиотеке lit-html от проекта Polymer для повторения разметки из данных, дополнительно используя lit-hml для добавления слушателей событий в ваш HTML-код;
- как в качестве примера создать проект веб-компонента с настройщиком визиток, ориентируясь на визуальный макет и стили, что позволяет нам исследовать рабочие процессы с более сложными HTML-кодом и CSS.

# 7

# *Шаблонирование контента с помощью HTML*

***Эта глава охватывает следующие темы:***

- концепции (ныне устаревшего) HTML-импорта;
- фрагменты документа;
- тег `<template>`;
- использование шаблонов для замены HTML-кода и стилей CSS в веб-компоненте;
- загрузка шаблонов из файла index.html или с помощью сетевого запроса;
- именованные и безымянные слоты.

Мы уже прошли длинный путь, изучая веб-компоненты! Помимо создания некоторых относительно простых приложений на основе веб-компонентов, мы довольно подробно рассмотрели стратегии использования HTML-кода и стилей CSS в наших веб-компонентах.

Конечно, до сих пор эти стратегии касались хранения разметки в строках JS. Несмотря на большое разделение проблем, которые мы получаем, сохраняя наш HTML-код и стили в импортируемых JS-модулях, как вид-

но из главы 6, несомненно, будут возникать ситуации, когда предпочтительнее сохранять HTML-код как HTML.

## 7.1 Покойся с миром, HTML-импорт

В действительности веб-компоненты начинались со стратегии «HTML-first». Под этим я подразумеваю, что если бы вы начали работать с веб-компонентами несколько лет назад, вы вряд ли бы рассчитывали на то, что вам придется импортировать JS-модули для управления своими компонентами – вместо этого вы бы полагали, что будете импортировать фактический HTML-код.

Импортированный HTML-код будет содержать тег `<script>`, в котором находится определение вашего класса веб-компонентов. Этот класс будет извлекать HTML-код и стили CSS из документа владельца, чтобы использовать их для содержимого пользовательского компонента. Поскольку это немного сложный вопрос, взгляните на рис. 7.1, и давайте разберем в качестве примера файл index.html из приведенного ниже листинга.

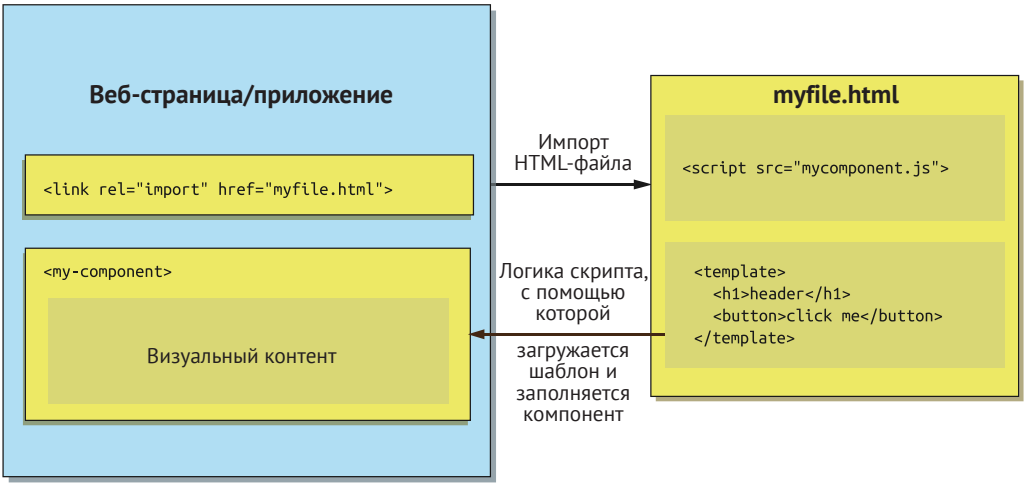



**Рис. 7.1 HTML-импорт позволяет загружать веб-компоненты посредством HTML-файла**

**Листинг 7.1 Использование HTML-импорта**

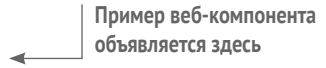
```
<html>
    <head>
        <title>HTML Import Demo</title>
        <script src="html-imports.min.js"></script>
        <link rel="import" href="samplecomponent.html"></link>

        <style>
            button {
                background-color: #c09853;
```

HTML-импорт, который загружает образец веб-компонента

```
            }
        </style>
    </head>

    <body>
        <sample-component></sample-component>
    </body>
</html>
```




Вы заметите, что тег `<sample-component>` используется так же, как мы обычно применяем веб-компоненты. Просто это еще один пользовательский элемент, который мы определили. Разница, конечно, состоит в том, как мы определяем этот веб-компонент.

В теге `<head>` мы ссылаемся на две вещи. Первая – это полифил:

```
<script src = "html-imports.min.js"> </ script>
```

Вторая – это фактический HTML-файл, который мы пытаемся импортировать:

```
<link rel = "import" href = "samplecomponent.html">
```

Конечно же, причина использования полифила заключается в том, что хотя Chrome был единственным браузером, который когда-либо поддерживал HTML-импорт, в последних версиях эта функция теперь считается устаревшей.

### 7.1.1 Полифилинг

Веб-компоненты в целом были изобретены под руководством компании Google. Рабочий проект спецификации был реализован в Chrome, чтобы вызвать интерес, прокладывая путь в надежде, что все браузеры последуют за ним. Пользовательские элементы оказались относительно бесспорной спецификацией. Другие разработчики браузеров работали с Google, чтобы внести свой вклад, и спецификация перешла из версии 0 в версию 1 в сотрудничестве с другими компаниями. Технология Shadow DOM, хотя и намного более сложная и, следовательно, более медленная для принятия, находилась на аналогичном пути и в конечном итоге была принята, как и пользовательские элементы.

С другой стороны, HTML-импорт, кажется, не пользуется популярностью. В частности, Firefox не хотел использовать нечто, подобное JS-модулям. Можно предположить, что когда-нибудь модули смогут импортировать больше, чем просто JS. Возможно, когда-нибудь мы сможем использовать модули для импорта файлов других типов, таких как HTML, и Chrome уже рассматривает это.

Несмотря на отсутствие поддержки, у веб-компонента на базе HTML-импорта есть несколько приличных идей. В случае с полифилом это, безусловно, выполнимый рабочий процесс, даже если большинство разработчиков веб-компонентов, вероятно, не будут использовать все это.

Чтобы еще больше сбить вас с толку, скажу, что официальный полифил от Google (https://github.com/webcomponents/html-imports) перешел с версии 0 (теперь она больше не поддерживается нативно ни в одном

браузере) на очень похожую реализацию версии 1. Он обеспечивает простую поддержку в любом браузере. Реализация версии 1 – это то, о чем пойдет речь здесь.

## 7.1.2 Что внутри

Теперь, когда мы знаем, с чем имеем дело, давайте заглянем в импортированный HTML-файл.

Конечно, в действительности это может быть любой HTML-код, но в целях создания веб-компонента мы делаем некоторые очень специфические вещи.

**Листинг 7.2 Содержимое веб-компонента на базе HTML-импорта**

```
<script src="samplecomponent.js"></script>

<template>
   <style>
       span {
           padding: 20px;
           background-color: yellow;
       }
   </style>

   <span>Hi from an HTML Import component</span>
</template>
```

Импорт класса веб-компонента

HTML-содержимое, предоставляемое тегом <template>

Обратите внимание на тег `<template>` из предыдущего листинга. Этот тег имеет последствия, выходящие далеко за рамки умирающего HTML-импорта, и его можно непосредственно применять к современной разработке веб-компонентов, поэтому я подробно расскажу о нем в следующем разделе этой главы. Пока просто отметим, что в теге `<template>` находится содержимое, которым мы бы хотели заполнить наш компонент.

Перед тегом `<template>` в первой строке листинга у нас идет ссылка в виде JS-скрипта на класс определения нашего веб-компонента:

```
<script src="samplecomponent.js"></script>
```

Это определение веб-компонента очень похоже на другие определения компонентов, которые мы рассматривали ранее, за некоторыми небольшими исключениями.

**Листинг 7.3 Заполнение разметки веб-компонента из шаблона HTML-импорта**

```
class SampleComponent extends HTMLElement {
   connectedCallback() {
       HTMLImports.whenReady( () => {
           const template =
               ownerDoc.querySelector('template');
           const clone =
               template.content.cloneNode(true);
```

Немодульный (без экспорта и неимпортируемый) класс веб-компонентов

Создает ссылку на шаблон

Клонирует содержимое

```
            this.appendChild(clone); ◄──── Добавляет содержимое в наш компонент
        });
    }
}

const ownerDoc = ◄──── Получает ссылку на документ-владелец
        HTMLImports.importForElement(document.currentScript);
if (!customElements.get('sample-component')) {
    customElements.define('sample-component', SampleComponent);
}
```

Как и другие определения компонентов, в листинге 7.3 мы определяем класс, расширяющий `HTMLElement`. Поскольку мы не импортируем его как `<script type = "module">`, он не начинается со слов `export default SampleComponent`.

Мы также по-прежнему используем тот же API пользовательского элемента для определения имени тега, как и во всех созданных ранее компонентах. Однако прямо над этой строкой есть нечто странное. Мы получаем «документ-владелец» этого скрипта. Помните, что мы не работаем с нашей страницей index.html, как обычно. Сейчас мы говорим об импорте другого HTML-документа целиком в нашу страницу index.html.

При участии еще одной (импортированной) HTML-страницы было бы неплохо, чтобы скрипт на этой импортированной странице знал, на какой из двух страниц он фактически запускается. В данном случае мы можем использовать метод `querySelector` для шаблона из импортированного HTML-кода, как мы это делаем в методе `connectedCallback` в листинге 7.3. Конечно, для этого скрипту необходимо знать, на какой странице он работает – документ-владелец.

Вот как выглядит общий поток HTML-импорта:

1. импортируем HTML-страницу;
2. заставляем импортированный HTML-/JS-код найти свой документ-владелец;
3. определяем веб-компонент на этой импортированной странице;
4. получаем и клонируем ссылки на шаблон на импортированной странице;
5. добавляем клонированный шаблон в веб-компонент.

Этот поток показан на рис. 7.2, и это тот же процесс, который мы используем в листинге 7.3.

Как только наш импортированный HTML будет полностью готов, как определено нашей функцией обратного вызова `HTMLImports.whenReady`, мы можем использовать для шаблона из этого документа-владельца метод `querySelector`, скопировать его, а затем добавить его в качестве дочернего элемента нашего компонента. В результате при предварительном просмотре в нашем браузере мы видим то, что изображено на рис. 7.3.

Итак, это было довольно легко, верно? Если бы не отсутствие у браузера HTML-импорта, это был бы довольно хороший рабочий процесс! Для тех, кто хочет держаться подальше от JS как способа написания HTML-кода и стилей CSS, это могло бы иметь потенциал.

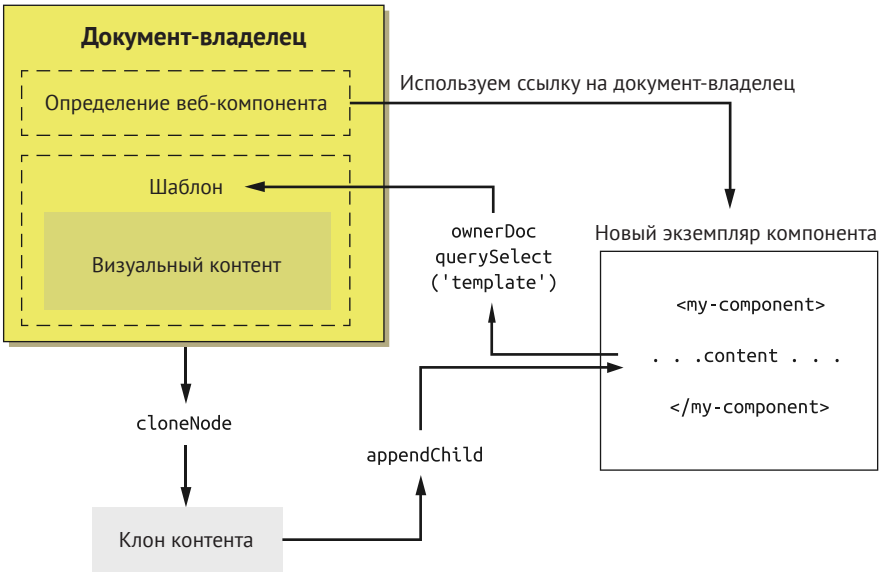



**Рис. 7.2 Типичный поток HTML-импорта. В документе-владельце содержится наше определение веб-компонента и шаблон требуемого HTML-кода и стилей CSS. Компонент отвечает за клонирование этого шаблона и вставку клона в качестве содержимого**

Hi from an HTML Import component

**Рис. 7.3 Вывод из нашего простого компонента**

Опять же, вы заметите, что я ничего не упомянул о работе с тегом `<template>`. Это связано с тем, что, несмотря на то что HTML-импорт не имеет каких-либо преимуществ, тег `<template>` доступен во всех современных браузерах, и некоторые считают его важной частью современного рабочего процесса веб-компонента. Таким образом, он заслуживает надлежащего объяснения, которое должно идти отдельно от HTML-импорта.

## 7.2 Тег <template>

Сам по себе тег `<template>` очень прост. Тем не менее его использование требует небольшого объяснения.

Давайте посмотрим на обычный, привычный HTML-код:

```
<p>
   This is content that's not in a template tag.
</p>
```

После удаления этого абзаца и его содержимого на HTML-странице просто отобразится содержимое. С другой стороны, мы могли бы использовать тег `<template>`:

```
<template>
   This is content that IS in a template tag.
</template>
```

Теперь содержимое нигде не отображается! Что произошло? Если изучить этот элемент в Chrome, как показано на рис. 7.4, элемент существует. Внутри него можно увидеть слова «document fragment». Затем вы можете развернуть фрагмент, чтобы увидеть фактический текст. Firefox показывает пустой тег `<template>`, но если вы щелкнете правой кнопкой мыши, чтобы просмотреть свойства DOM, то сможете увидеть свойство `content`, где находится фрагмент документа, содержащий текст.

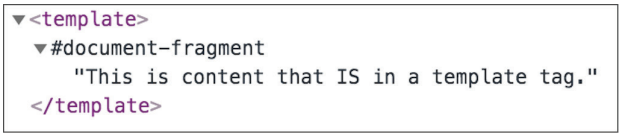



**Рис. 7.4 Проверка тега <template> в Chrome**

На самом деле это не дает ответа ни на какие вопросы, а просто меняет вопрос: что такое фрагмент документа?

## 7.2.1 Фрагменты документа

Чтобы узнать, что такое фрагмент документа, давайте просто создадим его с помощью JS, как показано в приведенном ниже листинге.

**Листинг 7.4 Использование фрагмента документа**

```
<html>
<head>
   <title>Document Fragment Demo</title>
</head>
<body>
<script>
   const fragment =                                  ← Создаем фрагмент документа
       document.createDocumentFragment();
   for (let c = 0; c < 5; c++) {
       const li = document.createElement('p');
       li.innerText = 'paragraph ' + c;              ← Добавляем дочерние узлы
       fragment.appendChild(li);                       во фрагмент (абзацы)
   }
   document.body.appendChild(fragment);              ← Добавляем фрагмент в тело страницы
</script>
</body>
</html>
```

Здесь мы сначала создаем фрагмент документа, а затем используем цикл `for`, чтобы добавить пять абзацев, содержащих некий текст. После добавления фрагмента к телу наше дерево DOM выглядит так:

```
<p>paragraph 0</p>
<p>paragraph 1</p>
<p>paragraph 2</p>
<p>paragraph 3</p>
<p>paragraph 4</p>
```

Довольно просто, как показано на рис. 7.5, но зачем заморачиваться по поводу фрагмента документа, когда можно просто использовать метод `createElement`?

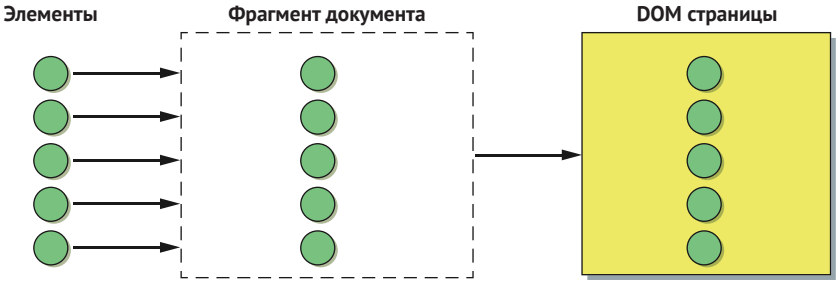



**Рис. 7.5 Добавляем элементы в DOM с помощью фрагмента документа**

Ну, во-первых, если бы мы хотели выполнить одну и ту же операцию, добавляя элементы к телу одним вызовом `appendChild` с помощью метода `createElement`, нам нужно было бы создать родительский элемент для хранения своих абзацев, как на рис. 7.6. Наш DOM выглядел бы так:

```
<div>
<p>paragraph 0</p>
<p>paragraph 1</p>
<p>paragraph 2</p>
<p>paragraph 3</p>
<p>paragraph 4</p>
</div>
```

Если это то, что нам нужно, отлично; но если нет, иной альтернативой будет добавление каждого тега `<p>` одного за другим в тело страницы, как показано на рис. 7.7. Все прекрасно, но каждый раз, когда вы это делаете, это приводит к пересчету всего DOM страницы. Чем меньше вы это делаете, тем лучше будет ваша производительность.

Еще один нюанс фрагментов документа заключается в том, что после добавления этих элементов к DOM из фрагмента они исчезают из самого фрагмента. В предыдущем примере, если бы мы записали в консоль нашу переменную `fragment` перед `document.body.appendChild(fragment);`, мы бы увидели `#documentFragment`, который может расширяться и показывать свои дочерние элементы. После добавления этот зарегистрированный

`#documentFragment` будет пустым. Имейте это в виду, потому что это будет важно позже, когда мы начнем работать с шаблонами.


**Рис. 7.6 Добавляем элементы в родительский элемент перед добавлением в DOM страницы**

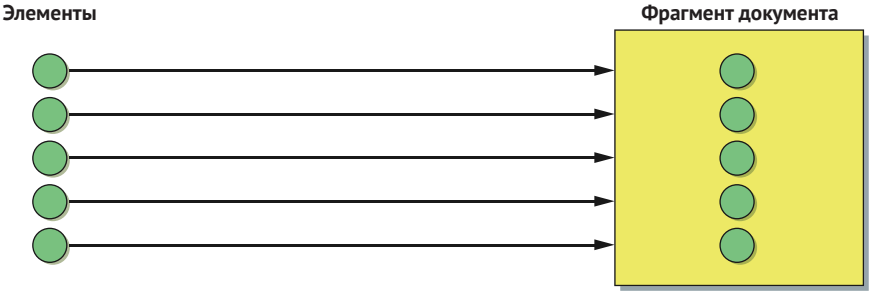



**Рис. 7.7 Добавление элементов по одному в DOM страницы каждый раз будет приводить к плачевному эффекту повторной визуализации DOM**

Фрагмент документа, похоже, не является широко известным; конечно, если вы никогда не использовали его раньше, это не будет шокирующим. Похоже, что он подходит для очень ограниченного варианта использования, но тег `<template>` сделал его чуть более популярным.

## 7.2.2 Использование содержимого шаблона

Учитывая все, что мы уже рассмотрели, вы могли бы догадаться, что тег `<template>` является своего рода областью хранения содержимого, который фактически не отображается на странице. Идея состоит в том, что ваша HTML-страница содержит различные теги `<template>`, в каждом из которых находится некий фрагмент HTML-кода или стилей CSS, который вы хотели бы использовать позже, скопировав его и добавив в основную модель DOM.

Давайте заполним HTML-файл несколькими чрезвычайно простыми шаблонами, как показано в приведенном ниже листинге.

**Листинг 7.5 Добавляем несколько шаблонов на страницу**

```
<html>
<body>
   <template id="button">                          Первый пример шаблона из трех
       <button>Click Me</button>
       <p>
           This is a template with a button
       </p>
   </template>

   <template id="textfield">
       <label>Enter</label>
       <input type="text">
       <p>
           This is a template with a text input
       </p>
   </template>

   <template id="list">
       <ul>
           <li>Item 1</li>
           <li>Item 2</li>
           <li>Item 3</li>
           <li>Item 4</li>
       </ul>
   </template>

<script>
   const template =                                Получаем ссылку на шаблон «button»
       document.getElementById('button');
   const clone =                                   Клонируем шаблон
       template.content.cloneNode(true);
   document.body.appendChild(clone);               Добавляем клонированное содержимое
</script>                                          на нашу страницу

</body>
</html>
```

Конечно, если мы откроем эту HTML-страницу в нашем браузере без этого блока скрипта, ничего не отобразится. Но наши шаблоны ждут и готовы к использованию. Однако с помощью этого блока мы можем получить один из шаблонов, и на нашей странице появится содержимое, как показано на рис. 7.8.

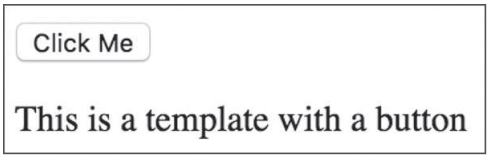



**Рис. 7.8 Один из наших шаблонов добавлен в браузер**

Получить тег <template>, который мы хотели бы использовать, очень просто! Он такой же, как и любой другой элемент. Мы можем исполь-

зовать методы `querySelector`, `querySelectorAll` или `getElementById`. В этом примере мы воспользуемся этим `document.getElementById('button');`. Попробуйте выбрать один из двух других шаблонов и добавьте его на свою страницу, если вы внимательно изучили код. Как только тег `<template>` будет сохранен в нашей переменной `template`, мы сможем получить фрагмент документа через свойство `content`. Чтобы использовать шаблон, мы должны сначала его клонировать, чтобы он не был пустым после добавления: `template.content.cloneNode(true)`. После этого мы можем добавить его на страницу с помощью `document.body.append-Child(clone)`. Передача значения `true` методу `cloneNode` просто означает, что нам нужно *глубокое клонирование*, т. е. нужно клонировать элемент, а также все его дочерние узлы.

Давайте сначала рассмотрим этот передовой метод клонирования, чтобы объяснить, как можно очистить шаблон. В частности, в этом ограниченном примере нам не нужно ничего клонировать. Мы можем просто добавить содержимое на нашу страницу с помощью `document.body.appendChild(template.content)`. Однако после добавления фрагмента документа к другому элементу ваш фрагмент будет пустым…

Это означает, что мы можем добавить тег `<template>` один и только один раз! Последующие попытки приведут к добавлению пустого содержимого. На рис. 7.9 показаны наши элементы в движении. Они переходят из фрагмента документа/шаблона в DOM страницы.

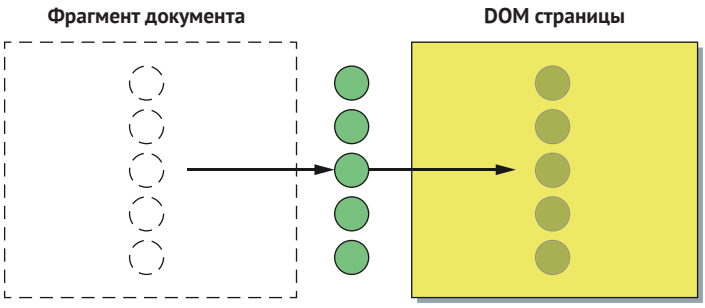



**Рис. 7.9 Присоединение к DOM страницы из фрагмента документа внутри шаблона означает, что эти элементы фактически выходят из шаблона/фрагмента**

Если мы клонируем наш тег `<template>`, вместо того чтобы добавить его напрямую, то можем использовать один и тот же тег снова и снова, как в приведенном ниже листинге.

**Листинг 7.6 Многократное клонирование**

```
const template = document.getElementById('button');
const clone = template.content.cloneNode(true);
document.body.appendChild(clone);
const clone2 = template.content.cloneNode(true);
document.body.appendChild(clone2);
```

Клонируем первый раз

Клонируем второй раз

Добавляем клон во второй раз

Добавляем клон на нашу страницу в первый раз

## 7.3 Выберите свой вариант шаблона

В последней главе мы занимались настройкой визитки на основе веб-компонента. Если вы помните, мы смогли поменять местами разные фоны и логотипы. Что, если бы мы могли выбирать между разными макетами карточек? Давайте создадим три отдельных шаблона и макеты, как показано на рис. 7.10.

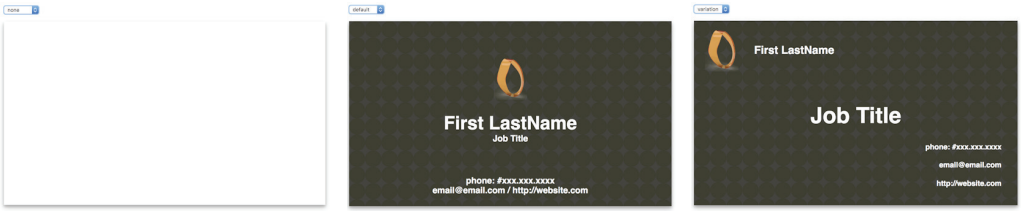



Рис. 7.10 Три разных макета визитки (начиная с пустого)

Для этого давайте упростим процесс и уберем настройки логотипа и фона, которые у нас были ранее, чтобы мы могли сосредоточиться на общем HTML-коде и стилях визитки.

**Листинг 7.7 Упрощенный пример визитки с использованием шаблонов для управления HTML-кодом и стилями CSS**

```
export default class BizCard extends HTMLElement {
   static get observedAttributes() { return ['layout']; }

   attributeChangedCallback(
       name, oldvalue, newvalue) {
       this.innerHTML = '';
       const template = document.getElementById(newvalue);
       const clone = template.content.cloneNode(true);
       this.appendChild(clone);
   }
}

if (!customElements.get('biz-card')) {
   customElements.define('biz-card', BizCard);
}
```

Упрощенная функция attributeChangedCallback с упором на содержимое

Говоря просто и коротко, это определение класса веб-компонента существует как bizcard.js в той же файловой структуре, которую мы использовали ранее в нашем примере с настройщиком визитных карточек. Напомним, что это на рис. 7.11.

Кроме того, чтобы не усложнять, мы будем просто стирать весь `innerHTML` компонента каждый раз, когда хотим загрузить новый макет визитки. Учитывая это, обратите внимание, что теги `<template>` макета карточки находятся в основном файле index.html вне этого компонента. Эти шаблоны выбираются по идентификатору, клонируются, а затем добавляются в наш веб-компонент. Теперь мы присоединяемся к пустому

узлу, учитывая, что мы только что очистили innerHTML компонента с помощью этого кода: `this.innerHTML = ''`. Это простота простой замены всего внутреннего HTML – именно поэтому наше новое меню в следующем разделе для выбора макета карты не будет находиться в компоненте. В противном случае это меню также было бы уничтожено!

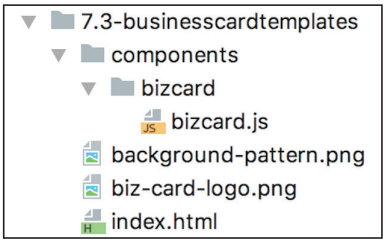



**Рис. 7.11 Приложение для создания визиток с использованием шаблона**

Вся эта логика содержится в функции компонента `attributeChangedCallback`. Это сделано для изменения имени макета, который мы хотели бы использовать, на основе атрибута компонента `layout`. Это, конечно, означает, что нам нужно объявить атрибут `layout` в геттере `observedAttributes`:

```
static get observedAttributes() { return ['layout']; }
```

Мы поместим в компонент пустой макет с именем «none», поскольку объявляем сам тег компонента в index.html. Но этот пустой макет, показанный на рис. 7.12, пока еще не представляет особого интереса:

```
<biz-card layout="none"></biz-card>
```

Опять же, учитывая то, как атрибуты работают в жизненном цикле веб-компонента, это начальное значение «none» инициирует функцию `attributeChangedCallback` и заполнит компонент этим конкретным макетом. Помимо этого, однако, чтобы фактически изменить макеты, мы можем реализовать раскрывающееся меню на странице с событием изменения, которое обновляет атрибут `layout` (см. листинг 7.8).



**Рис. 7.12 Начинаем с макета пустой/белой визитки**

**Листинг 7.8 Установка атрибута layout из меню за пределами нашего компонента**

```
<body>
   <p>
       <select onchange="updateLayout(event)">  <-- Меню для выбора макета визитки
           <option value="none">none</option>
           <option value="default-card">default</option>
           <option value="variation">variation</option>
       </select>
   </p>
   <biz-card layout="none"></biz-card>

   <script>
       function updateLayout(event) {
           document.querySelector('biz-card').setAttribute('layout',
           ➥event.target.value);  <-- Событие изменения для обновления
       }                               атрибута веб-компонента
   </script>
</body>
```

Конечно, помимо тега <head>, в котором содержится ссылка на наш модуль скрипта, существуют и реальные шаблоны для использования. В приведенном ниже листинге показана верхняя часть файла index.html и заполнители для трех разных шаблонов.

**Листинг 7.9 HTML-страница с шаблоном визитки**

```
<head>
    <title>Business Card</title>
    <script  <-- Импорт модуля веб-компонента
        type="module"
        src="components/bizcard/bizcard.js">
    </script>
</head>

<template id="default-card"> . . . </template>  <-- Три наших шаблона (заполнители)
<template id="variation"> . . . </template>
<template id="none"> . . . </template>
```

Чтобы сжать здесь код на странице, в особенности длинный список стилей CSS в шаблонах, я включил только теги <template> без внутреннего содержимого. Обратитесь к моему репозиторию на Github, если хотите увидеть все это целиком (https://github.com/bengfarrell/webcomponentsinaction/blob/master/chapter7/7.3-businesscardtemplates/index.html). Как показано на рис. 7.13, наш компонент обращается к документу, выбирает каждый шаблон по этому идентификатору и, как мы уже видели, заполняет компонент.

Сейчас, хотя все это прекрасно подходит, для того чтобы быть помещенным в наш файл index.html, это выглядит немного неряшливым и длинным. Я еще не определился относительно того, действительно ли он не организован, – длинный список из тегов <template> легко исполь-

зовать, поскольку он не мешает фактической отображаемой структуре DOM страницы. С другой стороны, когда есть несколько пользовательских компонентов, не ясно, какой тег `<template>` какому компоненту принадлежит. В связи с этим такой код кажется сложным для управления в зависимости от вашего конкретного случая использования. Кроме того, при наличии множества компонентов, используемых в проекте, может быть слишком много шаблонов, чтобы ваша HTML-страница оставалась управляемой.

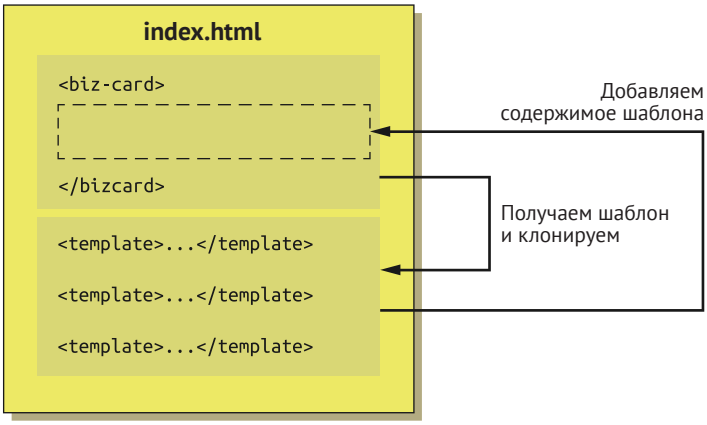



**Рис. 7.13 Компонент, обращающийся к HTML-странице и получающий шаблон по идентификатору**

Учитывая эти оговорки, мне бы хотелось, чтобы код был чище. Как вы, наверное, помните, HTML-импорт содержал все в чистоте! Можно ли придумать другой способ динамической загрузки шаблонов без него?

## 7.4 Динамически загружаемые шаблоны

В этом примере давайте подумаем над двумя моментами. Во-первых, я бы хотел оставить наши различные теги `<template>` внутри нашего компонента в качестве дочерних узлов. Сделав это, вы поймете, что шаблоны на самом деле принадлежат данному веб-компоненту. Во-вторых, я хотел бы загружать наши шаблоны откуда-то еще, вместо того чтобы загромождать компонент.

Вы, наверное, подумали, что это можно было бы сделать с помощью шаблонных литералов и модулей, как мы это делали в предыдущих главах, и это, безусловно, так! Да, могли бы. Но я буду избегать подхода «HTML-в-JS» только потому, что мы это уже делали. Кроме того, может быть интересно загружать шаблоны в качестве еще одного удаленного ресурса, который можно извлечь с сервера.

Интересный момент, касающийся хранения тегов `<template>` внутри нашего компонента, заключается в том, что мы должны поддерживать

относительно постоянную разметку внутри компонента, а также очищать большие куски HTML-кода и стилей CSS при каждом обновлении макета.

Это означает, что если мы сразу же настроим `this.innerHTML`, этого будет недостаточно. Если мы заменим весь наш HTML-код, то, по сути, выбросим наши загруженные шаблоны. В приведенном ниже листинге метод нашего компонента `connectedCallback` должен отражать это.

**Листинг 7.10 Загрузка шаблонов с помощью сетевого запроса**

```
connectedCallback() {
   this.cardElement =                          // Создаем контейнер макета визитки
        document.createElement('div');
   this.templates =                            // Создаем контейнер шаблона
        document.createElement('div');
   this.appendChild(this.cardElement);
   this.appendChild(this.templates);
   const request = new XMLHttpRequest();
   request.open( 'GET', 'templates.html', true );   // Сетевой запрос для извлечения шаблонов
   request.addEventListener( 'load', (event) => {
       this.templates.innerHTML =                   // Заполняем контейнер шаблонов
               event.target.response;               // загруженными шаблонами
       this.populateCard();
   });
   request.send();
}
```

Мы сразу же создаем и добавляем два элемента `<div>`, `this.cardElement` и `this.templates`. Они будут действовать как контейнеры для визитной карточки и наших загруженных шаблонов соответственно.

Затем мы выполняем сетевой запрос на загрузку файла templates.html, содержащий все теги `<template>`, которые раньше находились в нашем файле index.html. После загрузки мы можем просто установить значение для свойства `innerHTML` для `this.templates` `<div>`, как показано на рис. 7.14.

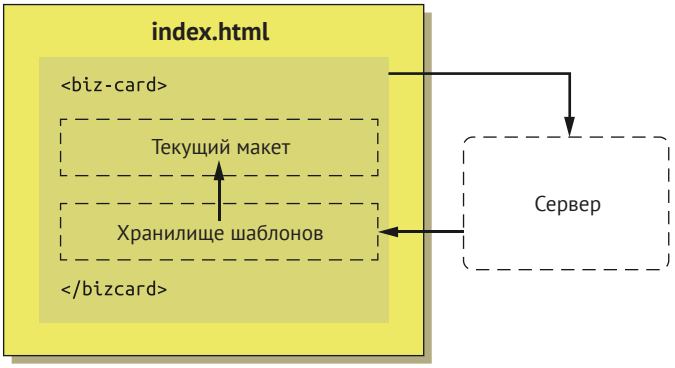



**Рис. 7.14 Обращаемся к серверу, чтобы извлечь шаблоны для нашего веб-компонента**

И эта функция обратного вызова, и наш метод `attributeChangedCallback` вызывают функцию `populateCard();`, чтобы загрузить текущий макет, как указано в нашем атрибуте `layout`. Но в этом случае неплохо было бы проверить: возможно, `this.templates` уже существует, учитывая, что `attributeChangedCallback` может сработать до метода `connectedCallback`, как показано в приведенном ниже листинге.

**Листинг 7.11 Вызов метода для заполнения макета нашей карты**

```
static get observedAttributes() { return ['layout']; }

attributeChangedCallback(name, oldvalue, newvalue) {
   if (this.templates) {
      this.populateCard();
   }
}
```

Вызываем метод для заполнения контейнера нашей визитки в components/bizcards/bizcard.js

Как бы вы это не называли – сетевой запрос для загрузки файла template.html или результат изменения атрибута, – у функции `populateCard();` из приведенного ниже листинга есть довольно простой метод добавления в тот же класс, чтобы поменять местами наш новый макет визитки.

**Листинг 7.12 Содержимое функции `populateCard`**

```
populateCard() {
   const template = this.templates.querySelector(
   ➥'template.' + this.getAttribute('layout'));
   if (template) {
      const clone =
            template.content.cloneNode(true);
      this.cardElement.innerHTML = '';
      this.cardElement.appendChild(clone);
   }
}
```

Метод populateCard в components/bizcards/bizcard.js

Получаем ссылку на шаблон

Клонирует шаблон

Очищаем текущий шаблон

Добавляем клон и заполняем HTML/CSS для текущего макета

Первое, что мы делаем, – это берем шаблон из компонента. Напомню, что я переместил шаблоны из файла index.html в отдельный файл templates.html и сделал немного по-другому. Вместо того чтобы использовать идентификаторы для имен шаблонов, я теперь использую классы. Если раньше было `<template id = "default-card">`, то теперь вы видите это: `<template class="default-card">`.

Как правило, когда вы будете искать в интернете информацию о том, как использовать тег `<template>`, то увидите, что люди применяют атрибут `id` для идентификации и получения своего тега `<template>` из DOM. В этом упражнении, поскольку мы хотим сохранить наши шаблоны в качестве дочернего узла компонента, не имеет особого смысла использовать `id`. Помните, что каждый отдельный идентификатор можно использовать только один раз на всей HTML-странице. Когда шаблоны находятся на странице вне структуры DOM, идентификаторы имеют смысл, потому что мы рассматриваем пул шаблонов на всей странице, каждый из которых вызывается по своему уникальному идентификатору.

Теперь, вместо того чтобы выполнять запрос ко всей странице на предмет уникального идентификатора, мы запрашиваем не только дочерние элементы нашего компонента, но, в частности, дочерние элементы контейнера `this.templates`. Если они будут найдены (а могут быть и не найдены из-за того, что файл template.html еще не загружен), содержимое контейнера нашей визитки будет очищено с помощью `this.cardElement.innerHTML = ''`, шаблон будет клонирован, а затем новый дочерний элемент будет добавлен к `this.cardElement`.

С точки зрения шаблонов, поскольку содержимое в них такое же, как и раньше, мы лишь заменили идентификаторы на классы:

```
<template class="default-card">
. . .
</template>
```

Конечно, после того как мы убрали шаблоны, наш файл index.html становится намного проще.

**Листинг 7.13 После удаления шаблонов наш файл index.html снова становится управляемым**

```
<html>
   <head>
       <title>Business Card</title>
       <script type="module" src=
       ➥"components/bizcard/bizcard-template-loading.js"></script>
   </head>

   <body>
       <p>
           <select onchange="updateLayout(event)">
               <option value="none">none</option>
               <option value="default-card">default</option>
               <option value="variation">variation</option>
           </select>
       </p>
       <biz-card layout="none"></biz-card>

       <script>
           function updateLayout(event) {
               document.querySelector('biz-card').setAttribute
               ➥('layout', event.target.value);
           }
       </script>
    </body>
</html>
```

Гораздо более короткое содержимое тега <body> с шаблонами, которые загружаются компонентом

При всем при этом наш пример выглядит точно так же, как и раньше, просто он намного чище. Кроме того, мы могли бы пойти дальше и указать другой HTML-файл для загрузки. Мы даже могли бы использовать атрибут для компонента, чтобы указывать на конкретные файлы HTML, заполненные шаблонами для конкретного случая использования:

```
request.open( 'GET', this.getAttribute('templatefile'), true );
```

Готово? Не совсем. Вы, наверное, обратили внимание на один момент. У наших новых визиток в этой главе (когда мы использовали шаблоны) отсутствует пользовательская информация, такая как имя, фамилия, название должности и т. д.

Возможно, одно из решений – более тщательное обеспечение того, чтобы каждый элемент, содержимое для которого нам бы хотелось заменить, был помечен соответствующим классом. Затем мы можем запросить наш макет для элемента, помеченного классом, и заменить свойство `innerHTML`.

Например, если мы гарантируем, что каждый элемент, который содержит заполнитель `firstname`, имеет класс `firstname`, то можно сделать следующее:

```
this.cardElement.querySelector('firstname').innerHTML = someObject.firstname;
```

Однако у этого метода есть некоторые сложности. Давайте обратимся к нашему шаблону по умолчанию, где адреса электронной почты и веб-сайта перечислены в теге заголовка `<h3>`:

```
<div class="bottom-text">
   <h3>phone: #xxx.xxx.xxxx</h3>
   <h3>email@email.com / http://website.com</h3>
</div>
```

Как заменить содержимое этого комбинированного поля, особенно если оно может быть разделено на отдельные элементы в других шаблонах? Кроме того, в этом шаблоне косая черта разделяет значения электронной почты и веб-сайта. Устанавливая значение для свойства `innerHTML` тега `<h3>`, вам нужно знать, что эта черта является проектным решением данного шаблона и вы обязательно должны включить и ее тоже!

Дело усложняется. В качестве одного из решений можно было бы вставить теги `<span>`, чтобы отметить каждое значение, которое вы хотите заменить, и использовать их для выбора запроса:

```
<div class="bottom-text">
   <h3 class="phone">phone: #xxx.xxx.xxxx</h3>
   <h3><span class="email">email@email.com</span> / <span
     class="website">http://website.com</span></h3>
</div>
```

Это хорошее решение, но мы добавляем немного больше сложности в наш HTML-код, когда в действительности нам это не нужно. К счастью, существует более свежее решение, предназначенное только для этой проблемы. Я говорю о теге `<slot>`!

## 7.5 Вход в теневую модель DOM с помощью тега <slot>

В самом деле, тег `<slot>` является идеальным решением для нашей дилеммы, связанной с полями, но перед тем, как заняться этим детально,

нужно кое о чем сказать. Тег <slot> работает только в сочетании с теневой моделью DOM. Это обширная тема, и я думаю, что лучше начать углубленное изучение Shadow DOM в следующей главе. А пока давайте рассмотрим тег <slot>! В результате мы получим визитную карточку с полями, заполненными пользовательскими значениями вместо значений заполнителей, как показано на рис. 7.15.

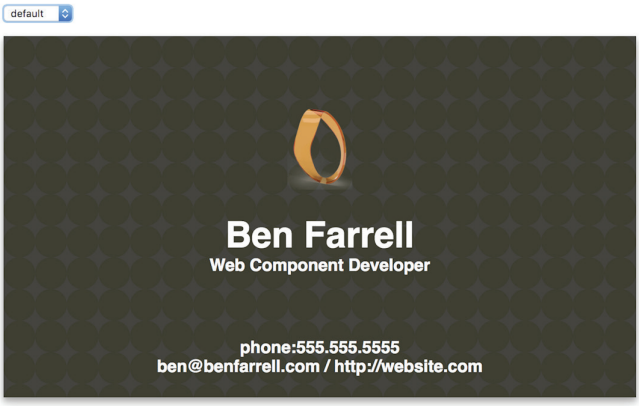



**Рис. 7.15 Визитная карточка, где используется шаблонный макет и слоты для вставки пользовательских значений**

Тег <slot> немного похож на тег <template> в том смысле, что он фактически не отображается в макете DOM. В отличие от тега <template>, мы ничего не копируем из него. Вместо этого содержимое автоматически помещается внутрь. По сути, слоты являются целями для замены контента. Давайте возьмем один из наших макетов с тегами <template> и создадим несколько слотов для содержимого, которое мы можем поменять, как показано в листинге 7.14.

**Листинг 7.14 Помещаем теги <slot> в наш шаблон для замены содержимого**

```
<div class="biz-card">
   <div class="logo"></div>
   <div class="top-text">
       <h1><slot name="firstname">First</slot>
           <slot name="lastname">LastName</slot></h1>
       <h3>
           <slot name="title">Job Title</slot>
       </h3>
   </div>

   <div class="bottom-text">
       <h3>phone:
           <slot name="phone">#xxx.xxx.xxxx</slot>
       </h3>
       <h3><slot name="email">email@email.com</slot> /
```

Первые теги <slot>, содержащие имя и фамилию

Третий тег <slot>, содержащий название должности

Четвертый тег <slot> с номером телефона

```
        <slot name="website">http://website.com</slot></h3>
    </div>
</div>
```

Последние теги <slot>, содержащие адреса электронной почты и сайта

Здесь я обернул каждое отдельное значение заполнителя в тег `<slot>`. У каждого такого тега также есть атрибут `name`, чтобы определить, как ссылаться на слот. Итак, как же заменить содержимое? Как я уже сказал, для того чтобы это действительно сработало, нужно использовать в нашем компоненте теневую модель DOM. К счастью, нужно сделать лишь несколько изменений, как показано в приведенном ниже листинге, чтобы использовать ее.

**Листинг 7.15 Изменение метода `createdCallback` для использования теневой модели DOM**

```
connectedCallback() {
    this.root = this.attachShadow({mode: 'open'});
    this.cardElement = document.createElement('div');
    this.templates = document.createElement('div');
    this.root.appendChild(this.cardElement);
    this.root.appendChild(this.templates);
```

Создаем теневую модель DOM для использования тегов <slot>

Добавляем элементы в теневую модель DOM вместо компонента

В нашем методе `connectCallback` мы присоединяем *корень теневого дерева*. Рассматривайте его как отдельное защищенное дерево DOM, доступное только для внутренней работы нашего компонента. Затем мы можем сохранить этот корень как `this.root` (вы можете использовать любое другое имя переменной) и добавить к нему любые дочерние элементы. Несмотря на то что `this.cardElement` и `this.templates` находятся внутри корня теневого дерева, они уже добавлены в новую теневую модель DOM, поэтому их использование абсолютно не меняется. Мы можем использовать эти ссылки на элементы так же, как и всегда, и установить значение для их свойства `innerHTML` или добавить дополнительные дочерние элементы.

Теперь, чтобы фактически заполнить наши слоты-заполнители, показанные на рис. 7.16, мы можем поместить соответствующие именованные значения прямо в наш компонент `<biz-card>`, как показано в приведенном ниже листинге.

**Листинг 7.16 Заполнение слотов значениями внутри тега компонента**

```
<biz-card layout="none">
    <span slot="firstname">Ben</span>
    <span slot="lastname">Farrell</span>
    <span slot="phone">555.555.5555</span>
    <span slot="email">ben@benfarrell.com</span>
</biz-card>
```

Один из четырех слотов, которые мы заполняем значением firstname

Взглянув на результат, приведенный на рис. 7.15, видно, что мы забыли создать значение для «website». Обратите внимание, что, вместо того чтобы выдать какую-нибудь ошибку, в теге `<slot>` для «website» остается

содержимое по умолчанию. Также имейте в виду, что тег `<span>` целиком вставляется в тег `<slot>`. Это легко может быть `<button slot="firstname">Ben</button>` или даже еще один тег `<slot>`, где будет отображаться только его содержимое: `<slot slot="firstname">Ben</slot>`.

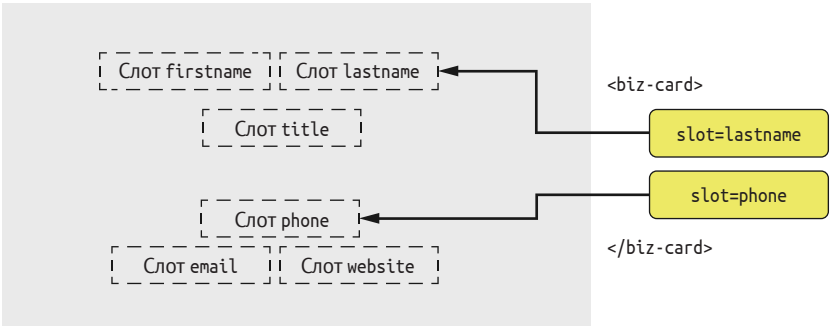



**Рис. 7.16 Использование именованных слотов в качестве заполнителей в веб-компоненте визитки**

Я закончу этот пример с визиткой парой нерешенных проблем. Первая – это, конечно, добавление тега `<slot>` для адреса сайта, чтобы заполнить последний заполнитель. Что еще более важно, мы потеряли немало функциональных возможностей начиная с предыдущей главы. Наш фон и логотип больше не являются настраиваемыми. Если вы готовы принять вызов, возможно, вы захотите попытаться включить их снова!

## 7.5.1 Тег <slot> без имени

Теги `<slot>` могут быть даже более обобщенными. В нашем примере мы используем *именованные теги*, но им вообще не нужно давать имена – вы просто потеряете возможность указывать и использовать несколько тегов `<slot>` в одном и том же компоненте, как в приведенном ниже листинге с результатами, выводящимися в браузере, которые показаны на рис. 7.17.

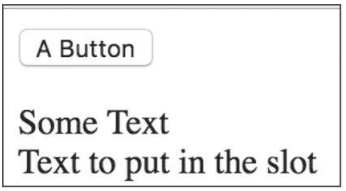



**Рис. 7.17 Результаты, появляющиеся в браузере (см. листинг 7.17)**

**Листинг 7.17 Использование тегов <slot> без имени**

```
<script>
   class SlotsDemo extends HTMLElement {
       connectedCallback() {
           this.root = this.attachShadow({mode: 'open'});
```

```
            this.root.innerHTML = `<div>
                   <button>A Button</button>
                   <p>
                        Some Text
                        <br />
                        <slot>placeholder text</slot>  <-- Создаем тег <slot> без имени
                    </p>
                </div>`;
        }
    }

    if (!customElements.get('slots-demo')) {
        customElements.define('slots-demo', SlotsDemo);
    }
</script>
<body>
    <slots-demo>  <-- Заполняем тег <slot>
         Text to put in the slot
    </slots-demo>
</body>
```

Конечно, именованные теги `<slot>` лучше подходят для нашей визитки, потому что мы можем вставлять несколько значений во все нужные места. Однако мне нравится простота использования тега `<slot>` без имени. Добавление простого текста в качестве содержимого вашего тега – что может быть проще!

Немного странно то, что мы создали защищенную модель DOM, которая недоступна за пределами нашего компонента, но мы делаем это для создания заполнителей со значениями, которые заменяются извне. Это выглядит несколько нелогично, но не лишено смысла, если учесть, что корень теневого дерева слегка меняет использование нашего компонента.

В этой книге я довольно часто использовал API JS для настройки свойства `innerHTML` нашего компонента изнутри. Не менее правильным является установка `innerHTML` напрямую при использовании вашего компонента на странице:

```
<my-component>This text is the innerHTML</my-component>
```

Однако при применении теневой модели DOM свойство `innerHTML` больше не отображается, поскольку этот тип внешнего содержимого не будет проникать в нее. Отображается только `innerHTML` внутри корня теневого дерева. Это создает прекрасную возможность для использования этого содержимого внутри вашего тега иным образом. Конечно, в этом ином способе используются теги `<slot>`. Этим тегам разрешено пробиваться сквозь теневую модель DOM очень специфическими способами, описанными здесь. Если бы мы не использовали эту модель, то было бы немного двусмысленно, если бы содержимое должно было отображаться как фактическое содержимое вашего компонента или для того, чтобы заполнить тег `<slot>`.

Как мы увидим в последующих нескольких главах, теневая модель DOM – это действительно мощная вещь. В сочетании с тегами `<slot>` она отлично работает, но, вероятно, это не та причина, по которой вы захотите использовать эту модель. В последующих главах мы подробно рассмотрим, что такое теневая модель DOM и что она может сделать для вашего рабочего процесса разработки компонентов, потому что пока мы лишь поверхностно коснулись этой темы.

## Резюме

Планируете вы использовать шаблоны или нет, здорово, когда у вас есть еще один инструмент в своем наборе. Лично у меня был замечательный опыт использования шаблонных литералов для хранения HTML-кода и стилей CSS в JS, о чем мы говорили в предыдущих главах, но не все ситуации одинаковы. Это приложение является отличным примером того, где шаблоны могут действительно оказаться полезными, особенно когда нам нужно создать множество взаимозаменяемых шаблонов и не навязывать frontend-разработчикам из нашей команды, которые, возможно, не знают JS настолько хорошо, чтобы помочь нам.

Из этой главы вы узнали:

- как работать с HTML-импортом, несмотря на то что он уже устарел, а также с тем, как он работает, что может иметь отношение к разработке современных веб-компонентов;
- что такое тег `<template>`, что вы будете неоднократно использовать эти теги при первом клонировании их содержимого и что фрагменты документа являются основным драйвером для них;
- как использовать шаблоны на практическом примере, в котором мы полностью поменяли HTML-код и стили CSS, чтобы познакомиться с новыми макетами и стилями в одном и том же компоненте, загруженном со страницы index.html или удаленно через сервер;
- как заменить определенное содержимое или содержимое из нескольких заполнителей в одном и том же компоненте с именованным тегом `<slot>` или одним заполнителем в теге `<slot>`, но без использования имен.

# 8

# Теневая модель DOM

***Эта глава охватывает следующие темы:***

- инкапсуляция компонентов и классов;
- как Shadow DOM защищает DOM вашего компонента;
- открытая и закрытая теневая модель DOM;
- терминология Shadow DOM: корень теневого дерева, теневые границы и элемент, размещающий в себе теневое дерево;
- полифилинг и Shady DOM.

В предыдущей главе мы кратко коснулись теневой модели DOM, чтобы познакомиться с концепцией тега `<slot>`. Если вы помните, тег `<slot>` используется для получения шаблонного содержимого и добавления значений заполнителей, которые конечный пользователь вашего веб-компонента может заменить. Мы отметили области, которые могут принимать новое HTML-содержимое в качестве тегов `<slot>`.

Хотя тег `<template>` является автономной концепцией и доступен во всех современных браузерах, с тегом `<slot>` дело обстоит иначе. На самом деле тег зависит от теневой модели DOM. На данный момент мы рассмотрели все основные функции веб-компонентов, кроме теневой модели DOM.

Есть причина, по которой я рассказываю о ней в последнюю очередь. Дело в том, что я хочу показать, что она не является абсолютно необхо-

димой для веб-компонентов, какой бы удивительной она ни была. В предыдущих главах мы рассмотрели пользовательские элементы, шаблоны и HTML-импорт, а также методы, не основанные на веб-компонентах, такие как модули ES2015 и шаблонные литералы. Все эти концепции сейчас либо доступны для всех современных браузеров, либо здесь можно с легкостью использовать полизаполнение.

Теневая модель DOM немного сложнее. Что касается поддержки со стороны браузеров, мы только сейчас видим практически универсальный охват в современных браузерах, когда Microsoft выпускает свою последнюю версию Edge с поддержкой Chrome в качестве предварительной версии для разработчиков. Это происходит после выпуска Firefox в октябре 2018 года с полной поддержкой веб-компонентов.

Даже несмотря на неповсеместное использование этой концепции до недавнего времени, в последние годы вокруг веб-компонентов было немало шумихи с активной рекламой теневой модели DOM. Я согласен, что это революционная функция браузера для рабочих процессов веб-разработки, но веб-компоненты – это гораздо больше, чем одна эта функция. Несмотря на это, сообщество разработчиков веб-компонентов отчасти было разочаровано медлительностью внедрения Shadow DOM в сочетании с тем, насколько проблематичным является полифилинг.

Вот почему я не рассматривал теневую модель DOM до настоящего момента. Для меня это дополнительная функция в моей повседневной работе, которую я использую, только когда не беспокоюсь о поддержке со стороны браузера, и мне хотелось отразить это здесь. За последние несколько месяцев эта озабоченность значительно уменьшилась, учитывая, что мы ждем только одного браузера (Edge); тем временем команда Polymer усердно работает над LitElement и lit-html, которые обещают интеграцию и поддержку полифилов даже в IE11.

Вы можете быть разработчиком веб-компонентов и выбирать, какие функции использовать, включая теневую модель DOM. Тем не менее, после того как она будет поставляться со всеми современными браузерами, я планирую использовать ее постоянно – и этот день быстро приближается и, скорее всего, уже наступит к тому времени, когда эта книга будет опубликована!

## 8.1 Инкапсуляция

Что касается шумихи в отношении теневой модели DOM, я видел утверждения относительно того, что она устраняет хрупкость при создании веб-приложений и, наконец, ускоряет веб-разработку на других платформах. Соответствует она этим требованиям?

Я позволю вам решать, так это или нет, потому что, как и везде, ответ зависит от вашего проекта и потребностей. Однако оба этих заявления сделаны с уклоном на один центральный элемент: *инкапсуляцию*.

Когда речь идет об инкапсуляции, обычно имеются в виду две вещи. Первая – это возможность обернуть объект так, чтобы он выглядел прос-

то снаружи, но внутри он может быть сложным и управлять своими внутренним устройством и поведением.

Пока все, что мы узнали о веб-компонентах, поддерживает все вышесказанное, являясь отличным примером данного определения инкапсуляции. Веб-компоненты предлагают:

- простой способ включить себя в HTML-страницу (пользовательские элементы);
- множество способов управления собственными зависимостями (модули, шаблоны ES2015 и даже устаревшая теперь функция HTML-импорта, где можно легко использовать полизаполнение);
- пользовательский API для управления ими с помощью атрибутов или методов на основе классов, геттеров и сеттеров.

Все это замечательно, но часто, говоря об инкапсуляции, мы придаем ей более широкое определение. Инкапсуляция – это то, что мы только что обсудили; но она также может означать, что ваш инкапсулированный объект защищен от конечных пользователей, взаимодействующих с ним, даже непреднамеренно, таким образом, каким вы и не предполагали. Это показано на рис. 8.1.

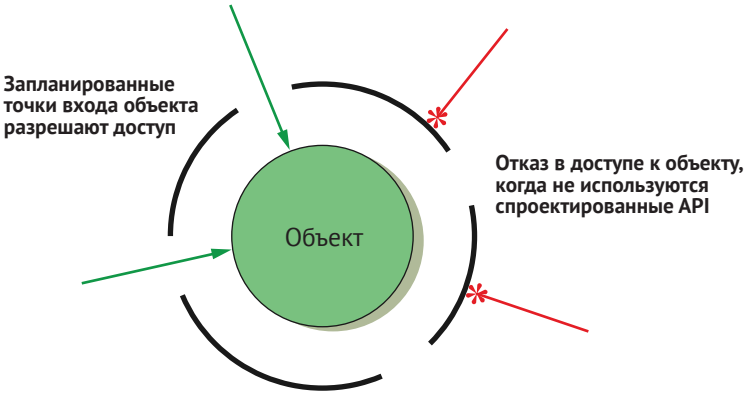



**Рис. 8.1 Инкапсуляция означает сокрытие внутреннего устройства объекта, но часто включает в себя выбор того, как и где обеспечить доступ извне**

## 8.1.1 Защита API вашего компонента

В приложении я упоминаю несколько способов сделать ваши переменные закрытыми (private) в вашем классе веб-компонентов. Важно то, что вы, будучи разработчиком, задумывались над тем, как используется ваш класс, и приложили некоторые усилия, чтобы ограничить внешнее использование ваших свойств и методов только тем, как вы собираетесь их использовать.

Одно важное различие заключается в фактическом ограничении свойств и методов и ограничении их только условным соглашением. Хорошим примером ограничения по соглашению является использование нижнего подчеркивания для свойств и переменных в вашем классе.

Например, кто-то из вашей команды может передать вам компонент, у которого есть метод для добавления нового элемента списка в его пользовательский интерфейс:

```
addItemToUI(item) {
        this.appendChild(`<li>${item.name}</li>`);
}
```

Когда вы впервые используете этот компонент, вы можете подумать: «Эй, я просто воспользуюсь данной функцией, чтобы добавить новый элемент в свой список!» Но вы не знаете, что у класса компонента есть внутренний массив данных элементов. Будучи пользователем этого компонента, вы должны использовать метод `add()`, который добавляет элемент в модель данных, а затем вызывает функцию `addItemToUI`, чтобы добавить элемент `<li>`:

```
add(item) {
    this.items.push(item);
    this.addItemToUI(item);
}
```

Когда размер компонента изменяется или он сворачивается/скрывается и отображается снова, эти элементы списка уничтожаются, а затем создаются снова с использованием внутренней модели данных. Как человек, применяющий этот компонент впервые, вы не знали, что такое произойдет! Когда вы использовали метод `addItemToUI` вместо `add`, компонент был воссоздан заново, а добавленный элемент теперь отсутствует.

В этом примере метод `addItemToUI` не должен использоваться пользователем компонента; его нужно применять только внутри компонента. Если бы первоначальный разработчик компонента потратил время и усилия на то, чтобы сделать метод закрытым, его вообще было бы невозможно вызвать.

С другой стороны, разработчик компонента может сделать метод закрытым по соглашению. Самый популярный способ сделать это – использовать нижнее подчеркивание, и в этом случае метод назывался бы `_addItemToUI`. Вы по-прежнему можете вызывать метод как пользователь компонента, но при виде нижнего подчеркивания вы знаете, что в действительности не должны этого делать.

И это еще не все. Инкапсуляция веб-компонентов – это понятие защиты вашего компонента на самом деле, или просто сделайте это по соглашению, которое вступает в игру за пределами определения класса вашего компонента.

## 8.1.2 Защита DOM вашего компонента

Защита пользовательских методов и свойств класса веб-компонента, вероятно, меньше всего вас беспокоит! Что еще в вашем компоненте должно быть защищено? Давайте рассмотрим компонент из приведенного ниже листинга.

**Листинг 8.1 Пример простейшего компонента**

```
<head>
   <script>
       class SampleComponent extends HTMLElement {
           connectedCallback() {
               this.innerHTML =
               ➥`<div class="inside-component">My Component</div>`
           }
       }

       if (!customElements.get('sample-component')) {
           customElements.define('sample-component', SampleComponent);
       }

   </script>
</head>
<body>
   <sample-component></sample-component>
</body>
```

Простейший веб-компонент, размещенный на веб-странице

Как вы, возможно, заметили, в этом компоненте нет ничего особенного. Он просто отображает тег  с текстом «My component» внутри, как показано на рис. 8.2.


**Рис. 8.2 Простой веб-компонент, отображающий короткую строку в браузере**

Насколько защищен тег  снаружи с точки зрения инкапсуляции? Оказывается, совсем не защищен. Мы можем добавить тег <script> сразу же после нашего компонента:

```
<script>
   document.querySelector('.inside-component').innerHTML +=
   ' has been hijacked';
</script>
```

На рис. 8.3 из вывода нашего браузера явствует, что свойство innerHTML нашего компонента в действительности было задано извне. Если проанализировать то, что произошло, становится ясно, что кто-то посторонний успешно воспользовался методом querySelector внутри нашего компонента и установил HTML-содержимое элемента .

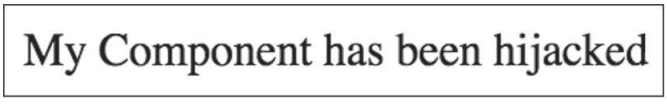



**Рис. 8.3 Установка свойства innerHTML DOM нашего компонента извне**

Прежде чем мы поговорим о том, что можно сделать для решения этой проблемы, нужно разбить ее на две части. В первой части я притворяюсь,

что у меня есть злой умысел при использовании этого компонента таким образом, чтобы его нельзя было использовать, преднамеренно нарушая его функциональность и структуру извне. В этом примере я точно знаю, что существует тег `<div>` с классом `inside-component`. Я знаю, что у него есть какой-то текст, который он отображает, и я специально изменяю его.

Вторая часть носит менее злонамеренный характер. Что, если бы мы сделали нечто подобное случайно? Когда на странице есть простой пользовательский тег, например `<sample-component>`, легко позабыть, что он может содержать любое количество элементов, например дополнительную кнопку, с именами классов, которые вы использовали снова и снова. Например, что, если бы на вашей странице был бы такой HTML-код и вы бы хотели добавить к кнопке слушателя события по клику, когда у вашего компонента уже есть кнопка?

```
<sample-component></sample-component>
<button>Click Me</button>
```

Учитывая, что в этом коротком фрагменте кода Click Me – это кнопка в исходном коде страницы, у вас может возникнуть соблазн сделать следующее:

```
document.querySelector('button').addEventListener('click', . . .);
```

В гипотетической ситуации, изображенной на рис. 8.4, наш тег `<sample-component>` уже содержит кнопку, и, что еще хуже, он стилизован так, что даже не выглядит как кнопка! В результате вы использовали метод `querySelector` не для той кнопки и совершенно запутались, не понимая, почему ничего не работает, когда вы пытаетесь нажать на кнопку в браузере.

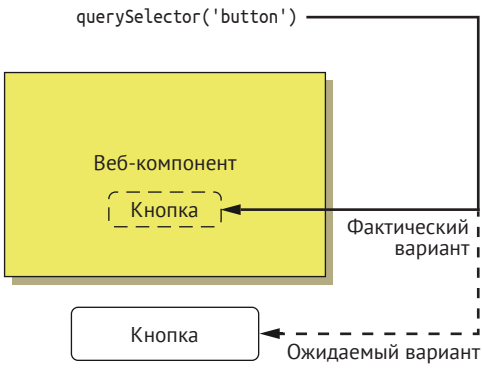



**Рис. 8.4 Ошибочное использование метода `querySelector` для кнопки на странице, в результате чего кнопка не работает**

## 8.2 Использование теневой модели DOM

Теневая модель DOM пытается решить обе проблемы, но для злонамеренных пользователей это не очень удобно. Чтобы объяснить, о чем идет речь, давайте опробуем ее!

Сначала можно попробовать защитить тег  из нашего предыдущего примера от взлома. Используя теневую модель DOM, можно легко заблокировать обычный доступ к этому тегу, и для этого нам просто нужно изменить две строки в нашем методе connectedCallback:

**Листинг 8.2 Использование теневой модели DOM в простом компоненте**

```
connectedCallback() {
   this.attachShadow({mode: 'open'});
   this.shadowRoot.innerHTML =
   `<div class="inside-component">My Component</div>`
}
```

Создаем открытую теневую модель DOM и присоединяем ее к нашему компоненту

Устанавливаем в качестве значения HTML-код нашего компонента

Здесь не так много кода, но он требует объяснения. Первое, что мы делаем, – создаем корень теневого дерева и присоединяем этот корень к нашему компоненту. В этом примере мы используем режим open, чтобы создать его. Обратите внимание, что это обязательный параметр. Поскольку разработчики браузеров не могут договориться о том, какой режим должен быть по умолчанию, open или closed, они возложили это на вас, вместо того чтобы занять какую-то позицию. Разницу между этими режимами легче объяснить, изучив то, что происходит в коде.

Помимо того, является он закрытым или открытым, что такое корень теневого дерева? Не забудьте вернуться к главе 7 и нашему обсуждению тега <template>. Напомню, что основой шаблона стал фрагмент документа. Фрагмент документа – это совершенно отдельное дерево DOM, которое не отображается как часть вашей главной страницы. Корень теневого дерева, по сути, является фрагментом документа. Это означает, что это совершенно отдельная модель DOM! На самом деле это не та же самая модель DOM, что и остальная часть вашей страницы.

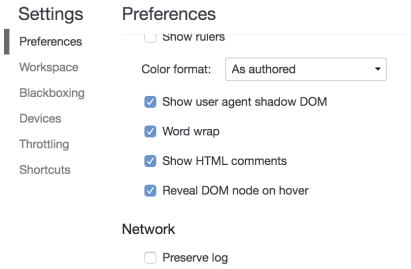
В этом примере можно увидеть корень теневого дерева в действии, открыв инструменты разработчика в браузере Chrome, как показано на рис. 8.5. Чего вы вряд ли ожидали, так это того, что элементы, которые вы используете ежедневно, имеют собственный корень теневого дерева.


**Рис. 8.5 Просмотр теневой модели DOM и связанного с ней корня теневого дерева в инструментах разработчика Chrome**

Давайте посмотрим на тег <video>. Нам не нужно соответствующим образом задавать значение элемента <source> для этого тега, чтобы увидеть его корень теневого дерева и остальную часть его теневой модели DOM. Просто добавьте тег <video></video> в свой HTML-код. Если проверить его

в Chrome, используя настройки по умолчанию, это мало что даст. Чтобы увидеть его теневую модель DOM, нужно активировать пункт «Show user agent shadow DOM», как показано на рис. 8.6. По сути, Chrome отобразит любую созданную вами теневую модель DOM, но по умолчанию будет скрывать ее в обычных элементах браузера, которые ее используют. `<select>` – еще один тег с собственной теневой моделью DOM, который можно просматривать таким образом.

**Настройки Chrome для просмотра теневой модели DOM стандартных элементов**




**Просмотр корня теневого дерева с помощью инспектора элементов в инструментах разработчика браузера Chrome**


**Рис. 8.6 Просмотр теневой модели DOM и корня теневого дерева повседневно используемых элементов**

## 8.2.1 Корень теневого дерева

Поскольку мы переходим к соответствующей терминологии, такой как «корень теневого дерева», ознакомьтесь с терминами, показанными на рис. 8.7:

- *корень теневого дерева* – фрагмент документа, содержащий отдельную модель DOM;
- *теневое дерево* – дерево из DOM-узлов, самый верхний из которых является теневым корнем;
- *теневой хост* – DOM-узел вашей страницы, в котором размещается теневое дерево и его корень. В нашем случае это ваш веб-компонент, хотя его можно легко использовать за пределами пользовательского элемента;
- *теневая граница* – представьте, что это линия между вашим теневым хостом и теневым деревом. Например, если мы попадем в теневое дерево из нашего компонента и зададим текст для кнопки, можно сказать, что мы пересекаем «теневую границу».

Помимо терминологии, важным выводом является то, что мы имеем дело с новой моделью DOM внутри фрагмента документа. В отличие от фрагмента документа, используемого тегом `<template>`, этот фрагмент фактически отображается в браузере, но при этом сохраняет свою независимость.

Мы можем использовать новое и автоматически созданное свойство нашего компонента `shadowRoot`, чтобы обращаться к любому из свойств

нашего элемента, например innerHTML. Вот что мы сделали в нашем примере:

```
this.shadowRoot.innerHTML =
  `<div class="inside-component">My Component</div>`
```

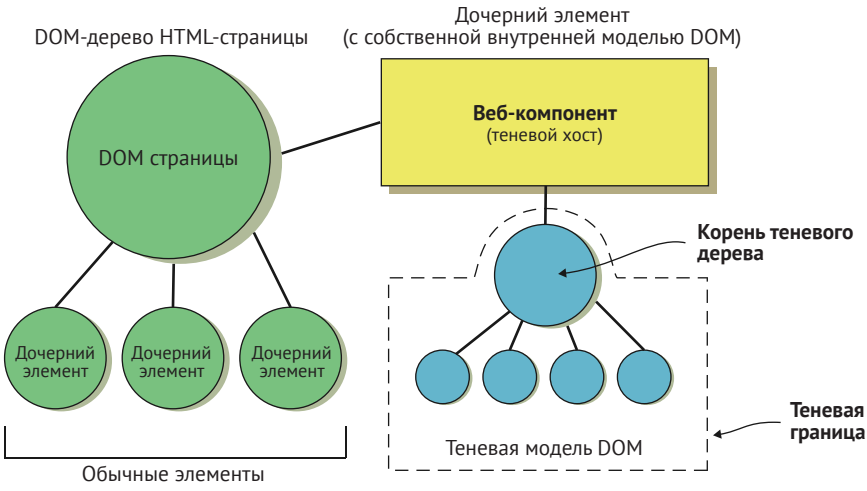



**Рис. 8.7 Теневая модель DOM, хост, корень и граница (пунктирная линия)**

С помощью этого изменения мы защитили наш компонент от случайных вторжений. Теперь, когда мы используем тот же метод querySelector и пытаемся установить значение для свойства innerHTML, то терпим неудачу:

```
document.querySelector('.inside-component').innerHTML +=
  ' has been hijacked';
```

Ошибка гласит:

```
Uncaught TypeError: Cannot read property 'innerHTML' of null
```

Что происходит? Использование метода querySelector (document.querySelector('.inside-component')) ничего не дает, а настройка свойства innerHTML предпринимается для пустого объекта, как показано на рис. 8.8, потому что мы изолировали HTML-код внутри нашего компонента с помощью Shadow DOM.

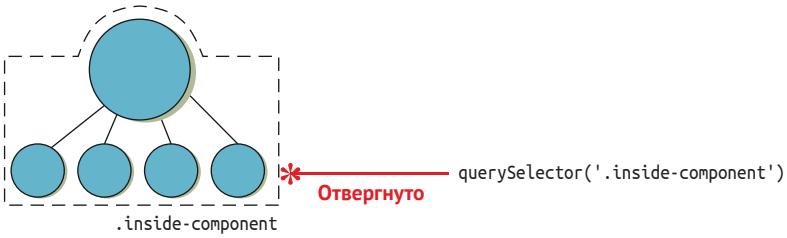



**Рис. 8.8 Попытка использовать метод querySelector внутри теневой модели DOM**

### 8.2.2 Закрытый режим

Но вот в чем дело. Если бы мы хотели действовать как злоумышленники, у нас по-прежнему была бы такая возможность. То же свойство shadowRoot доступно снаружи. Мы могли бы настроить метод querySelector, чтобы сделать его более сложным, и при этом установить значение свойства innerHTML для этого тега :

```
document.querySelector('sample-component').shadowRoot.querySelector
➥ ('.inside-component').innerHTML += ' has been hijacked';
```

Здесь мы показываем JS-код, с помощью которого мы легко устанавливаем значение для свойства innerHTML нашего компонента. Можно ли помешать этим злонамеренным пользователям заходить и манипулировать нашим компонентом? Похоже, что нет, но тут-то и вступает в дело закрытый режим. Ограничить действия таких лиц – вот цель обоих режимов. Чтобы было понятно, давайте установим режим в состояние closed, как показано в приведенном ниже листинге.

**Листинг 8.3 Установка теневого режима в состояние closed**

```
connectedCallback () {
   this.attachShadow({mode: 'closed'});
   this.shadowRoot.innerHTML =
      `<div class="inside-component">My Component</div>`
}
```

Устанавливаем теневой режим в состояние closed

Однако это не сработает так, как задумано, если не поменять еще кое-что! Когда корень теневого дерева закрыт, свойство shadowRoot не существует (он равен null), поэтому мы не можем установить значение для свойства innerHTML с его помощью. Тогда как же нам взаимодействовать с нашим собственным компонентом, работая изнутри?

Вызов метода attachShadow возвращает ссылку на корень теневого дерева, независимо от того, находитесь ли вы в открытом или закрытом режиме. Если вам нужна ссылка только в той же функции, в которой вы создали корень теневого дерева, можно просто объявить переменную следующим образом.

**Листинг 8.4 Использование переменной для ссылки на корень теневого дерева**

```
connectedCallback () {
   const root = this.attachShadow(
      {mode: 'closed'});
   root.innerHTML = `<div class="inside-component">My Component</div>`
}
```

Устанавливаем значение для переменной

Если это единственная точка взаимодействия с теневой моделью DOM вашего компонента, проблема решена! Вы предприняли шаги, чтобы закрыть свой компонент от злонамеренных пользователей... за исключением еще одной вещи. Притворимся, что мы злоумышленники и не остановимся ни перед чем, чтобы саботировать данный компонент. Мы

можем изменить определение функции `attachShadow` после объявления класса компонента:

```
SampleComponent.prototype.attachShadow = function(mode) { return this; };
```

Это действительно очень сложно, но мы изменили функцию `attachShadow`, чтобы она фактически не создавала корень теневого дерева и ничего не делала, а только передавала обратно естественную область видимости веб-компонента. Первоначальный создатель компонента, который намеревался создать закрытую теневую модель DOM, вообще не создает ее. Ссылка на корень – это то, что он намеревался получить, но в действительности это просто область видимости компонента. Эта хитрость по-прежнему работает так же, потому что и у `this`, и у корня теневого дерева примерно одинаковый API.

И теперь мы возвращаемся к нашему исходному и простому способу получения контроля над компонентом:

```
document.querySelector('.inside-component').innerHTML +=
   ' has been hijacked';
```

Стоит ли ожидать, что те, кто используют ваш компонент, попытаются взломать его таким образом? Возможно, нет. Но они могли бы это сделать. Это не настоящая защита, потому что ее очень легко обойти.

Вспомните, что в начале этой главы мы говорили о том, как защитить ваш компонент по-настоящему или сделать это по соглашению. Там мы обсуждали использование нижнего подчеркивания для защиты закрытых переменных и методов в вашем классе, вместо того чтобы использовать более безопасные способы. Здесь то же самое, но вместо переменных и методов мы говорим о DOM вашего компонента.

Вот почему в документации Google по веб-компонентам говорится, что не стоит использовать закрытый режим (https://developers.google.com/web/fundamentals/web-components/shadowdom). Вы закрываете теневую модель DOM, чтобы обезопасить свой компонент, но уверены, что те, кто используют его, не обойдут защиту простыми способами. В конце концов, вы защищаете свой компонент по соглашению независимо от того, что вы делаете; просто закрытый режим усложняет разработку.

Google утверждает, что при закрытом режиме ваш компонент пострадает по двум причинам. Во-первых, пуская пользователей компонента в Shadow DOM своего компонента с помощью свойства `shadowRoot`, вы по меньшей мере создаете лазейку. Независимо от того, используете ли вы нижнее подчеркивание или оставляете теневую модель DOM открытой, ваш класс или компонент защищен по соглашению.

Несмотря на свои лучшие намерения в отношении компонента, вероятно, вы вряд ли будете постоянно учитывать все варианты использования. Возможность попасть в свой компонент дает некую гибкость, но также важно понимать, что это противоречит более взвешенным решениям с вашей стороны как разработчика компонента. Это сигнал для того, кто использует ваш компонент, к тому, что он должен делать это на свой страх и риск. Конечно, это опрометчиво, но когда сроки поджима-

ют, а веб-приложение должно быть отправлено завтра, было бы неплохо указать, в каком направлении двигаться, используя свойство `shadowRoot` для доступа к вещам, которые в настоящее время вам не нужны. Вы также увидите, что лазейка, предоставляемая при использовании открытого режима, довольно удобна для выполнения автоматического тестирования, о чем мы поговорим в главе 13.

Второй недостаток, связанный с закрытым режимом, – утверждение, согласно которому он делает теневую модель DOM недоступной изнутри вашего собственного компонента. Но все несколько сложнее. Свойство `shadowRoot` больше не доступно в закрытом режиме, но мы можем легко сделать ссылку на него.

В нашем примере есть локальная переменная, как показано в этом листинге.

**Листинг 8.5 Локальная переменная корня теневого дерева**

```
connectedCallback() {
   const root = this.attachShadow(  <-- Локальная переменная корня теневого дерева
      {mode: 'closed'});
   root.innerHTML = `<div class="inside-component">My Component</div>`
}
```

Теперь давайте изменим ее.

**Листинг 8.6 Открытое свойство, содержащее корень теневого дерева**

```
connectedCallback () {
   this.root = this.attachShadow(  <-- Корень теневого дерева сохранен в качестве открытого свойства
        {mode: 'closed'});
   this.root.innerHTML = `<div class="inside-component">My Component</div>`
}
```

С другой стороны, превращение переменной в открытое свойство противоречит самой сути. И снова у вас есть открытая ссылка на теневую модель DOM; она носит имя `root` (это может быть любое другое имя на ваш выбор) вместо свойства `shadowRoot`, созданное открытой теневой моделью DOM. И опять же, с ее помощью легко получить доступ к DOM вашего компонента. Тем не менее если бы вы использовали более надежный способ защиты свойств вашего класса, например ассоциативные массивы со слабыми ссылками (weak maps), чтобы сделать ваши свойства закрытыми, это все равно не было бы надежным, но очень хорошо закрывало бы все и разрешало бы внутренний доступ к вашей закрытой модели DOM. Возможно, стоит предположить, что по-настоящему закрытую теневую модель DOM можно получить, когда у нас будут нативные закрытые поля классов, доступные во всех браузерах, но пока такого еще нет.

Понятно, что закрытая теневая модель DOM в большинстве случаев того не стоит. Абсолютно надежного способа полной блокировки вашего компонента не существует, и защита компонента по соглашению с использованием открытой теневой модели DOM – это выход.

### 8.2.3 Конструктор вашего компонента и метод connectedCallback: сравнение

Еще в главе 4 при обсуждении API компонента я предупреждал, что конструктор не очень полезен при инициализации вашего компонента. Это связано с тем, что, когда он запускается, у него по-прежнему нет доступа к свойству и методам, связанным с DOM вашего компонента, таким как `innerHTML`.

Теперь с Shadow DOM ничего не изменилось по отношению к DOM страницы.

При использовании теневой модели DOM у вашего компонента по-прежнему не будет доступа к свойствам и методам, связанным с DOM вашего элемента, пока он не будет добавлен в DOM страницы с помощью метода `connectedCallback`.

Несмотря на все это, больше это не проблема. Мы больше не полагаемся на DOM страницы, а создаем отдельную мини-модель DOM для нашего компонента, когда вызываем функцию `attachShadow`. Эта мини-модель становится доступной сразу же, и мы можем сразу написать ее свойство `innerHTML`!

Вот почему вы увидите, что в большинстве примеров веб-компонентов для выполнения всей работы по инициализации используется конструктор вместо метода `connectedCallback`, который мы использовали до сих пор. В дальнейшем по ходу книги, скорее всего, я все буду делать в конструкторе, потому что буду использовать Shadow DOM. Но важно помнить об этом различии, учитывая, что теневая модель DOM – всего лишь одна из частей головоломки, связанной с веб-компонентами, и, как таковая, не является обязательной (даже если вы, вероятно, захотите использовать ее здесь и далее).

Давайте немного изменим наш предыдущий простой пример, чтобы продемонстрировать это.

**Листинг 8.7 Использование конструктора вместо метода connectedCallback**

```
<html>
<head>
    <script>
        class SampleComponent extends HTMLElement {
            constructor() {
                super();
                this.attachShadow({mode: 'open'});
                this.shadowRoot.innerHTML =
                 `<div class="inside-component">My Component</div>`
            }
        }

        if (!customElements.get('sample-component')) {
            customElements.define('sample-component', SampleComponent);
        }
```

Конструктор

Необходимо вызвать метод super(), так как мы расширяем HTMLElement

Устанавливаем значение для свойства innerHTML в конструкторе

```
    </script>
</head>
<body>
    <sample-component></sample-component>
</body>
</html>
```

Хотя теневая модель DOM выглядит довольно потрясающе, она может быть ненадежна. Я говорю не о ее реализации или спецификации, а о медленном включении ее в качестве поддерживаемой функции во всех современных браузерах, о чем я упоминал в начале этой главы. Лично я до недавнего времени находился в режиме ожидания. Когда в октябре прошлого года Firefox включил поддержку веб-компонентов, и, зная, что Edge готовится к этому, теперь я с радостью использую Shadow DOM в своих новых проектах.

Что делать, если ваш браузер не поддерживает Shadow DOM? Очевидный ответ – использовать полифил, как и в случае с любой другой функцией. К сожалению, данный вариант представляет некоторые сложности для Shadow DOM.

Самая большая проблема при использовании полифилов – тема следующей главы. Что касается защиты от случайных вторжений в ваш компонент, мы рассмотрели API вашего компонента и его локальную модель DOM, когда доступ осуществляется через JS. Их прекрасно можно защищать с помощью инкапсуляции, которую нам предоставляет теневая модель DOM. Однако я мог бы поспорить и сказать, что защита от проникающих CSS-правил является абсолютным лучшим вариантом использования Shadow DOM. Причина, по которой я так люблю ее, состоит в том, что веб-разработчики боролись с этой проблемой, поскольку CSS широко распространен и известен, и ситуация только ухудшилась, т. к. веб-разработка стала более сложной. Существуют довольно новые обходные пути, но Shadow DOM полностью снимает эту проблему.

В настоящее время усилия по полизаполнению теневой модели DOM разделены на эти два варианта использования. Мы поговорим о CSS и полифилах в следующей главе. Однако полизаполнение доступа с использованием JS к вашей модели DOM действительно простое. В главе 2, когда мы применяли полизаполнение для пользовательских элементов, то использовали полифил для пользовательского элемента.

Можно пойти немного шире и охватить все, что не поддерживается. Полифилы, о которых идет речь на странице www.webcomponents.org/polyfills, предлагают интеллектуальные функции обнаружения и, при необходимости, заполняют функции. Сюда входят как пользовательские элементы, так и Shadow DOM.

В качестве одного из вариантов можно использовать это:

```
npm install @webcomponents/webcomponentsjs
```

а затем добавить на свою страницу тег `<script>`:

```
<script src="node_modules/@webcomponents/webcomponentsjs/
    ➥webcomponents-bundle.js"></script>
```

Кроме того, доступен вариант с CDN. В конце у нас должен получиться код, который работает во всех современных браузерах, как показано в приведенном ниже листинге.

**Листинг 8.8 Компонент с полифилом**

```
<html>
<head>
    <script src="https://unpkg.com/@webcomponents/webcomponentsjs@2.0.0/
    ➥webcomponents-loader.js"></script>
    <script>
        class SampleComponent extends HTMLElement {
            constructor() {
                super();
                this.root = this.attachShadow({mode: 'open'});
            }

            connectedCallback() {
                if (!this.initialized) {
                    this.root.innerHTML = 'setting some HTML';
                    this.initialized = true;
                }
            }

        }

        if (!customElements.get('sample-component')) {
            customElements.define('sample-component', SampleComponent);
        }
    </script>
</head>
<body>

<sample-component></sample-component>

<script>
    setTimeout(function() {
        document.querySelector('sample-component').innerHTML =
        ➥'Component is hijacked';
    }, 500);
</script>
</body>
</html>
```

Полифил, загруженный из CDN

Устанавливаем значение для свойства innerHTML нашего компонента снаружи

Мы используем полифил, а затем тестируем его, пытаясь установить значение для свойства `innerHTML` нашего компонента. Для этого я использовал здесь таймер, чтобы убедиться, что мы пытаемся перехватить компонент, после того как он попытается установить собственный текст в методе `connectedCallback`. При использовании теневой модели DOM в большинстве браузеров установка значения для свойства `innerHTML` снаружи компонента оканчивается неудачей. При использовании полифила и «Shady DOM» в браузерах, которые не поддерживают Shadow DOM, например Microsoft Edge (в ближайшее время поддержка появится) и IE, происходит то же самое.

Однако, как я упоминал ранее, Shady DOM довольно хорошо работает в случае доступа к DOM с помощью JS. Shady CSS – это отдельная история, о которой мы поговорим в следующей главе!

## Резюме

Из этой главе вы узнали:

- что такое инкапсуляция и что отдельный объект – это только половина дела. Также важно обеспечивать защиту и предлагать контролируемый доступ к своему объекту;
- что Shadow DOM предлагает защиту внутренней модели DOM вашего компонента, и она наиболее полезна при случайных вторжениях извне;
- что хотя Shadow DOM предлагает закрытый режим, это нецелесообразно, и защита вашего компонента по соглашению с открытой теневой моделью DOM – это путь вперед, особенно потому, что он предлагает способ обойти свою защитную границу в крайнем случае;
- о различиях между конструкторами и методом `connectedCallback` для работы с DOM вашего компонента при использовании или неиспользовании теневой модели DOM;
- как использовать полифилы с Shady DOM и о том, что существует отдельное решение для инкапсуляции CSS.

# 9 Shadow CSS

***Эта глава охватывает следующие темы:***

- сохранение внешних стилей за пределами ваших веб-компонентов;
- теневая модель DOM для инкапсуляции CSS;
- CSS-селекторы теневой модели DOM;
- повторное обнаружение атрибута идентификатора веб-компонентов.

Давайте продолжим изучение теневой модели DOM! В предыдущей главе мы сосредоточились на действительно прекрасном аспекте Shadow DOM. Каким бы замечательным ни было инкапсулирование DOM, CSS еще лучше! Несмотря на то что в течение многих лет мы разрабатывали умные способы борьбы с утечкой стилей в нашей веб-разработке, это всегда было проблемой.

## 9.1 Утечка стилей

Иногда *утечка стилей* может стать головной болью в процессе веб-разработки. Если говорить кратко, речь идет о явлении, когда правила CSS влияют на элементы, которых вы не и собирались касаться. Возможно,

вы работаете над стилизацией элемента где-то в одном месте, но некоторые правила стилей, которые вы определили в своем CSS-файле для другого элемента на своей странице, подгружаются непреднамеренно, потому что селекторы CSS совпадают. Хотя утечка стилей не ограничивается веб-компонентами, давайте рассмотрим в качестве примера веб-компонент, чтобы увидеть, как это влияет на нас.

На рис. 9.1 показан простой маленький веб-компонент, который, по сути, представляет собой стилизованный числовой счетчик с возможностью увеличения или уменьшения значения.


**Рис. 9.1 Стилизованный счетчик с двумя кнопками и текстовым интервалом**

Для этого гипотетического варианта использования допустим, что независимо от того, как выглядят другие кнопки в нашем веб-приложении, важно, чтобы этот счетчик был красного цвета, а кнопки «плюс» и «минус» находились вровень по бокам от цифры посередине. Посмотрите на приведенный ниже листинг.

**Листинг 9.1 Счетчик без логики, только стили**

```
<html>
<head>
    <script>
        class SampleComponent extends HTMLElement {
            connectedCallback() {
             this.innerHTML = `
                <button class="big-button">-</button>
                <span class="increment-number">5</span>
                <button class="big-button">+</button>
                <style>
                  sample-component {
                     display: flex;
                  }
                  sample-component .increment-number {
                    font-size: 24px;
                       background-color: #770311;
                       color: white;
                       font-family: Helvetica;
                       display: inline-block;
                       padding: 11px;
                       border: none;
                    }

                    sample-component button {
                      border-radius: 0 50px 50px 0;
                      border: none;
                      width: 50px;
                      height: 50px;
                      font-size: 36px;
```

Кнопка «минус»

Текущее значение

Кнопка «плюс»

Стили компонента

Стили компонентов, продолжение

```
                    font-weight: bold;
                    background-color: red;
                    color: white;
                  }

                  sample-component button:first-child {
                    border-radius: 50px 0 0 50px;
                  }

                  sample-component .big-button:active {
                    background-color: #960000;
                  }

                  sample-component .big-button:focus {
                    outline: thin dotted;
                  }
                </style>`;
            }
        }

        if (!customElements.get('sample-component')) {
            customElements.define('sample-component', SampleComponent);
        }

    </script>
</head>
<body>
<sample-component></sample-component>
</body>
</html>
```

Пример компонента на странице

Обратите внимание, как каждому правилу стиля предшествуют слова `sample-component`. В таком простом примере с одним компонентом на странице не обязательно писать `.sample-component button`. В конце концов, у нашего компонента здесь есть все кнопки на странице. Однако кнопка – это такой распространенный элемент, что, как только мы начнем добавлять на нашей странице другое содержимое, стили этой кнопки начнут влиять на это содержимое. Создавая правило, конкретно для `.sample-component`, мы предотвращаем утечку стилей из этого компонента в другие элементы.

Полезно освежить в памяти то, как работают такие глобальные стили. На рис. 9.2 видно, что правила CSS, которые мы определяем в нашем компоненте, становятся частью глобального пространства стилей страницы. В свою очередь, эти стили будут влиять на все элементы нашей страницы.

### 9.1.1 Утечка стилей в нижестоящие компоненты

Даже с учетом этой специфики утечка может произойти и другим способом. Что, если бы у нас был еще один компонент внутри этого компонента с собственными кнопками? У этих кнопок по-прежнему где-то есть тег `<sample-component>`, поэтому здесь стили будут утекать во все нижестоящие компоненты.

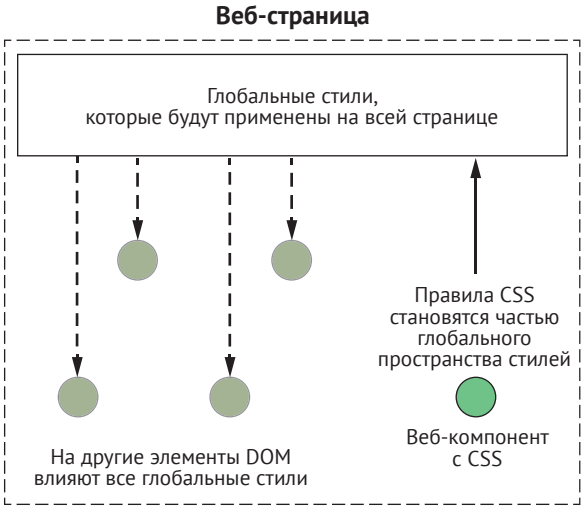



**Рис. 9.2 Без использования теневой модели DOM стили, определенные в вашем веб-компоненте, будут применяться ко всей странице**

Вы неизбежно столкнетесь с утечкой стилей, независимо от того, насколько специфичны ваши селекторы, и вам придется приступать к отладке. Но опять же веб-разработчики всегда сталкивались с этой проблемой. Тем не менее при использовании веб-компонентов легче пропустить подобные проблемы, потому что мы склонны рассматривать компоненты, с которыми работаем, как отдельные, инкапсулированные объекты и пропускать внутреннее содержимое при сканировании DOM в инструментах отладки.

## 9.1.2 Утечка стилей в ваш компонент

Итак, допустим, вы учли все. Вы тщательно спланировали имена своих классов и правила CSS, чтобы быть хорошим разработчиком компонентов и не допустить утечки стилей из своих компонентов. Это только полдела – стили по-прежнему могут проникать в ваш компонент со страницы и других родительских компонентов.

Давайте представим, что ваше веб-приложение управляется некой дизайн-системой. Такие системы, как Bootstrap, определяют единый внешний вид ваших веб-страниц или приложений. Например, вы, вероятно, захотите, чтобы большинство кнопок в вашем приложении приняли единый вид, как показано на рис. 9.3.


**Рис. 9.3 Пример глобально стилизованной кнопки, предлагаемый дизайн-системой**

В этом листинге мы добавим эту кнопку на нашу страницу с помощью простого элемента кнопки и стилей на уровне страницы.

**Листинг 9.2 Стилизованная кнопка, сосуществующая на нашей странице с веб-компонентом**

```
<head>
    <style>
        button {  <-- Стили, не относящиеся к компоненту
            border-top: 1px solid #96d1f8;
            background: #65a9d7;
            background: linear-gradient(90deg, #3e779d, #65a9d7);
            padding: 5px 10px;
            border-radius: 8px;
            box-shadow: rgba(0,0,0,.5) 0 8px 8px;
            text-shadow: rgba(0,0,0,.4) 0 2px 2px;
            color: white;
            font-size: 14px;
            font-family: Helvetica;
            text-decoration: none;
            vertical-align: middle;
        }
        button:hover {
            border-top-color: #28597a;
            background: #28597a;
            color: #ccc;
        }
        button:active {
            border-top-color: #1b435e;
            background: #1b435e;
        }
    </style>

    <script>
        . . . такое же определение компонента, как и раньше
    </script>
</head>
<body>
<sample-component></sample-component>
<br /><br />
<button>Button from Design System</button>  <-- Элемент кнопки, не относящийся к компоненту
</body>
</html>
```

Глядя на результаты, показанные на рис. 9.4, уже можно увидеть, что стили кнопки проникают в наш компонент и ведут себя не очень хорошо.

Приступим к адаптации внешнего вида кнопки в нашем счетчике. У нас есть тень и синий градиентный фон. Конечно, теперь они не совпадают с цифрой посередине. Когда вы нажимаете на кнопку, все становится еще хуже – цвет фона меняется на красный. В общем, получается путаница!

Все это вызвано тем, что в общих стилях кнопок есть несколько других правил, по сравнению с кнопкой нашего компонента. Цвета фона счетчика переопределяются фоном универсальной кнопки. И конечно же, у кнопки счетчика не должно быть свойств `text-shadow` или `box-shadow`, как у обычной кнопки.

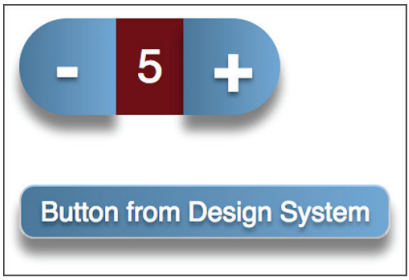



**Рис. 9.4 Как глобальные стили кнопки могут негативно повлиять на наш компонент**

Мы даже не вдавались в специфику правил! Представьте, что у нашей универсальной кнопки также есть вариант «big-button», что точно соответствует названию правила внутри нашего компонента.

Давайте вернемся и сделаем такой вариант, увеличив размер шрифта и отступ для этой кнопки, чтобы она стала «большой кнопкой». Наша цель – получить нечто похожее на наши предыдущие универсальные кнопки, изображенные на рис. 9.3 и 9.4, просто размером побольше.

Однако реальность такова, что когда мы определяем эту разновидность, меняя все наши правила кнопок в стилях вне компонента с `button{}` на `button.big-button{}`, то получаем неожиданные результаты. Используя дополнительную специфичность правил, подобную этой, и одинаковое название «big button» для обеих кнопок (внутри нашего компонента и снаружи), мы только что создали ситуацию, при которой правила, которые мы определили за пределами нашего компонента, являются более специфичными, чем те, что находятся внутри. В действительности это отрицательно сказывается на форме кнопок нашего счетчика, показанных на рис. 9.5, которые мы тщательно определили с помощью правила `border-radius`.


**Рис. 9.5 Дополнительная специфичность и одноименные классы еще больше портят внешний вид компонента**

Разумеется, это можно исправить. Мы можем добавить еще больше специфичности в наши CSS-селекторы внутри компонента, так же как мы это сделали для универсальной кнопки, и перейти от `button{}` к `button.big-button{}`. Однако, кроме того, мы должны отказаться от свойств, которые не охвачены в нашем компоненте, определенных в нашей универсальной кнопке:

```
sample-component button.big-button {
  box-shadow: none;
```

```
  text-shadow: none;
  padding: 0;
}
```

После этих изменений мы возвращаемся к нашему компоненту, который выглядит просто отлично. Теперь очевидно, что следует быть начеку, когда речь идет о проблемах подобного рода. Насколько начеку, в действительности зависит от того, как вы можете контролировать окружающее приложение и предвидеть, что эти стили могут просочиться и навредить вам. Ситуация «кнопка против счетчика» действительно помогла бы, если бы правила для элемента `<button>` в целом не определялись в глобальных стилях. Также было бы полезно создать дополнительные уникальные имена.

Как бы это ни было похоже на беспорядок, а это он и есть, нам, веб-разработчикам, постоянно приходилось сталкиваться с этим. Учитывая все вышесказанное, теневая модель DOM обещает исправить ситуацию!

## 9.2 Проблема утечки стилей решается с помощью теневой модели DOM

В предыдущей главе мы увидели, что создание корня теневого дерева в нашем компоненте привело к появлению отдельной и независимой модели DOM: доступ к ней был ограничен, и JS-вызовы не могли проникать туда, чтобы менять элементы или использовать метод `querySelector` для компонентов. Учитывая все сказанное и сделанное, это было суперпросто!

Мы можем защитить DOM нашего веб-компонента таким же образом и здесь. С помощью приведенного ниже листинга мы можем вернуться к нашему счетчику и использовать теневую модель DOM.

**Листинг 9.3 Использование теневой модели DOM для защиты стилей нашего счетчика**

```
class SampleComponent extends HTMLElement {
    connectedCallback() {
        const root =
            this.attachShadow({mode: 'open'});
        root.innerHTML = `<button class="big-button">-</button>
                          <span class="increment-number">5</span>
                          <button class="big-button">+</button>
            <style>
                sample-component {
                   display: flex;
                }

                span {
                   font-size: 24px;
                   background-color: #770311;
                   color: white;
                   font-family: Helvetica;
                   display: inline-block;
```

Создаем корень теневого дерева для использования Shadow DOM

При наличии менее крупной и более управляемой модели DOM CSS-селекторы не должны быть такими специфичными

```
                padding: 11px;
                border: none;
            }
            button {
                border-radius: 0 50px 50px 0;
                border: none;
                width: 50px;
                height: 50px;
                font-size: 36px;
                font-weight: bold;
                background: none;
                background-color: red;
                color: white;
            }

            button:first-child {
                border-radius: 50px 0 0 50px;
            }

            button:active {
                background-color: #960000;
            }

            button:focus {
                outline: thin dotted;
            }
        </style>`;
    }
}
```

Я не только ввел теневую модель DOM в наш компонент, но также немного взволнован и удалил все свои специфические правила. Мои CSS-селекторы теперь задают только правила для общих тегов `<button>` и `<span>`. После всего того, с чем нам пришлось иметь дело в этом примере, а также долгих лет этих проблем с CSS, возникающих при веб-разработке, это выглядит медлительным и склонным к поломке, не так ли?

Но дело в том, что теперь, когда у нас есть отдельная модель DOM и мы знаем, что наш компонент очень прост, как наш счетчик, мы можем совершенно спокойно применить стили к своим элементам здесь, и это прекрасно! Стили не будут просачиваться, как показано на рис. 9.6, и не будут утекать и влиять на дочерние компоненты, которые также используют теневую модель DOM.

Однако листинг 9.3 еще не идеален. По большей части рис. 9.7 выглядит нормально, но у счетчика присутствуют нежелательные отступы.

Что здесь случилось? У нашего компонента было свойство `display: flex`. Оно осталось, но не работает:

```
sample-component {
  display: flex;
}
```

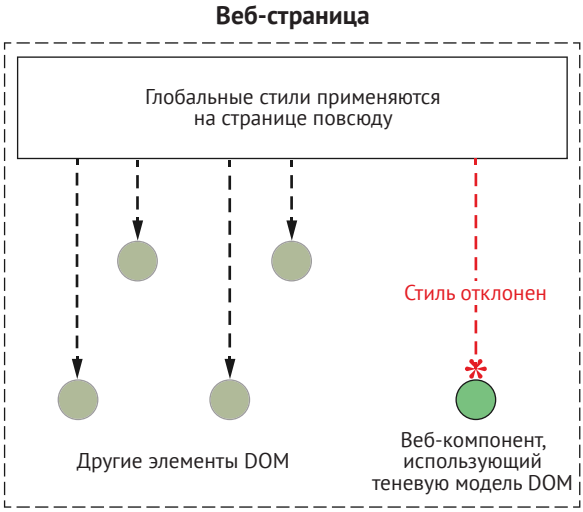



**Рис. 9.6 Веб-компоненты, использующие теневую модель DOM, не подвержены влиянию CSS-стилей на уровне страницы**


**Рис. 9.7 Почти исправленный счетчик находится рядом с кнопкой, к которой применены глобальные стили**

Это объясняется тем, что тег `<sample-component>` теперь находится за пределами нашей теневой модели DOM. Говоря технически, тег, который обозначает наш компонент, является теневым хостом, а в этом хосте содержится корень теневого дерева, где находится наша теневая модель DOM. Поскольку стили не могут утечь в нее, правило с использованием `sample-component` теперь не имеет смысла, чтобы добиться того, что нам здесь нужно.

Вместо этого для стилизации теневой модели DOM применяются новые способы использования CSS-селекторов. Первый – это новый селектор `:host`. Селектор `:host` служит для стилизации того, что находится внутри теневого хоста, как показано на рис. 9.8. Когда мы меняем наш селектор на

```
:host {
  display: flex;
}
```

это приводит в действие свойство `display: flex`.

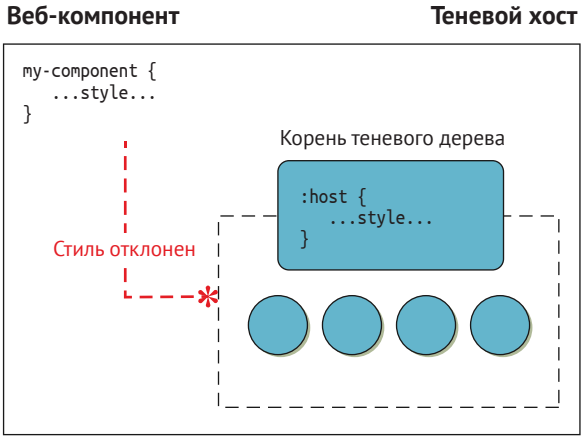



**Рис. 9.8 CSS-стили для теневого хоста (или использование тега компонента в качестве селектора) не проникнут в корень теневого дерева или теневую модель DOM**

## 9.2.1 Когда происходит утечка стилей

Однако в такого рода инкапсуляции *есть* некоторые нюансы. Теневая модель DOM прекрасно подходит для защиты от внешних стилей, попадающих в компонент, который находится под защитой Shadow DOM. Нюанс состоит в том, что мы защищаемся от утечки стилей, когда речь идет об определении с помощью селектора, а не стилей в целом. Чтобы объяснить, что я имею в виду, давайте опробуем еще один пример из приведенного ниже листинга, где мы определим стили для тега `<body>` вне теневой модели DOM.

**Листинг 9.4 Стили, влияющие на теневую модель DOM**

```
<html>
<head>
    <style>
        .text {                                  Стили для текста на внешней странице
            font-size: 24px;
            font-weight: bold;
            color: green;
        }
    </style>

    <script>                                     Тег <span>, в котором содержится текст внутри
                                                 теневой модели DOM нашего компонента
        class SampleComponent extends HTMLElement {
            connectedCallback() {
                const root = this.attachShadow({mode: 'open'});
                root.innerHTML = `<span>Some Text</span>`;
            }
        }
```

```
        if (!customElements.get('sample-component')) {
            customElements.define('sample-component', SampleComponent);
        }
    </script>
</head>
<body class="text">  <-- Применяем стили текста ко всему телу страницы
    <sample-component></sample-component>
</body>
</html>
```

Итак, что вы ожидаете здесь увидеть? Я обещал, что теневая модель DOM защитит вас от стилей, проникающих в ваш компонент, но при запуске примера, как видно на рис. 9.9, тег `<span>` содержит текст зеленого цвета, набранный жирным шрифтом!


**Рис. 9.9 Крупный жирный текст зеленого цвета указывает на то, что внешний стиль влияет на содержимое нашей теневой модели DOM**

Это связано с тем, что нюанс, о котором я говорю, заключается в том, что мы действительно защищаемся от CSS-селекторов извне, которые могут прилипнуть к классам внутри. Тем не менее когда к предку вашего компонента (используется теневая модель DOM или нет), которому не нужно ничего выбирать внутри вашего компонента, применяется какой-то стиль, эти стили все равно будут влиять на дочерние элементы. Теперь, если мы удалим класс text из тела следующим образом:

```
<body>
```

и поместим его в тег `<span>` внутри нашего компонента:

```
root.innerHTML = `<span class="text">Some Text</span>`;
```

то увидим, что стили не имеют никакого эффекта, как показано на рис. 9.10.


**Рис. 9.10 Когда мы помещаем класс прямо в тег `<span>`, теневая модель DOM успешно блокирует стили**

Селектор `"text"` не может проникнуть в теневую модель DOM, но те же правила, как стили извне, могут это сделать. Тем не менее даже что-то столь же простое, как внешние стили тега `<button>`, не будет просачиваться таким же образом, потому что `"button"` по-прежнему является селектором (хотя и универсальным). Это может быть очень полезно и имеет большой смысл. Если весь текст на вашей странице стилизован определенным образом или ваша страница имеет определенный цвет фона, вам не нужно, чтобы ваши компоненты отклонялись от этих основных стилей.

Что, если вы не хотите, чтобы просочились даже эти стили? Мы можем сделать небольшую хитрость, используя селектор `:host`.

**Листинг 9.5 Сброс стилей в теневой модели DOM**

```
<script>
    class SampleComponent extends HTMLElement {
        connectedCallback() {
            const root = this.attachShadow({mode: 'open'});
            root.innerHTML = `<span>Some Text</span>
            <style>
                :host {
                    all: initial;
                }
            </style>`;
        }
    }

    if (!customElements.get('sample-component')) {
        customElements.define('sample-component', SampleComponent);
    }
</script>
```

Применяем первоначальные стили ко всем элементам в корне теневого дерева

Хотя мы, безусловно, можем установить для каждого отдельного правила стилей значение `"initial"`, чтобы сбросить их, более важно сбросить все в нашем корне теневого дерева, используя свойство `all` и совершенно новый селектор `:host`.

Чтобы выйти за пределы селектора `:host` и исследовать дальше, давайте запустим новый демопроект, чтобы надлежащим образом опробовать теневую модель DOM!

## 9.3 План тренировок

Итак, это проект двойного значения. Да, мы будем проходить упражнения с теневой моделью DOM, чтобы познакомиться с новыми концепциями, но то, что мы будем делать, также является приложением для просмотра упражнений и создания тренировок.

Конечный продукт в этой главе не будет настолько интерактивным, каким он мог бы быть, потому что мы продолжим работать с этим демопроектом в главе 14, когда будем рассказывать о событиях для реализации остальной части функционала. В этой главе в итоге у нас появится библиотека упражнений слева и ваш собственный план тренировок справа, как показано на рис. 9.11. При нажатии кнопки мыши на каждое упражнение в библиотеке оно будет добавляться в ваш план.

Типы упражнений делятся на «силовые» или «кардио» и обозначаются синими или зелеными полосками соответственно. Чтобы страница выглядела проще, и поскольку у меня нет личной подборки видеороликов с упражнениями, которыми я могу поделиться с вами, мои миниатюры и фон будут серого цвета. Однако в репозитории GitHub для этой книги я включил ссылки на изображения в формате GIF из моей модели данных, определенной в файле components/exerciselibrary/exerciselibrary.js, чтобы каждое упражнение отображалось с миниатюрой движения, что позволит вам правильно просмотреть упражнение.

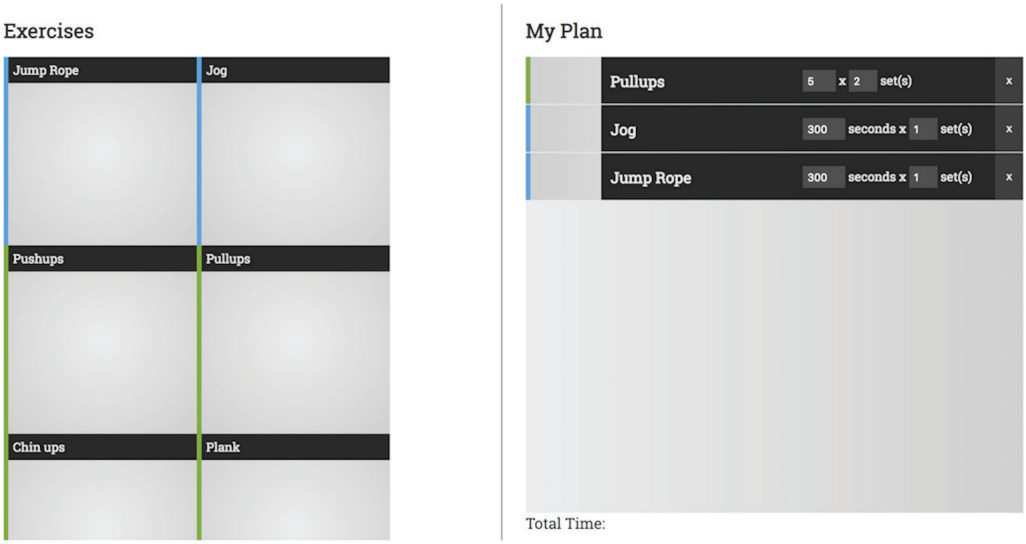



**Рис. 9.11 Демонстрационное приложение для просмотра упражнений из библиотеки и создания собственного плана тренировок**

## 9.3.1 Оболочка приложения

В качестве первого шага давайте создадим общую структуру приложения вместе с заполнителями для дочерних компонентов. В частности, мы создадим HTML-страницу, CSS-файл и компонент `<wkout-creator-app>`, где файловая структура выглядит, как показано на рис. 9.12. Если вы подписаны, пожалуйста, не забудьте использовать какой-нибудь простой веб-сервер, учитывая, что у нас есть зависимости, загруженные из нашего файла index.html, которые, возможно, не будут работать при использовании одной только файловой системы.

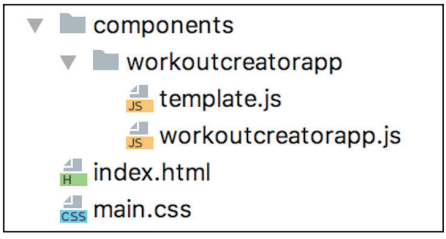



**Рис. 9.12 Базовая файловая структура, когда мы приступаем к созданию приложения**

Как и в других наших примерах, файл index.html будет очень простым, как показано в приведенном ниже листинге.

**Листинг 9.6 Файл index.html нашего приложения**

```
<html>
<head>
   <title>Workout Creator</title>
```

```
   <script type="module"                                    Импорт компонента
           src="components/workoutcreatorapp/workoutcreatorapp.js">
   </script>
   <link rel="stylesheet" type="text/css" href="main.css">
</head>
<body>
   <wkout-creator-app></wkout-creator-app>   Компонент, объявленный в HTML-коде
</body>
</html>
```

Наши стили еще проще. Мы просто обнуляем отступы и поля у всех элементов на странице, в то же время изменяя размер компонента `<wkout-creator-app>`, чтобы он занимал всю страницу с небольшим отступом.

**Листинг 9.7 Файл main.css для нашего приложения**

```
body {                                  Обнуляем поля и отступы на странице
   margin: 0;
   padding: 0;
}                                       Устанавливаем размеры приложения,
                                        чтобы оно занимало всю страницу
wkout-creator-app {
   height: calc(100vh - 20px);
   padding: 10px;
}
```

Что касается самого компонента `<workout-creator-app>`, его код, показанный в приведенном ниже листинге, также очень прост.

**Листинг 9.8 Основной компонент нашего приложения**

```
import Template from './template.js';

export default class WorkoutCreatorApp extends HTMLElement {
   constructor() {
       super();                                  Используем Shadow DOM
       this.attachShadow({mode: 'open'});        в нашем компоненте
       this.shadowRoot.innerHTML = Template.render();
   }
}

if (!customElements.get('wkout-creator-app')) {
   customElements.define('wkout-creator-app', WorkoutCreatorApp );
}
```

Обратите внимание, что, в отличие от предыдущих примеров, сейчас мы используем теневую модель DOM. Кроме того, в отличие от того, что мы делали ранее, теперь мы выполняем все настройки нашего компонента в конструкторе и напрямую используем свойство `shadowRoot` для доступа к нашей локальной теневой модели DOM.

Наконец, я буду использовать функции стилей этой модели, а также делать вещи, которые вы никогда бы не сделали без нее. От этого нелегко отказаться! Итак, я ставлю все на Shadow DOM, и пути назад нет.

## 9.3.2 Селекторы host и ID

Продолжая работу над нашим модулем `WorkoutCreatorApp`, который определяет компонент `<wkout-creatorapp>`, давайте взглянем на модуль template.js, содержащий наш HTML-код и стили, в приведенном ниже листинге.

**Листинг 9.9 Модуль шаблона приложения, определяющий HTML-код и стили**

```
export default {
   render() {
       return `${this.css()}
               ${this.html()}`;
   },

   html() {
       return `<wkout-exercise-lib>
               </wkout-exercise-lib>
               <div id="divider-line"></div>
               <wkout-plan></wkout-plan>`;
   },

   css() {
       return `<style>
                   :host {
                      display: flex;
                   }

                   wkout-exercise-lib,
                   wkout-plan {
                       flex: 1;
                       height: 100%;
                       background-color: #eaeaea;
                   }

                   #divider-line {
                       width: 1px;
                       height: 100%;
                       margin-right: 25px;
                       background-color: black;
                   }
               </style>`;
   }
}
```

Контейнер слева для библиотеки упражнений

Разделительная линия с атрибутом ID

Контейнер справа для списка тренировок

Прежде всего мы создаем три дочерних элемента. Два из них – это компоненты, которые еще не определены, поэтому они будут отображаться как пустые элементы `<div>`; они оформлены в цвете фона, поэтому на данный момент мы можем визуализировать их размещение, как показано на рис. 9.13. Посередине находится черная разделительная линия.

Даже имея этот код, у нас есть два момента для обсуждения касательно теневой модели DOM. Во-первых, мы используем ранее упомянутый CSS-селектор `:host`, чтобы назначить стили нашему хост-компоненту.

В этом случае мы просто хотим использовать тип отображения `"flex"` для разметки трех наших элементов.

Второй момент является важным. Он выглядит незначительным, но на самом деле очень серьезен. Нашей разделительной линии присваивается идентификатор `"divider-line"` в `<div id="dividerline"></div>`. Затем мы используем этот идентификатор, чтобы назначить стили: `#divider-line {}`.



**Рис. 9.13 Так выглядит наше приложение в браузере на данный момент**

Почему это так важно? В каждом веб-разработчике укоренилось мнение, что нужно использовать атрибут ID экономно. Причина состоит в том, что во всей DOM может быть только один элемент с таким идентификатором. Если вы ошиблись и назначили второй элемент с одним и тем же идентификатором, у вас могут возникнуть проблемы со стилями или методом `querySelector`, когда вы сможете выбрать или стилизовать только один из нескольких элементов с одинаковым идентификатором.

Как правило, наши селекторы представляют собой несколько классов, объединенных вместе, для того чтобы получить специфичность, необходимую для точного выбора или стилизации элемента. Для нашей разделительной линии мы могли бы использовать CSS-селектор, который выглядит так:

```
wkout-creator-app div.divider-line.center.thin {}
```

Да, сейчас я выгляжу немного смешно, используя `.center` и `.thin`, но я просто пытаюсь подчеркнуть, что можно переусердствовать в отношении специфичности, которая обычно необходима.

Однако теперь мы можем использовать теневую модель DOM. Возвращаясь к вопросу о том, что каждый идентификатор во всей вашей моде-

ли DOM должен быть уникальным, помните, что сейчас мы используем несколько моделей DOM!

Ваш идентификатор должен быть уникальным только внутри области видимости вашего веб-компонента. Элемент с идентификатором `#divider` может легко находиться в другом месте на странице или в других веб-компонентах, и никакого конфликта не будет.

Еще лучше, учитывая, что в этом веб-компоненте всего три элемента, и только разделительная линия использует тег `<div>`, можно было бы не беспокоиться об идентификаторе, вместо этого используя такой селектор: `div {}`.

Лично я думаю, что это действительно захватывающе. Возвращаясь к моменту, когда я познакомил вас с теневой моделью DOM в предыдущей главе, я сказал, что она устраняет хрупкость веб-разработки. Это яркий тому пример. Мы можем сосредоточиться на структуре и стилях нашего компонента и не беспокоиться о конфликтах где-либо еще. Наши селекторы могут быть настолько просты и легки для чтения, насколько это позволяет внутренняя структура нашего компонента.

### 9.3.3 Сетка упражнений и список планов

Сейчас мы продолжим работу с концепциями, которые только что изучили, чтобы создать сетку упражнений и наш список планов тренировок. Это еще два компонента, благодаря которым структура нашего проекта выглядит как на рис. 9.14.

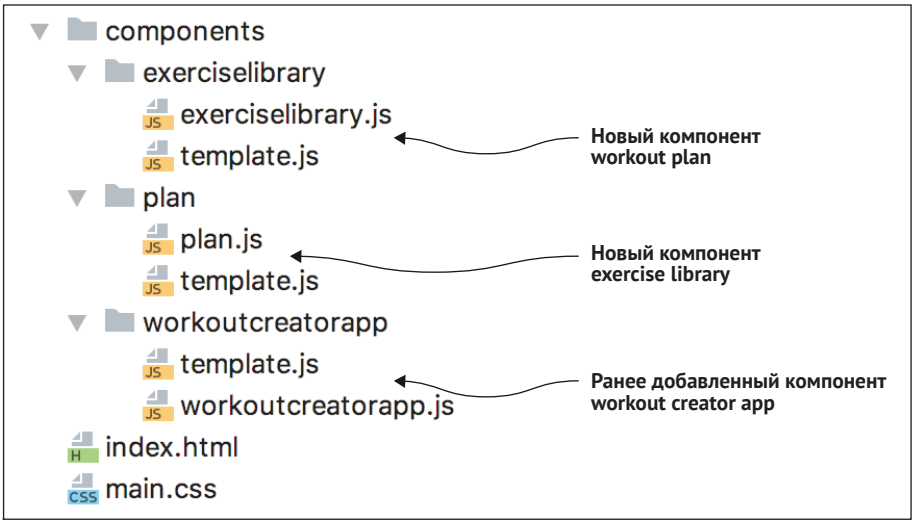



**Рис. 9.14 Файловая структура проекта, после того как в нее были добавлены два компонента: библиотека упражнений и план тренировок**

Помните, что мы фактически отображаем компоненты `<wkout-plan>` и `<wkout-exerciselib>`, которые уже есть в компоненте приложения; прос-

то они еще не определены, поэтому отображаются как элементы . Таким образом, наш первый шаг после создания новых файлов и папок компонентов состоит в том, чтобы импортировать эти модули во главе workoutcreatorapp/template.js:

```
import ExerciseLibrary from '../exerciselibrary/exerciselibrary.js';
import Plan from '../plan/plan.js';
```

Определившись с этим, давайте приступим к конкретизации данных компонентов!

На самом деле оба они довольно простые. Во многом это связано с тем, что мы еще не уделяем никакого внимания интерактивности. В приведенном ниже листинге показаны наши файлы plan/plan.js и plan/template.js.

**Листинг 9.10 Файлы компонента «план тренировки»**

```
// Plan.js
import Template from './template.js';

export default class Plan extends HTMLElement {
   constructor() {
       super();
       this.attachShadow({mode: 'open'});
       this.shadowRoot.innerHTML =
           Template.render();
   }
}

if (!customElements.get('wkout-plan')) {
   customElements.define('wkout-plan', Plan);
}

// Template.js
export default {
   render() {
       return `${this.css()}
               ${this.html()}`;
   },

   html() {
       return `<h1>My Plan</h1>
               <div id="container"></div>
               <div id="time">Total Time:</div>`;
   },

   css() {
       return `<style>
                   :host {
                       display: flex;
                       flex-direction: column;
                   }

                   #time {
                       height: 30px;
```

Назначаем HTML-код и стили нашему компоненту

HTML-код, который будет отображаться

Стили, которые будут отображаться

```
                }

                #container {
                    background: linear-gradient(90deg, rgba(235,235,235,1)
                        0%, rgba(208,208,208,1) 100%);
                    height: calc(100% - 60px);
                    overflow-y: scroll;
                }
            </style>`;
    },
}
```

Поскольку наш список из плана тренировок пока еще пуст, не отображается ничего, кроме контейнера, текста заголовка и нижнего колонтитула, чтобы показать общую продолжительность плана.

И снова мы используем теневую DOM, что позволяет нам применять идентификаторы элементов для тегов `<div id="container"></div>` и `<div id="time"></div>` с целью стилизации. В обоих случаях мы просто устанавливаем размер и цвет заливки фона, а также делаем так, чтобы контейнер для списка упражнений прокручивался, когда он становится слишком высоким. Кроме того, мы снова используем селектор `:host`, чтобы теневой корень нашего компонента отображался с использованием вертикального выравнивания.

Компонент `<wkout-exerc-lib>` не сильно отличается, за исключением того, что нам нужно заполнить его данными. Цель этого компонента – показывать список упражнений на выбор, поэтому все они должны присутствовать при загрузке приложения. Таким образом, мы будем отображать заголовок и контейнер, как и последний компонент, но также будем заполнять контейнер нашими упражнениями. В приведенном ниже листинге показаны файлы exerciselibrary/exerciselibrary.js и exerciselibrary/template.js.

**Листинг 9.11 Файлы компонента «библиотека упражнений»**

```
// exerciselibrary.js ◄──── Модуль определения компонента для библиотеки упражнений
import Template from './template.js';

export default class ExerciseLibrary extends HTMLElement {
   constructor() {
       super();
       this.attachShadow({mode: 'open'});
       this.shadowRoot.innerHTML = Template.render([
           { label: 'Jump Rope', type: 'cardio', thumb: '', time: 300, sets: 1},
           { label: 'Jog', type: 'cardio', thumb: '', time: 300, sets: 1},
           { label: 'Pushups', type: 'strength', thumb: '', count: 5, sets: 2,
             estimatedTimePerCount: 5 },
           { label: 'Pullups', type: 'strength', thumb: '', count: 5, sets: 2,
             estimatedTimePerCount: 5},
           { label: 'Chin ups', type: 'strength', thumb: '', count: 5, sets: 2,
             estimatedTimePerCount: 5},
           { label: 'Plank', type: 'strength', thumb: '', time: 60, sets: 1}
           ]);
```

```
    }
}

if (!customElements.get('wkout-exercise-lib')) {
    customElements.define('wkout-exercise-lib', ExerciseLibrary);
}

// template.js
export default {
    render(exercises) {
        return `${this.css()}
                ${this.html(exercises)}`;
    },

    html(exercises) {

        let mkup = `<h1>Exercises</h1>
                    <div id="container">`;
        for (let c = 0; c < exercises.length; c++) {
            mkup +=
            `<wkout-exercise class="${exercises[c].type}" ></wkout-exercise>`;
        }
        return mkup + `</div>`;
    },

    css() {
    return `<style>
          host {
                 display: flex;
                 flex-direction: column;
               }

                #container {
                    overflow-y: scroll;
                    height: calc(100% - 60px);
                }
            </style>`;
    }
}
```

Шаблонный модуль для библиотеки упражнений, в котором хранится наш HTML-код и стили

Перебираем упражнения и отображаем их

Вы сразу же заметите большой список упражнений, которые мы вводим в функцию `Template .render`. Каждое упражнение имеет метку, а также тип: `cardio` или `strength`. В зависимости от того, учитываете ли вы количество повторений или просто выполняете упражнение в течение определенного промежутка времени, в упражнении будет указано число для count и `sets` или `time`. Если мы отслеживаем количество и настройки, единственный способ оценить общее время нашей тренировки – вычислить, сколько времени занимает каждое повторение нашего упражнения, поэтому мы используем еще одно свойство с именем `estimatedTimePerCount`.

Наконец, в каждом упражнении есть пустое свойство `thumb`. Как я уже говорил в начале этой главы, мы оставим его пустым, чтобы не показывать миниатюру в этой книге. Вы можете поискать свои собственные изображения или GIF-файлы в интернете, чтобы вставить их, или найти

на GitHub репозиторий для данной книги, чтобы просмотреть те изображения, которые нашел я. Также в моем репозитории есть дополнительные упражнения для нашей модели данных.

Наш файл exerciselibrary/template.js в основном совпадает с предыдущим файлом plan/template.js. Конечно, главное отличие в том, что мы принимаем список упражнений и отображаем каждое из них. Опять же, нам еще предстоит определить компонент `<wkout-exercise>`, а пока мы сосредоточимся на том, что дает нам нечто похожее, изображенное на рис. 9.15.

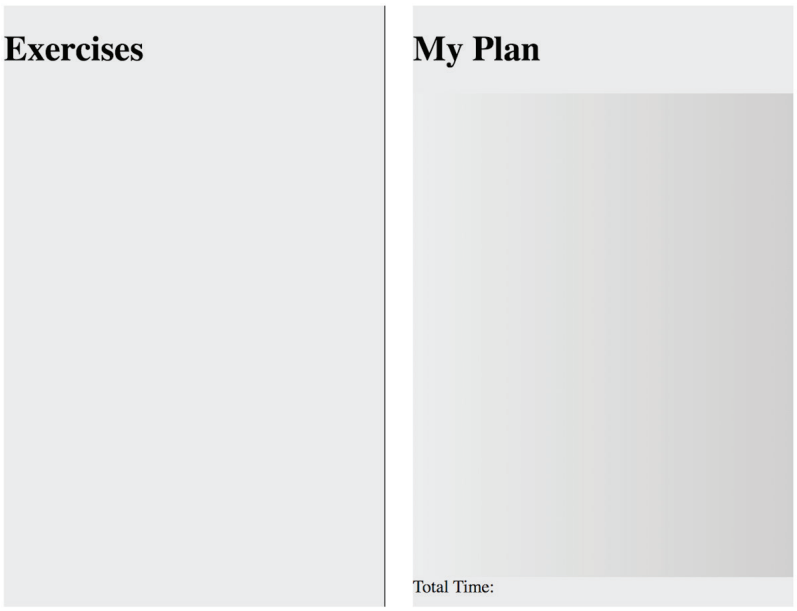



**Рис. 9.15 Заполнение компонентов в левой и правой частях приложения**

Вы заметите, что хотя мы отобразили наши упражнения, они не появляются. Все дело в том, что хотя они есть в DOM, у них нет размера или фона – поэтому, несмотря на то что они присутствуют, их высота составляет 0 пикселей и они не отображаются визуально. Мы решим этот вопрос с помощью компонента `<wkout-exercise>`. Это последнее, о чем нужно рассказать, и это довольно интересный компонент.

## 9.4 Адаптируемые компоненты

Почему я нахожу компонент `<wkout-exercise>` таким интересным? Потому, что мы приступаем к компоненту, который должен выглядеть немного иначе, в зависимости от того, как он используется, и изучим альтернативный способ использования селектора `:host`. В следующей главе мы более подробно рассмотрим этот адаптируемый компонент и сделаем так, чтобы он выглядел совершенно иначе в контейнере плана тренировки.

### 9.4.1 Создание компонента упражнения

Поскольку нашему плану тренировок нужна интерактивность, давайте сначала сосредоточимся на библиотеке упражнений, поскольку выполнять итерацию по стилю для того, что появляется при загрузке страницы, проще, нежели требовать дополнительного шага нажатия кнопки «Добавить». Нам, конечно, понадобится создать файлы компонентов, и в итоге мы получим файловую структуру, показанную на рис. 9.16.

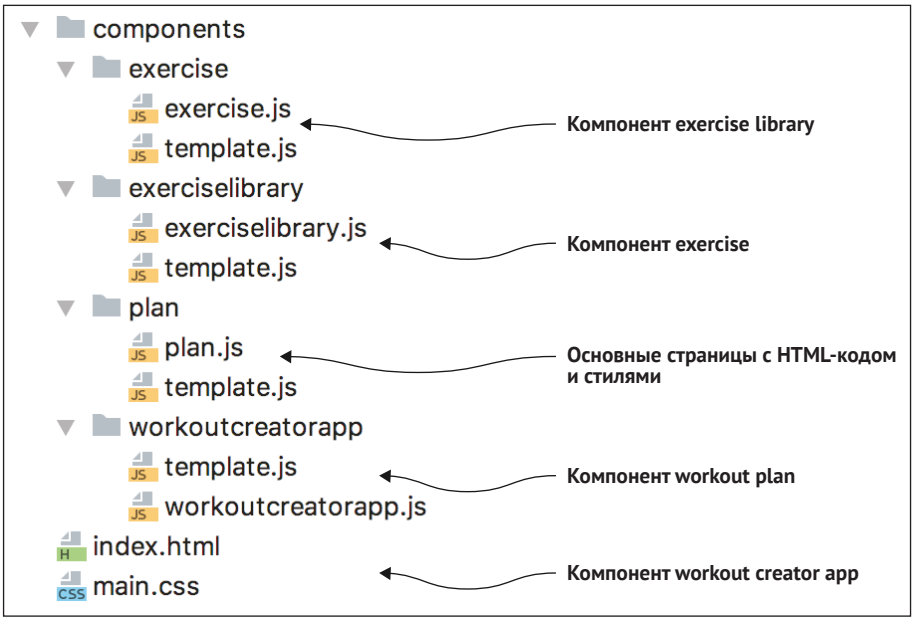



Рис. 9.16 Конечная файловая структура приложения

Поскольку и план тренировки, и библиотека упражнений отображают компонент упражнения, мы должны поместить этот импорт в модули plan/template.js и exerciselibrary/template.js:

```
import Exercise from '../exercise/exercise.js';
```

Давайте посмотрим на определение веб-компонента для `<wkout-exercise>` в приведенном ниже листинге.

**Листинг 9.12 Файлы для компонента упражнения**

```
import Template from './template.js';

export default class Exercise extends HTMLElement {
   constructor() {
       super();
       this.attachShadow({mode: 'open'});

       const params = {
           label: this.getAttribute('label'),
```

```
            type: this.getAttribute('type'),
            thumb: this.getAttribute('thumb'),
            time: this.getAttribute('time'),
            count: this.getAttribute('count'),
            estimatedTimePerCount: this.getAttribute('estimatedtimepercount'),
            sets: this.getAttribute('sets'),
        };
        this.shadowRoot.innerHTML = Template.render(params);
    }
  get label() { return this.getAttribute('label'); }
  set label(val) { this.setAttribute('label', val); }
  // Дополнительные геттеры и сеттеры для свойств thumb, type, time, count,
  // estimateTimePerCount и sets;
  serialize() {
     return {
         label: this.label,
         type: this.type,
         thumb: this.thumb,
         time: this.time,
         count: this.count,
         estimatedTimePerCount: this.estimatedTimePerCount,
         sets: this.sets,
     }
  }
  static toAttributeString(obj) {
       let attr = '';
       for (let key in obj) {
           if (obj[key]) {
               attr += key + '="' + obj[key] + '" ';
           }
       }
       return attr;
  }
}

if (!customElements.get('wkout-exercise')) {
   customElements.define('wkout-exercise', Exercise);
}
```

Геттеры и сеттеры для каждого свойства

Функция для сериализации всех свойств в объект

Функция, чтобы выполнить сборку строки атрибута для клонированного компонента упражнения

В целях экономии пространства я убрал все, кроме одного, из моих методов получения/установки. В этом определении компонента мы используем то, что у нас было в главе 3. Мы применяем рефлексию, чтобы использовать атрибуты и свойства взаимозаменяемо. Мы можем использовать либо `element.setAttribute(property, value)` для элемента, либо `element.property = value` для настройки свойства. В любом случае, получаем или устанавливаем данные, которые внутренне основаны на атрибуте элемента. Если бы я не сократил код, у нас были бы методы получения/установки для `thumb`, `type`, `time`, `count`, `estimateTimePerCount` и `sets`.

Два других метода используются для сбора данных. Во-первых, у нас есть метод `serialize`. Он просто собирает наши данные в один объект,

который мы можем легко передать. Другой статический метод `toAttributeString` действует похожим образом. Он собирает все наши данные, как это делает `serialize`, но создает строку, которую можно использовать для заполнения атрибутов. В итоге мы получим строку в формате

```
свойство="значение" свойство2=" значение2" свойство3=" свойство3"
```

Этот дополнительный метод может и не показаться необходимым, но мы хотим отсеять неопределенные свойства. Помните, что из-за различий в упражнениях некоторые из них будут иметь свойство подсчета повторений, например когда вы поднимаете вес, в то время как другие будут иметь свойство продолжительности, например когда вы бегаете трусцой. Поэтому это хорошая альтернатива вместо наличия `property="undefined"` в качестве атрибута в нашем теге, когда фактическое неопределенное значение преобразуется в строку, или необходимости проверять наличие слова `undefined` для каждого свойства в наших шаблонах, делая их длиннее и труднее для чтения. Все это объясняет, почему в файле exerciselibrary/template.js мы изменим наш цикл в функции `html()` на:

```
for (let c = 0; c < exercises.length; c++) {
   mkup += `<wkout-exercise class="${exercises[c].type}"
     ${Exercise.toAttributeString(exercises[c])}></wkout-exercise>`;
}
```

Таким образом, мы можем создавать атрибуты в нашем новом элементе для каждого *действительного* свойства в наших данных. Поскольку это статический метод (доступ к которому осуществляется из класса, а не из экземпляра класса), мы можем использовать его либо для объектов необработанных данных, которые у нас есть в файле exerciselibrary/exerciselibrary.js перед созданием компонента, либо для уже созданного компонента `<wkoutexercise>`, чтобы скопировать эти значения. Будь то простой объект или компонент, все свойства присутствуют и могут использоваться одним и тем же способом этим методом. Тег, который мы получаем в итоге, выглядит как один из приведенных ниже вариантов, в зависимости от упражнения:

```
<wkout-exercise class="cardio" label="Jog" type="cardio" time="300"
     sets="1"></wkout-exercise>

<wkout-exercise class="strength" label="Pushups" type="strength" count="5"
     sets="2" estimatedtimepercount="5"></wkout-exercise>
```

## 9.4.2 Стили компонента упражнений

Разобравшись со всеми необходимыми нам атрибутами и создав определение компонента, остается сделать последнее: создать HTML-код и стили, представленные в этом листинге:

**Листинг 9.13 Первый проход компонента тренировки**

```
export default {
   render(exercise) {
```

```
        return `${this.css(exercise)}
                ${this.html(exercise)}`;
    },

    html(exercise) {
        return `<div id="info">
                    <span id="label">${exercise.label}</span>
                    <span id="delete">x</span>
                </div>`;
    },

    css(exercise) {
        return `<style>
                    :host {                          Стили для всего компонента
                        display: inline-block;
                        background: radial-gradient(circle,
                        rgba(235,235,235,1) 0%, rgba(208,208,208,1) 100%);
                    /*background-image:
                        url('${exercise.thumb}');*/
                        border-left-style: solid;
                        border-left-width: 5px;
                    }

                    :host(.cardio) {
                        border-left-color: #28a7ff;
                    }

                    :host(.strength) {
                        border-left-color: #75af01;
                    }

                    #info {
                        font-size: small;
                        display: flex;
                        align-items: center;
                        background-color: black;
                        color: white;
                    }

                    :host {
                        width: 200px;
                        height: 200px;
                        background-size: cover;
                    }

                    :host #info {
                        padding: 5px;
                    }
                </style>`;
    }
}
```

Закомментированный фон с миниатюрой

Общий стиль компонента с вариацией для класса в теге компонента

Теперь, когда все это собрано воедино, наш компонент `<wkout-exercise>` отображает все имеющиеся у нас компоненты `<wkout-exercise>`. На рис. 9.17 первое, что следует отметить, – это фоны наших компонентов:

```
background: radial-gradient(circle, rgba(235,235,235,1) 0%,
     rgba(208,208,208,1) 100%);
/*background-image: url('${exercise.thumb}');*/
```


**Рис. 9.17 Компоненты упражнений с новыми стилями**

Я закомментировал фоновое изображение, но если вы нашли в интернете отличные миниатюры для каждого упражнения и добавили их к данным в компоненте `<wkoutexercise-lib>`, можете раскомментировать эту строку. Если мы этого не сделали, то просто показываем градиентный серый фон.

Также обратите внимание на то, насколько прост HTML-код. Мы показываем блок размером 200×200 пикселей с черной полосой вверху. Для библиотеки это нормально, но вы, наверное, представляете себе, что для отображения в виде представления списка в плане упражнений все это может быть немного проблематично.

И снова мы используем концепции, которые рассматривали ранее в этой главе. Мы идентифицируем и выбираем элементы с помощью атрибута ID, а также с помощью селектора `:host` в контексте теневого корня нашего компонента.

Однако обратите внимание на то, что у нас есть небольшая разновидность селектора `:host`:

```
:host(.cardio) {
     border-left-color: #28a7ff;
}

:host(.strength) {
```

```
            border-left-color: #75af01;
        }
```

Вернувшись при рендеринге каждого из этих компонентов, мы добавили класс `strength` или `cardio` для каждого компонента:

```
mkup += `<wkout-exercise class="${exercises[c].type}"
  ${Exercise.toAttributeString(exercises[c])}></wkout-exercise>`;
```

Эта разновидность селектора `:host` позволяет нам рассматривать любые классы самого тега компонента и использовать его для большей специфичности стилей. Чтобы было понятнее, если говорить коротко, `:host(.cardio)` позволяет нам стилизовать элемент `<wkout-training class="cardio">` на основе класса `cardio`. На практике эти разные цвета границ дают возможность пользователю различать два разных типа упражнений при просмотре библиотеки. Есть еще несколько CSS-селекторов, которые вы, возможно, видели в сети и о которых я здесь не рассказывал, но они не поддерживаются либо устарели. Мы закончим делать компонент `<wkout-exercise>` адаптируемым к различным контекстам в следующей главе, а пока поговорим о проблемах, связанных с Shadow DOM.

## 9.5 Обновляем ползунок

Перед тем как приступить к изучению проблем, связанных с Shadow DOM, и обновлению приложения Workout Creator, мы узнали достаточно, чтобы обновить наш ползунок, над которым мы работали на протяжении всей этой книги. Приятный момент – менять нужно не так уж и много!

Прежде всего давайте используем теневую модель DOM. Ранее код инициализации компонента находился в функции `connectedCallback`, но теперь мы знаем, что можно использовать конструктор, потому что есть теневая модель DOM. Этот конструктор показан в приведенном ниже листинге. Имейте в виду: мы полностью удалили метод `connectedCallback`, переместив установочный код сюда.

**Листинг 9.14 Конструктор ползунка**

```
constructor() {                                  // Функционал метода connectedCallback перенесен в конструктор
    super();
    this.attachShadow({mode: 'open'});           // Присоединяем теневое дерево DOM
    this.shadowRoot.innerHTML =                  // Используем свойство shadowRoot вместо this для области видимости
       Template.render();
    this.dom = Template.mapDOM(this.shadowRoot);

    document.addEventListener('mousemove', e => this.eventHandler(e));
    document.addEventListener('mouseup', e => this.eventHandler(e));
    this.addEventListener('mousedown', e => this.eventHandler(e));
}
```

Кроме того, поскольку конструктор запускается до функции `attributeChangedCallback`, проблемы с синхронизацией, с которой мы сталкива-

лись ранее при использовании метода connectedCallback, больше не возникает. Вы заметите, в коде установки нашего конструктора больше нет этих строк:

```
this.refreshSlider(this.getAttribute('value'));
this.setColor(this.getAttribute('backgroundcolor'));
```

Также нам больше не нужно проверять, существует ли свойство this.dom, как мы делали ранее:

```
setColor(color) {
    if (this.dom) { . . .
```

Конечно, такая проверка не повредит. Но поскольку вся инициализация происходит до изменения входящего атрибута при запуске компонента, это просто не нужно.

Модуль template.js также может немного измениться. Помимо использования селектора :host для корня компонента, теперь мы можем использовать идентификаторы вместо классов для применения стилей и выбора. Как я уже упоминал, использование идентификаторов – это роскошь, которую мы не могли себе позволить раньше, когда у нас не было инкапсулированной модели DOM. В приведенном ниже листинге показан новый файл template.js для ползунка.

**Листинг 9.15 Новый модуль шаблона ползунка**

```
export default {
    render() {
        return `${this.css()}
                ${this.html()}`;
    },

    mapDOM(scope) {
        return {
            overlay: scope.getElementById(
'bg-overlay'),
            thumb: scope.getElementById('thumb'),
        }
    },

    html() {
        return `<div id="bg-overlay"></div>
                <div id="thumb"></div>`;
    },

    css() {
        return `<style>
                    :host {
                        display: inline-block;
                        position: relative;
                        border-radius: 3px;
                    }

                    #bg-overlay {
```

Теперь, применяя идентификаторы, мы будем использовать метод getElementById вместо querySelector

Ссылаемся на элементы по идентификатору вместо класса

Используем селектор :host для стилизации всего компонента

Используем идентификаторы для стилей вместо классов

```
                width: 100%;
                height: 100%;
                position: absolute;
                border-radius: 3px;
            }

            #thumb {
                margin-top: -1px;
                width: 5px;
                height: calc(100% - 5px);
                position: absolute;
                border-style: solid;
                border-width: 3px;
                border-color: white;
                border-radius: 3px;
                pointer-events: none;
                box-shadow: 0 4px 8px 0 rgba(0, 0, 0, 0.2), 0 6px
                    20px 0 rgba(0, 0, 0, 0.19);
            }
        </style>`;
    }
}
```

Теперь, когда в нашем компоненте работает теневая модель DOM, мы сделали почти все, что нужно было сделать с этим конкретным компонентом. Хотя пока мы не будем оставлять его в покое! Этот ползунок станет неотъемлемой частью более крупного компонента, который мы создадим в последних главах данной книги, где также рассмотрим тестирование, процесс сборки и запуск веб-компонентов в IE11.

## Резюме

Из этой главы мы узнали:

- как CSS-стили могут проникать в ваш веб-компонент и выходить из него, если не использовать теневую модель DOM;
- что Shadow DOM полностью защищает DOM вашего компонента от внешних стилей;
- что при применении Shadow DOM можно использовать меньше конкретики в отношении наших CSS-селекторов, полностью полагаясь на преимущества отдельной модели DOM;
- как использовать определенные CSS-селекторы Shadow DOM для стилизации вашего компонента, делая это по-разному в разных контекстах.

# 10 Проблемы Shadow CSS

***Эта глава охватывает следующие темы:***

- не пользующийся широкой поддержкой селектор `host-context()`;
- устаревшие селекторы `::shadow` и `/deep/`;
- CSS-переменные;
- полизаполнение теневой модели DOM;
- дизайн-системы.

В предыдущих двух главах я нарисовал довольно радужную картину Shadow DOM. Не волнуйтесь, я не изменю своего мнения! Как бы хороша ни была теневая модель DOM, есть несколько предостережений, о которых следует знать. К сожалению, эти предостережения, вероятно, являются наиболее запутанной частью веб-компонентов. Устаревшие функции, функции, которые не поддерживаются в определенных браузерах, или просто необходимость знать, как работать в браузерах, которые вообще не поддерживают Shadow DOM, – все это может быть несколько сложно.

## 10.1 Контекстные селекторы

Первое, о чем нужно знать, – это селектор `:host-context()`. Сам по себе он не является устаревшим – просто он не поддерживается ни в одном

браузере, кроме Chrome. Дальше хуже. Разработчики браузера Safari, основанного на свободно распространяемом коде движка WebKit, еще в 2016 году заявили, что никогда больше не будут поддерживать его, потому что считают его антипаттерном (https://bugs.webkit.org/show_bug.cgi?id=160038). Разработчики Firefox также полагают, что реализация его в движке Gecko – не очень хорошая идея, из-за того, что это влияет на производительность, и даже открыли тикет на странице https://github.com/w3c/csswg-drafts/issues/1914.

Итак, у нас остался симпатичный маленький селектор, который никто, кроме Chrome, похоже, не хочет поддерживать, но он по-прежнему является частью спецификации Shadow DOM. Хуже того, чтобы узнать, что происходит на самом деле, вам нужно выполнить поиск в общедоступных списках рассылки каждого браузера или отслеживать проблемы!

Что это значит? Если бы мне пришлось делать ставки, я бы сказал, что селектор `:host-context()` будет исключен из спецификации Shadow DOM, а Chrome, вероятно, удалит его когда-нибудь в далеком будущем, хотя бы для поддержания здравого смысла и общего набора функций Shadow DOM в браузерах.

Интересно, что Angular поддерживает селектор `:host-context()`, и, учитывая, что это фреймворк, ему не требуется поддержка со стороны браузера, чтобы использовать его. В общем и целом ситуация с этим селектором немного запутанная. Лично я думаю, что важно подумать о том, что предлагает этот селектор и как преодолеть проблему, если он недоступен, когда мы хотим использовать его. Если согласны, читайте дальше! Если нет, переходите к разделу 10.2.

> **ПРЕДУПРЕЖДЕНИЕ** В оставшейся части раздела 10.1 говорится о селекторе:`host-context()`, который хотя и не считается устаревшим, скорее всего, станет таковым в будущем.

### 10.1.1 Немного интерактивности

Чтобы приступить к использованию этого нового, но злополучного селектора, давайте вернемся к нашему приложению Workout Creator. В главе 9, когда мы закончили, дела у нас шли довольно хорошо. Каждое упражнение в нашем наборе данных было визуализировано с помощью миниатюрного изображения в представлении библиотеки слева, в то время как справа у нас – пустой контейнер, ожидающий использования, как показано на рис. 10.1.

Как вы можете себе представить, следующий очевидный шаг – позволить пользователю добавлять упражнения из библиотеки слева в свой план тренировок справа. Поскольку в предыдущей главе мы начали с более тонких контекстных стилей, мы будем использовать тот же компонент упражнения для обеих областей. Теперь разница состоит в том, что вместо разноцветных линий для обозначения типов упражнений наш компонент `<wkout-exercise>` будет выглядеть совершенно по-разному в двух разных контейнерах.

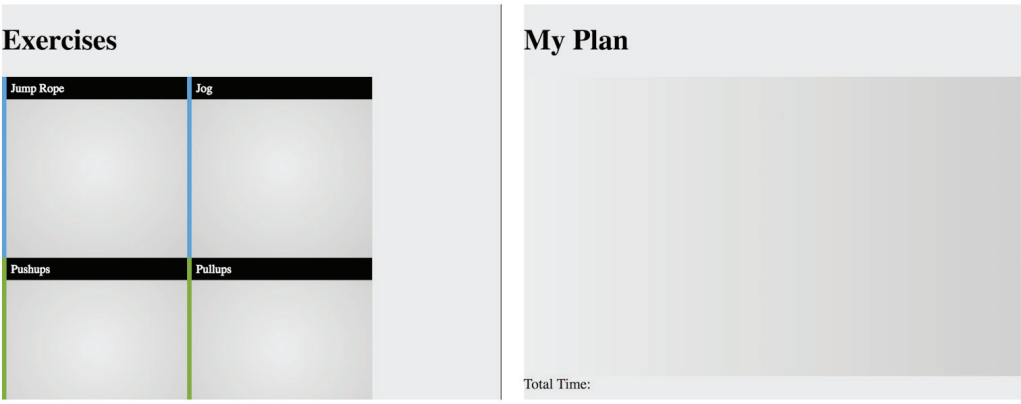



Рис. 10.1 На чем мы остановились в главе 9

Учитывая это, нам нужно активировать некое взаимодействие. Давайте сделаем пару вещей, чтобы привести в действие компонент `<wkout-creator-app>`. Для начала мы можем кешировать ссылки на элементы для обоих контейнеров (библиотеки и плана).

Внутри файла workoutcreatorapp/template.js давайте добавим это:

```
mapDOM(scope) {
   return {
       library: scope.querySelector('wkout-exercise-lib'),
       plan: scope.querySelector('wkout-plan')
  };
},
```

Следует помнить, что метод `mapDOM` – просто мой любимый способ сохранения ссылок на элементы, при этом метод `querySelector` остается за пределами основного класса компонента.

Пока мы находимся в этом файле, напомню, что для облегчения просмотра вначале мы добавили цвет фона для обоих используемых здесь компонентов, чтобы их можно было видеть по мере продвижения. Теперь, когда мы разобрались со всем этим, давайте удалим этот цвет:

```
wkout-exercise-lib,
wkout-plan {
   flex: 1;
   height: 100%;
   background-color: #eaeaea;
}
```

← Удалите эту строку

Когда ссылки на элементы отобразятся в этом модуле, мы можем вернуться в файл workoutcreatorapp/workoutcreatorapp.js и добавить слушателя событий по клику, как показано в приведенном ниже листинге.

**Листинг 10.1 Добавление слушателя событий по клику, чтобы разрешить выбирать упражнения**

```
import Template from './template.js';
import Plan from "../../plan/plan.js";
```

```
import Exercise from "../../exercise/exercise.js";

export default class WorkoutCreatorApp extends HTMLElement {
   constructor() {
       super();
       this.attachShadow({mode: 'open'});
       this.shadowRoot.innerHTML = Template.render();
       this.dom =
               Template.mapDOM(this.shadowRoot);
       this.shadowRoot.addEventListener('click',
           e => this.onClick(e));
   }

   onClick(e) {
       const path = e.composedPath().reverse();
       for (let c = 0; c < path.length; c++) {
           if (path[c] instanceof Plan) {
               return;

           }
           if (path[c] instanceof Exercise) {
               const exercise = path[c];
               this.dom.plan.add(exercise);
           }
       }
   }
}

if (!customElements.get('wkout-creator-app')) {
   customElements.define('wkout-creator-app', WorkoutCreatorApp );
}
```

Сохраняем ссылки на элементы в объект для последующего использования

Назначаем слушателя событий по клику нашему компоненту

Метод-обработчик кликов

В конструкторе есть всего два небольших дополнения. Во-первых, мы вызываем метод `mapDOM`, который только что добавили из шаблона, поэтому теперь `this.dom` содержит все наши ссылки на элементы. Во-вторых, мы добавляем слушателя событий по клику в наш компонент.

Когда вы посмотрите на содержимое функции `onClick` и увидите, что слушатель событий по клику находится в компоненте `<wkout-creator-app>`, а не в компоненте библиотеки упражнений, то можете подвергнуть сомнению мои методы. И будете правы! Причина, по которой я все делаю таким грязным способом, заключается в том, что мы можем надлежащим образом изучить события и дизайн приложений с помощью этого примера в главе 14.

Даже если вы знакомы с событиями и пользовательскими событиями, которые я должен был использовать здесь, у теневой модели DOM действительно есть некоторые недостатки. На данный момент это быстрый и грязный способ, чтобы слушать события и реагировать на них. Здесь мы проходим все элементы, через которые прошло событие click, чтобы добраться до этой функции, и если оно пришло из компонента, определенного классом `Plan`, мы выходим из функции. Но если оно пришло откуда-то еще и есть компонент, определенный классом `Exercise`, мы знаем, что его можно добавить в компонент `<wkout-plan>`.

Тем не менее строка `this.dom.plan.add(exercise);` пока еще ничего не делает. Нам нужно добавить этот функционал в компонент `<wkout-plan>`. Для этого можно начать с модуля шаблона в файле plan/template.js и добавить содержимое приведенного ниже листинга.

**Листинг 10.2 Добавляем новые упражнения в компонент «план тренировок» (шаблонный модуль)**

```
mapDOM(scope) {
   return {
       exercises: scope.querySelector('#container')
   }
},
renderExercise(exercise) {
   return `<wkout-exercise class="${exercise.type}"
     ${Exercise.toAttributeString(exercise.serialize())}></wkout-exercise>`
}
```

Используем метод querySelector

Отображает каждое упражнение (возвращает строку шаблонного литерала)

И снова мы используем метод `mapDOM` для сохранения ссылки на элемент. На этот раз все наши упражнения должны содержаться в контейнере, когда мы добавляем их в план тренировки. Метод `renderExercise` просто создает новый компонент `<wkout-exercise>`, как мы делали в компоненте библиотеки ранее. Однако на этот раз источником данных является еще один компонент `<wkout-exercise>`, из которого мы копируем атрибуты. Это делается в улучшенном классе в файле plan/plan.js и отражено в приведенном ниже листинге.

**Листинг 10.3 Добавляем новые упражнения в компонент «план тренировок» (компонентный модуль)**

```
import Template from './template.js';

export default class Plan extends HTMLElement {
   constructor() {
       super();
       this.attachShadow({mode: 'open'});
       this.shadowRoot.innerHTML = Template.render();
       this.dom = Template.mapDOM(this.shadowRoot);
   }

   add(exercise) {
       this.dom.exercises.innerHTML += Template.renderExercise(exercise);
   }
}

if (!customElements.get('wkout-plan')) {
   customElements.define('wkout-plan', Plan);
}
```

❶ Сохраняем ссылку на элемент в объект для последующего использования

❷ Добавляем каждое упражнение в наш элемент контейнера списка

В этом обновленном классе мы добавили только две строки: первая – для того, чтобы получить объект, содержащий ссылки на наши элементы, как видно из примечания ❶ в листинге 10.3. Вторая – чрезвычайно

простая функция `add`, показанная в примечании ❷, которая добавляет новое упражнение к свойству `innerHTML` нашего контейнера, вначале визуализируя HTML-код из шаблонного модуля. После всего этого, обновив наш браузер и щелкнув мышью на упражнении Jump Rope (Прыжки со скакалкой) из библиотеки, наше приложение теперь выглядит как на рис. 10.2.

## 10.1.2 Контекстные стили

Пока все идет нормально! Сейчас мы показываем библиотеку упражнений и позволяем пользователю кликать на упражнения и добавлять их в личный план тренировок. Проблема, однако, состоит в том, что на самом деле нам нужно показывать личный план тренировок в виде списка. Мы также хотим, чтобы пользователи могли настраивать продолжительность или количество повторений/наборов, связанных с выбранным упражнением.

Не было бы большим преувеличением считать, что в этих двух разных контекстах использование компонента упражнения слишком разное, и нам следует создать два разных компонента для каждого контекста, что было бы обидно; в конечном итоге мы продублируем значительный объем кода, в особенности код, не связанный с визуальным оформлением. Мы также можем использовать эту возможность для изучения селектора `:host-context()`.

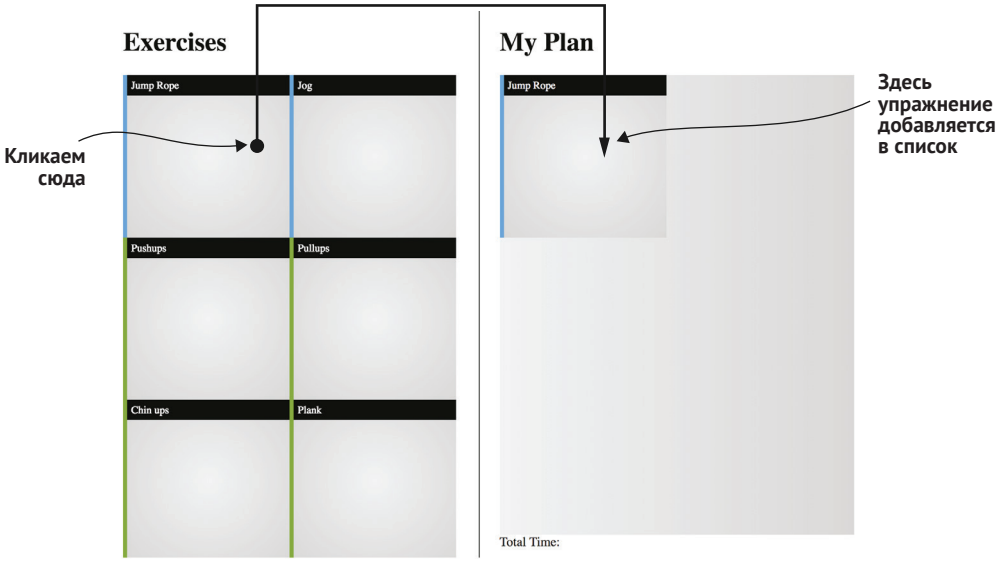



**Рис. 10.2 Состояние приложения после добавления упражнений в план тренировок**

Давайте сначала еще немного округлим HTML-код в файле exercise/template.js с помощью полей для ввода текста, чтобы поддержать идею настройки вашего упражнения:

**Листинг 10.4 Добавление дополнительных функций в компонент упражнения для представления плана**

```
html(exercise) {
   return `<div id="info">
               <span id="label">
                     ${exercise.label}  <-- Название упражнения
               </span>
               <div id="customize">
                  <label
                       class="${exercise.time?'visible':'hidden'}">
                       <input type="number" max="999" style="width:35px;"
                       value="${Number(exercise.time)}"> seconds
                  </label>
                  <label class="${exercise.count?'visible':'hidden'}">
                       <input type="number" max="99" style="width:25px;"
                       value="${Number(exercise.count)}">
                  </label>
                  <label class="${exercise.sets?'visible':'hidden'}">x
                       <input type="number" max="9" style="width:20px;"
                       value="${Number(exercise.sets)}"> set(s)
                  </label>
               </div>
               <span id="delete">x</span>  <-- Кнопка Delete
            </div>`;
},
```

Проверяем, существует ли свойство, и если нет, скрываем теги <label> и <input>

Мы оставили элементы с идентификаторами `"info"` и `"label"` такими же, как и раньше. Они не будут меняться. Однако мы добавили контейнер `<div id="customize">`, где содержится несколько элементов `<input>`, а также кликабельный тег `<span id="delete">x</span>`, чтобы в конечном итоге удалить этот элемент из нашего плана тренировок.

У каждого поля `<input>` есть тег `<label>`, который в зависимости от того, существует ли свойство в упражнении, будет иметь CSS-правило, где установлены свойства `visible` или `hidden`. Теперь, после того как мы добавили эту разметку, компонент выглядит довольно неопрятно независимо от того, в каком контексте он находится. Это можно исправить, и для этого мы будем использовать селектор `:host-context`! Этот селектор позволяет нам определять различные правила CSS, которые учитывают расположение компонента на странице, как показано на рис. 10.3.

В качестве примера давайте посмотрим на HTML-код, который мы только что добавили. Теги `<div>` и `<span>` не должны быть видны, когда компонент `<wkout-exercise>` находится в компоненте `<wkoutexercise-library>`. Поэтому мы можем добавить эти стили:

```
:host-context(wkout-exercise-lib) #customize {
   display: none;
}

:host-context(wkout-exercise-lib) #delete {
   display: none;
}
```

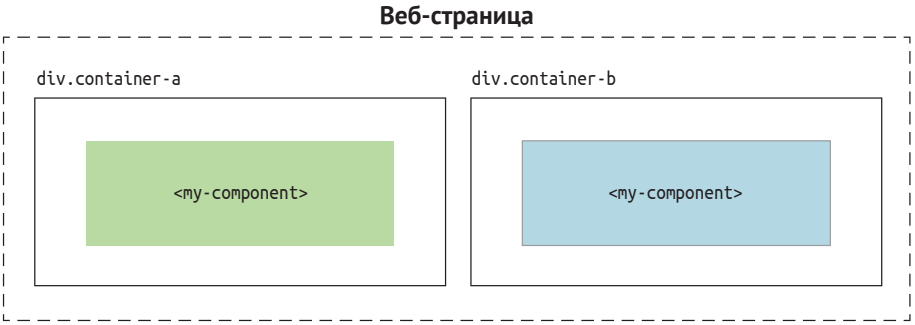



**Рис. 10.3 При использовании селектора :host-context стили по-разному зависят от контекста компонента**

Мы даже можем начать стилизацию теневого хоста компонента, используя новые правила определения размера, когда он появляется в компоненте <wkout-plan>:

```
:host-context(wkout-plan) {
   width: 100%;
   height: 50px;
   margin-bottom: 1px;
   background-size: contain;
}
```

Здесь вместо компонента в виде квадрата размером 200×200 пикселей мы сообщаем, что в контексте элемента <wkout-plan> мы хотим, чтобы ширина и высота составляли 100 % и 50 пикселей соответственно.

Теперь, имея возможность применять стили в соответствии с контекстом нашего компонента, мы можем создать набор общих CSS-правил, набор правил, когда мы находимся в компоненте <wkout-plan>, и набор правил, когда находимся в компоненте <wkout-exercise-lib>, как показано в приведенном ниже листинге.

**Листинг 10.5 Контекстные стили для компонента упражнения**

```
<style>
    :host {  <-- Стили компонента не зависят от контекста
        display: inline-block;
        background: radial-gradient(circle, rgba(235,235,235,1) 0%,
        rgba(208,208,208,1) 100%);
        background-image: url('${exercise.thumb}');
        border-left-style: solid;
        border-left-width: 5px;
    }

    :host(.cardio) {
        border-left-color: #28a7ff;
    }
```

```
:host(.strength) {
    border-left-color: #75af01;
}

#info {
    font-size: small;
    background-color: black;
    color: white;
    display: flex;
    align-items: center;
}

:host-context(wkout-exercise-lib) {
    width: 200px;
    height: 200px;
    background-size: cover;
}

:host-context(wkout-exercise-lib) #info {
    padding: 5px;
}

:host-context(wkout-exercise-lib) #customize {
    display: none;
}

:host-context(wkout-exercise-lib) #delete {
    display: none;
}

:host-context(wkout-plan) {
    width: 100%;
    height: 50px;
    margin-bottom: 1px;
    background-size: contain;
}

:host-context(wkout-plan) input {
    background-color: #505050;
    padding: 5px;
    color: white;
    border: none;
}

:host-context(wkout-plan) #delete {
    width: 30px;
    height: 100%;
    line-height: 50px;
    font-size: 12px;
    font-family: Arial;
    text-align: center;
    background-color: #404040;
    cursor: pointer;
}

:host-context(wkout-plan) #delete:hover {
```

**Стили компонента в контексте библиотеки упражнений**

**Стили компонента в контексте списка плана тренировок**

```
        background-color: #797979;
    }

    :host-context(wkout-plan) #info {
        width: calc(100% - 80px);
        height: 100%;
        margin-left: 75px;
        background-size: 75px 75px;
    }

    :host-context(wkout-plan) #customize {
        display: inline-block;
        flex: 1;
    }

    :host-context(wkout-plan) #label {
        padding-left: 10px;
        font-size: 16px;
        font-weight: bold;
        display: inline-block;
        flex: 1;
    }

    :host-context(wkout-plan) label.hidden {
        display: none;
    }

</style>
```

После того как мы разобрались со стилями, наш компонент, наконец, начинает приобретать законченную форму! Нужно сделать еще кое-что, например подключить взаимодействие для полей `<input>`, кнопку **Delete** и т. д., но с точки зрения визуального стиля и использования компонента упражнения в разных контекстах мы преуспели! В приложении, показанном на рис. 10.4, не хватает лишь пары мелких деталей: типа шрифта и его цвета.

### 10.1.3 Обходной путь

Что мне действительно нравится в селекторе `:host-context()`, так это то, что он заставляет нас задуматься над тем, как использовать один и тот же компонент самыми разными способами. Но, в конце концов, нам на самом деле не нужен селектор, чтобы достичь уровня другого макета или стиля. Конечно, чтобы добиться того же самого, нужно немного больше работы, но, учитывая состояние этого селектора, которое не поддерживается никем, кроме Chrome, вероятно, разумно сделать вид, что его не существует. Что же делать?

Один из вариантов – просто вернуться к использованию селектора `:host()` с той же целью и добавить контекст в качестве класса для самого компонента. Чтобы было понятнее, мы вернемся к шаблонному модулю компонента библиотеки упражнений из файла component/exerciselibrary/template.js.

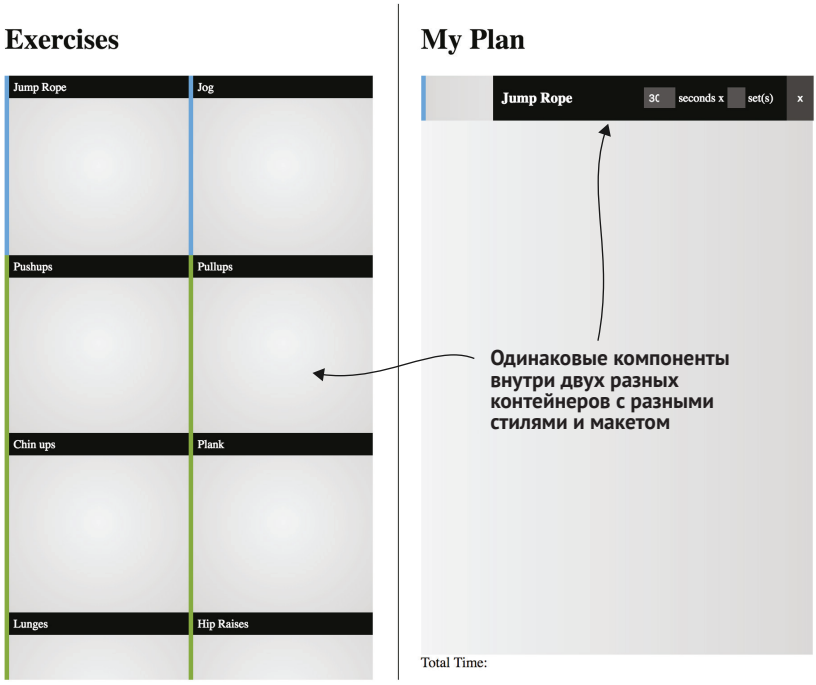



**Рис. 10.4 Компонент упражнения имеет другой стиль, макет и элементы управления в разных контекстах**

При отображении нашего HTML-кода давайте добавим еще один класс, как показано в приведенном ниже листинге.

**Листинг 10.6 Добавление контекста с помощью имени класса**

```
html(exercises) {
    let mkup = `<h1>Exercises</h1>
                <div id="container">`;
    for (let c = 0; c < exercises.length; c++) {
        mkup += `<wkout-exercise
class="${exercises[c].type} library"
${Exercise.toAttributeString(exercises[c])}></wkout-exercise>`;
    }
    Return
```

Добавляем класс library, чтобы указать на то, что компонент находится в библиотеке упражнений

Давайте сделаем то же самое с нашим планом тренировок из файла components/plan/template.js. Напомним, что, добавляя упражнение в наш план, мы отображаем некий HTML-код для каждого из них. Мы добавим класс `plan`, как показано в приведенном ниже листинге.

**Листинг 10.7 Добавляем другой контекст в план тренировок**

```
renderExercise(exercise) {
    return `<wkout-exercise
class="${exercise.type} plan"
```

Добавляем класс plan, чтобы указать на то, что компонент находится в плане тренировок

```
${Exercise.toAttributeString(exercise.serialize())}>
</wkout-exercise>`
},
```

Теперь давайте вернемся к компоненту тренировки из файла components/exercise/template.js, чтобы избавиться от всех селекторов `:host-context()`. Мы сделаем это в приведенном ниже листинге, но покажем лишь несколько измененных CSS-правил, чтобы не повторять всю таблицу стилей.

**Листинг 10.8 Меняем селекторы `host-context` на селекторы `host`**

```
:host(.library) #customize {
    display: none;
}

:host(.library) #delete {
    display: none;
}

:host(.plan) {
    width: 100%;
    height: 50px;
    margin-bottom: 1px;
    background-size: contain;
}

:host(.plan) input {
    background-color: #505050;
    padding: 5px;
    color: white;
    border: none;
}
```

Мы поменяли два селектора, чтобы использовать класс library непосредственно в компоненте

То же самое для класса plan

Конечно, в этом примере было легко добавить класс `.library` или `.plan` к компонентам, чтобы дать им другой контекст. Это становится немного сложнее, если нам нужно заменить селектор `:host-context()`, когда контекст, который мы хотим использовать, является родителем родителя родителя компонента. В этом случае вы просто не можете зайти так далеко, как этот селектор. Итак, вам нужно получить какой-то сигнал для вашего компонента через иерархию DOM, чтобы добавить желаемый класс, или, возможно, просто создать какой-нибудь API (может быть, сеттер, атрибут или и то, и другое) в вашем приложении для переключения режима.

## 10.2 Темы компонента

Как правило, когда мы говорим о темизации веб-приложения, речь идет об использовании стилей CSS, которые проникают во все аспекты нашего пользовательского интерфейса. В последней главе мы обсуждали кнопку, предлагаемую дизайн-системой.

Теперь возвращаемся на круги своя. Когда мы обсуждали эту кнопку ранее, вопрос заключался в том, как не дать дизайн-системе и правилам

CSS, стоящим за ней, испортить внешний вид кнопок в нашем счетчике, как показано на рис. 10.5.

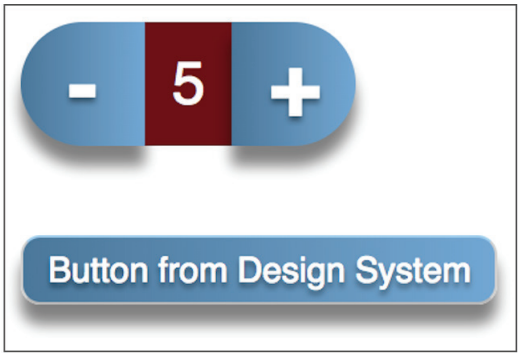



**Рис. 10.5 Рисунок из главы 9, напоминающий об опасности утечки стилей в наш компонент**

Теперь, когда мы используем Shadow DOM и успешно блокируем эти стили, возникает вопрос: «Что, если мы действительно *хотим*, чтобы эти стили появлялись?» Ответ не так прост.

## 10.2.1 Селекторы ::shadow и /deep/

> **ПРЕДУПРЕЖДЕНИЕ** В этом разделе обсуждаются устаревшие функции. Селекторы `::shadow` и `/deep/` основательно устарели, в отличие от селектора `:host-context()` из раздела 10.1. Хотя, возможно, это может немного сбивать с толку, потому что они устарели сравнительно недавно. В результате вы можете пойти по неверному пути. К сожалению, подходящей замены им нет, поэтому, даже если вам не нужны устаревшие функции, здесь мы обсудим несколько вариантов.

Я вас предупредил – то, что мы здесь рассматриваем, устарело, но немного истории будет полезно.

Чтобы решить проблему применения стилей к содержимому внутри корня теневого дерева, когда нам это действительно нужно, раньше использовалось два CSS-селектора: `::shadow` и `/deep/`. Эти селекторы были разработаны для преодоления границы Shadow DOM в первой версии Shadow DOM (v0). Они так и не попали в версию 1 и были окончательно удалены из Chrome в версии 63.

Селектор `::shadow` погружался в Shadow DOM и стилизовал все, что было внутри.

Например, с помощью этого

```
::shadow .example {
       color: red;
}
```

мы бы сделали текст шрифта в теге `<div>` внутри приведенного ниже компонента (при условии использования Shadow DOM) красным:

```
<my-component>
    <div class="example">Some red text</div>
</my-component>
```

Однако если бы вы также использовали div-элемент с классом `example` где-то за пределами Shadow DOM, правило не было бы применено:

```
<div class="example">This text is not red</div>
<my-component>
    <div class="example">Some red text</div>
</my-component>
```

Чтобы стилизовать оба варианта, независимо от того, появляются они внутри Shadow DOM или нет, можно использовать селектор `/deep/`:

```
/deep/ .example {
        color: red;
}
```

Когда в Chrome отказались от поддержки этих селекторов, по известным причинам никто не понимал, что делать дальше. Причины отказа, безусловно, благие. Эти селекторы в основном являются своего рода лейкопластырем – они позволяют вам вернуться к применению собственных CSS-правил для инкапсулированного компонента. Сам компонент теряет контроль над собственными стилями, как это было до появления Shadow DOM.

Одним из предлагаемых в настоящее время решений является проект Shadow Parts рабочей группы по разработке спецификаций каскадных таблиц стилей CSS (CSSWG). Здесь я не буду вдаваться в подробности, потому что он был представлен только в Chrome, вместе со связанным с ним селектором `::theme` и, вероятно, будет развиваться. Тем не менее если вы хотите следить за этим проектом, можете делать это на странице https://drafts.csswg.org/css-shadow-parts/.

Я хотел бы процитировать один отрывок:

> *Важно отметить, что селектор ::part() предлагает абсолютно нулевую теоретическую мощность. Это не переделывание комбинатора >>> на новый лад, это просто более удобный и согласованный синтаксис для того, что авторы уже могут делать с пользовательскими свойствами.*

Данная цитата заслуживает внимания, поскольку текущая рекомендация касательно использования пользовательских свойств в настоящее время является *единственным* способом проникновения в Shadow DOM. Эти пользовательские свойства, также известные как CSS-переменные, могут сочетаться с селекторами `::part` и `::theme`, что приводит к совершенно новым способам управления CSS, преодолевая устаревшие функции, упомянутые в данной главе.

Это пространство, за которым вы наверняка захотите следить, потому что, как только оно получит признание, я верю, что оно даст новые способы сделать ваши компоненты стилизуемыми и проложит путь для

дизайн-систем в будущем. Однако сейчас еще слишком рано строить какие-либо прогнозы относительно того, как все это будет развиваться.

### 10.2.2 CSS-переменные

Если вы пропустили последний раздел из-за обсуждения устаревших функций, то мы обсуждаем небольшую проблему. В настоящее время не существует подходящего механизма передачи правил CSS в ваш компонент, если это действительно то, что вам нужно. Лучшее, что у нас есть, – это CSS-переменные.

Возможно, вы уже знакомы с CSS-переменными или пользовательскими свойствами. Фактически они поддерживаются всеми современными браузерами (извините, для IE нужен полифил) и не имеют ничего общего с Shadow DOM, кроме того факта, что могут пересекать теневую границу.

Вы, вероятно, представляете себе, что это такое. Они позволяют вам определять переменную, которая обозначает некое CSS-свойство, и использовать эту переменную еще где-то в одном или нескольких местах. Взять, к примеру, этот листинг.

**Листинг 10.9 Использование CSS-переменных**

```
<head>
<style>
   body {
       --text-color: blue;
   }

   .container {
       --text-color: red;
   }

   .child {
       color: var(--text-color);
   }
</style>

</head>
<body>
   <div class="container">
       <div class="child">Some Text</div>
   </div>
</body>
```

Переменная цвета, определенная для тела страницы

Более конкретная переменная цвета, определенная для CSS-класса

Использование переменной цвета

Мы можем определить переменную, относящуюся к классу `container`, объявив, что переменная `text-color` имеет красный цвет. Переменные также имеют понятие наследования. Это означает, что у нас может быть та же самая переменная `text-color`, определенная для чего-то менее конкретного, например `body`. Переменная класса `container` по-прежнему имеет приоритет, но если ее удалить, наша переменная для `body` сработает, и цвет текста теперь будет синим. Это поведение можно увидеть на рис. 10.6.

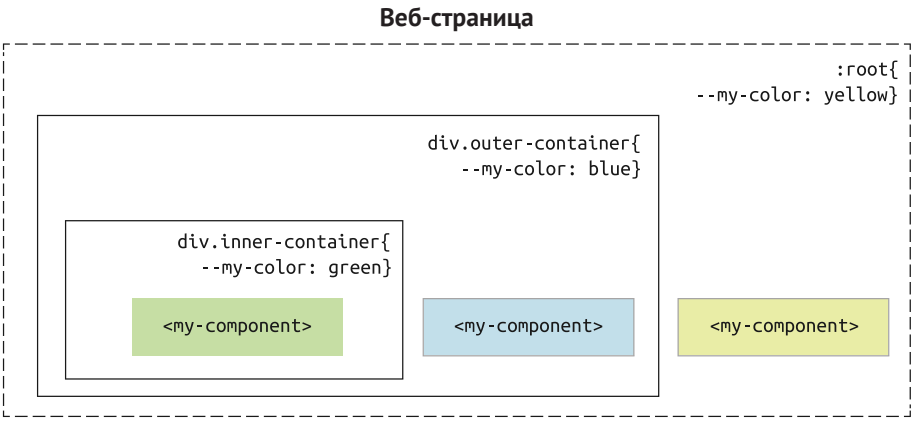



**Рис. 10.6 Использование одной и той же CSS-переменной на нескольких уровнях в DOM**

Это прекрасно работает и в Shadow DOM! В приведенном ниже листинге показано правило, определяемое пользовательским компонентом снаружи. Здесь мы наблюдаем прохождение мимо теневой границы в компонент.

**Листинг 10.10 Использование CSS-переменных в теневой модели DOM веб-компонента**

```
<style>
    sample-component {
        --text-color: blue;
    }

</style>
<script>
    class SampleComponent extends HTMLElement {
        connectedCallback() {
            this.attachShadow({mode: 'open'});
            this.shadowRoot.innerHTML = `<div class="inside-component">
                My Component
            </div>
            <style>
                .inside-component {
                    color: var(--text-color);
                    }
            </style>`
        }
    }

    if (!customElements.get('sample-component')) {
        customElements.define('sample-component', SampleComponent);
    }
```

Объявляем CSS-переменную цвета текста

Используем CSS-переменную цвета текста

```
    </script>
</head>
<body>
<sample-component></sample-component>
</body>
```

CSS-переменные не обязательно должны находиться в локальной области видимости – они могут становиться глобальными с помощью псевдоселектора `:root {...}`, который фактически имеет даже более низкую специфичность, чем `html { . . . }`. Поэтому, когда нам не нужно беспокоиться по поводу специфичности или мы просто устанавливаем эталон переменных, можно совершено спокойно использовать селектор `:root`.

## 10.2.3 Применяем CSS-переменные в нашем примере

Давайте начнем с простой темы на нашем примере с Workout Creator! Сейчас мы можем сосредоточиться только на нескольких переменных, но, основываясь на этом, можно легко представить себе более крупную дизайн-систему.

Во-первых, я хотел бы использовать другой шрифт для нашего приложения, поэтому мы должны изменить тег `<head>` в нашем файле index.html, чтобы загрузить шрифт, как показано в приведенном ниже листинге.

**Листинг 10.11 Загрузка шрифта**

```
<head>
   <title>Workout Creator</title>
   <script type="module"
   src="components/workoutcreatorapp/workoutcreatorapp.js"></script>
   <link rel="stylesheet" type="text/css" href="main.css">
   <link href=
    "https://fonts.googleapis.com/css?family=Roboto+Slab" rel="stylesheet">
</head>
```

Ссылка на шрифт

Во-вторых, мы можем определить глобальные переменные внутри файла main.css.

**Листинг 10.12 Определение глобальных переменных в CSS для цвета и размера текста**

```
:root {
   --inverted-text-color: #eaeaea;
   --text-color: #3a3a3a;
   --label-color: #2a2a2a;
   --header-font-size: 21px;
   --font: 'Roboto Slab', serif;
}
```

Переменные находятся в глобальной области видимости: root и определяют некоторые ключевые свойства, которые будут использоваться повсеместно

Наконец, мы можем обновить стили каждого компонента, как видно из этого листинга (показаны только правила, которые изменились).

**Листинг 10.13 Разрешаем легкую темизацию в Workout Creator с помощью CSS-переменных**

```
// exercise/template.js
:host {
   font-family: var(--font);
   display: inline-block;
   background: radial-gradient(circle, rgba(235,235,235,1) 0%,
   ➥rgba(208,208,208,1) 100%);
   background-image: url('${exercise.thumb}');
   border-left-style: solid;
   border-left-width: 5px;
}

#info {
   font-size: small;
   background-color: var(--label-color);
   color: var(--inverted-text-color);
   display: flex;
   align-items: center;
}

// exerciselibrary/template.js
:host {
   display: inline-block;
   font-family: var(--font);
   color: var(--text-color);
}

h1 {
   font-size: var(--header-font-size);
}
```

Темизация с использованием CSS-переменных в файле exercise/template.js

Обновление каждого компонента продолжается до следующего листинга, где CSS-переменные добавляются к предыдущему.

**Листинг 10.14 Облегченная темизация с помощью CSS-переменных в компоненте Workout plan**

```
// plan/template.js
:host {
   display: flex;
   flex-direction: column;
   font-family: var(--font);
   color: var(--text-color);
}

h1 {
   font-size: var(--header-font-size);
}

// workoutcreatorapp/template.js
#divider-line {
   width: 1px;
   height: 100%;
```

Темизация с использованием CSS-переменных для файла plan/template.js

```
   margin-right: 25px;
   background-color: var(--text-color);
}
```

После этих изменений стилей текста мы можем дополнительно настроить CSS-переменные нашей простой темы. Например, можно было бы выбрать цвет позеленее, как показано в следующем листинге, используемом для темизации нашего приложения. Результат можно увидеть на рис. 10.7.

**Листинг 10.15 Изменение переменных для перехода на другой цвет**

```
:root {
   --inverted-text-color: #daf8a1;
   --text-color: #47730c;
   --label-color: #59b624;
   --header-font-size: 18px;
   --font: 'Roboto Slab', serif;
}
```

Два CSS-правила для изменения исходных оттенков серого на зеленый цвет

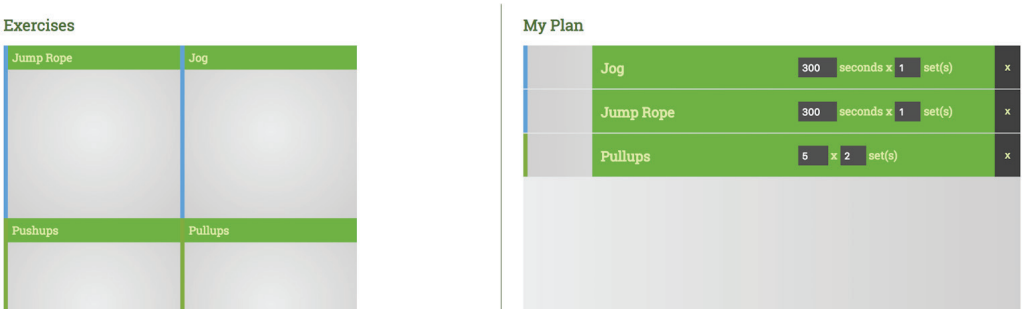



**Рис. 10.7 Простое изменение CSS-переменных на уровне корня для изменения цвета темы с черного на зеленый**

## 10.3 Использование теневой модели DOM на практике (сегодня)

Не так давно я подходил к использованию теневой модели DOM с осторожностью – настолько, что рекомендовал не использовать ее совсем и разрабатывать свои веб-компоненты без нее. С тех пор произошли удивительные вещи, и мы наконец-то получили действительно хорошую возможность их использовать. Конечно, прежде чем идти ва-банк, нужно учесть некоторые вещи!

### 10.3.1 Поддержка со стороны браузеров

Это главная и самая очевидная проблема. Поначалу теневая модель DOM поддерживалась только браузером Chrome, а затем к нему присоединился Safari. На протяжении некоторого времени это все, что у нас было! Затем,

в октябре 2018 года, Firefox выпустил веб-компоненты, которые, конечно же, включают в себя Shadow DOM. Мы знали, что Microsoft Edge работает над веб-компонентами (https://developer.microsoft.com/en-us/microsoft-edge/platform/status/shadowdom). Я очень ценю приведенное здесь упоминание о том, что Shadow DOM будет поддерживаться в семействах устройства XBOX, мобильных устройствах и устройствах смешанной реальности. Когда же это произойдет? Сейчас готовится предварительная версия для разработчиков, поэтому, возможно, мы увидим подходящий выпуск как раз к тому времени, когда будет опубликована эта книга! Конечно, это только предположение, но тем не менее похоже, что все последние версии современных браузеров (будут) иметь поддержку.

Поэтому, разумеется, нужно просто идти в ногу с Edge, прежде чем у всех пользователей появится возможность выполнить обновление и получить новую версию с поддержкой Chromium. Если ваш проект нуждается в его поддержке и вы не хотите играть на дату поставки Microsoft Shadow DOM, то на данный момент вы можете создавать свои веб-компоненты без Shadow DOM.

## 10.3.2 Полизаполнение

Конечно, в предыдущем разделе я говорил о нативной поддержке. Чаще всего можно с легкостью использовать полизаполнение. Это также относится и к веб-компонентам, за исключением CSS-реализации Shadow DOM.

Проблема здесь состоит в том, что мини-инкапсулированную модель DOM трудно эмулировать. Защитить DOM от использования метода `querySelctor` и манипулирования легче, чем от просачивания CSS-правил. В первом случае можно использовать полифил ShadyDOM. Во втором случае, когда речь идет о CSS, – полифил ShadyCSS.

К сожалению, в то время как в большинстве случаев полифил можно просто вставить и все будет готово, в случае с ShadyCSS необходима ручная фиксация. Тем не менее последние разработки усилили мой оптимизм.

Полизаполнение ShadyCSS было создано компанией Google, чтобы использовать его с библиотекой Polymer для создания веб-компонентов. Нужно отдать Google должное: компания сделала все возможное, чтобы не запирать эти полизаполнения в библиотеке. Все их можно найти на странице https://www.webcomponents.org и на GitHub по адресу https://github.com/webcomponents.

В случае с ShadyCSS полизаполнение было слишком привязано к библиотеке Polymer, которая в значительной степени была основа на ныне устаревшем рабочем процессе HTML-импорта. В результате документация по ShadyCSS и предлагаемый рабочий процесс были очень сфокусированы на том, как использовать полизаполнение с помощью шаблона, который нужно было реализовать с помощью HTML-импорта.

С тех пор команда Polymer прекратила разработку функциональных средств для библиотеки Polymer и теперь рекомендует более мелкие

и целевые библиотеки, которые были написаны и заброшены, и все это под эгидой проекта Polymer. Мы уже обсуждали lit-HTML в качестве одной из них. Еще один пример – проект LitElement, https://litelement.polymer.project.org, который использует lit-html за кулисами. К счастью для нас, этот новый рабочий процесс веб-компонента с LitElement, который они предлагают, работает точно так же, как мы уже видели в этой книге: шаблонные литералы в вашем JS-коде для управления HTML-разметкой и стилями CSS – все это управляется с помощью простого API жизненного цикла веб-компонентов. В результате их использование ShadyDOM совпадает с нашим. Там, где раньше нам нужен был шаблон для нашего HTML-кода, он по-прежнему необходим нам, но обо всем этом автоматически заботится lit-html. То, что мы поместили бы в корень теневого дерева, теперь оборачивается в качестве шаблона. В качестве шаблона ShadyCSS затем переписывает наши неспецифичные CSS-правила в нечто более уникальное, которое действует как инкапсулированная теневая модель DOM. Затем ShadyCSS добавляет этот шаблон в качестве дочернего элемента нашего компонента.

В конечном результате наши компоненты с поддержкой Shadow DOM работают во всех популярных браузерах (да, даже в IE). У ShadyCSS есть список известных проблем, о которых следует знать, поэтому не ждите, что все будет работать: https://github.com/webcomponents/shadycss. В следующей главе мы создадим готовый к бою компонент с учетом этих проблем, используя полизаполения, где это уместно.

### 10.3.3 Дизайн-системы

Даже простое оформление вашего приложения потребует переосмысления того, как мы сегодня используем дизайн-системы. Если вы незнакомы с дизайн-системами, они, как правило, представляют собой CSS-библиотеку для обеспечения единообразного просмотра вашего проекта, в том числе для часто используемых элементов пользовательского интерфейса, таких как кнопки, ползунки, контейнеры содержимого и т. д. Одна из самых популярных дизайн-систем, например Bootstrap, просто не будет работать с Shadow DOM как есть.

Сами дизайн-системы обычно довольно монолитные, то есть вы, вероятно, включите один основной (и огромный) CSS-файл в свою главную страницу. Этот файл будет проходить через вашу страницу и стилизовать все ваши элементы, как и должно быть.

Проблема, конечно же, состоит в том, что теневая модель DOM заблокирует все стили, определенные для вашей страницы, защищая все элементы, вложенные в компонент. Хорошая новость заключается в том, что дизайн-система, скорее всего, будет состоять из отдельных компонентов. Снэпшот компонентов Bootstrap показан на рис. 10.8.

В конце концов, все эти CSS-компоненты встраиваются в этот монолитный CSS-файл, но в дизайн-системах, подобных этим, существует возможность выделить каждый компонент в качестве собственных импортируемых стилей, которые будут использоваться именно тем веб-

компонентом, которому они нужны. Все становится немного сложнее, потому что в примере с Bootstrap карусель может зависеть от кнопки, которая зависит от базовых стилей текста, пробелов и цвета. Таким образом, необходимо управлять зависимостями, чтобы все получилось.

Тем не менее некоторые трудолюбивые люди уже приступили к созданию разновидностей веб-компонентов для таких популярных дизайн-систем, как Bootstrap. То же самое мы уже наблюдали в случае с Angular и React. Уже запущен процесс разработки разновидностей веб-компонентов для Marerial (https://github.com/material-components/material-componentsweb-components). Наверняка здесь будет предпринято множество усилий, учитывая, что команда Polymer перешла от библиотеки Polymer на более универсальные библиотеки веб-компонентов.

Проблемы определенно будут. Во многих случаях, когда речь идет о дизайн-системах, вы будете сталкиваться с тем, что стиль элемента зависит от контекста того, кто является его родительским элементом или идет дальше по цепочке предков. Конечно, здесь будет полезен селектор `:host-context`, но вместо этого просто следует быть осторожными при создании или преобразовании существующей дизайн-системы.

Конечно, пока делаются только первые шаги в использовании дизайн-систем в веб-компонентах, но уже виден существенный прогресс. По мере продвижения вперед я ожидаю увидеть гораздо более частое использование CSS-переменных и, возможно, даже CSS Shadow Parts, как только это предложение будет утверждено и будет поддерживаться браузерами.

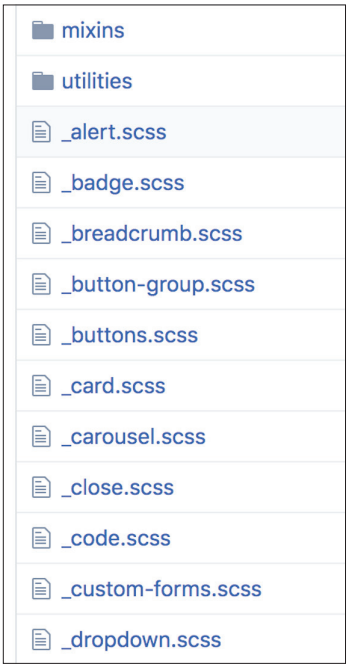



**Рис. 10.8 Bootstrap состоит из этих и многих других компонентов**

## Резюме

В этой главе мы увидели множество преимуществ, которые дает теневая модель DOM в плане борьбы со стилями. Но, с другой стороны, поскольку он еще поддерживается не полностью и никогда не будет поддерживаться в IE, мы знаем, что следует действовать с осторожностью. Продвигаясь вперед по мере обновления браузеров, мы берем на вооружение неплохие основополагающие приемы.

Из этой главы вы также узнали:

- как остерегаться селекторов, которые уже устарели, по мере стандартизации спецификации Shadow DOM;
- что к теневой модели DOM не так просто применить полизаполнение при использовании стилей. Поэтому если вы используете ее с CSS, будьте осторожны и следите за поддержкой со стороны браузеров, обращайте внимание на полизаполнение и нереализованные функции;
- дизайн-системы двигаются вперед в направлении поддержки теневой модели DOM, но на это потребуется время. Хотя сейчас и существует несколько проектов, разработанных для Shadow DOM, вам, возможно, придется закатать рукава и самостоятельно поработать над адаптацией.

# Часть III

# Объединяем компоненты воедино

В последней части этой книги мы возьмем все, что изучили, и убедимся, что наши компоненты готовы к промышленной эксплуатации, протестированы и работают вместе как единое целое – везде.

Начиная со второй главы мы создавали и совершенствовали очень простой компонент. В первой главе третьей части мы снова вернемся к этому компоненту, чтобы создать нечто большее, используя его в качестве более мелкого вложенного фрагмента более крупного веб-компонента. Мы опробуем этот новый компонент, познакомившись с процессом сборки и тестирования, и, наконец, воспользуемся рефакторингом, чтобы убедиться, что он работает во всех версиях браузеров вплоть до IE11.

Организация взаимодействия и обмена сообщениями в приложении тоже не простая задача, и как таковая она очень важна для диалога, особенно при принятии решения об использовании веб-компонентов вместо основного фреймворка. В этой части обсуждаются соответствующие стратегии, прежде чем окончательно остановиться на новых технологиях, в которых могут быть полезны веб-компоненты, такие как Web 3D, смешанная реальность и машинное обучение, и будут даны простые примеры для изучения этих футуристических понятий.

# 11

# Реальный компонент пользовательского интерфейса

***Эта глава охватывает следующие темы:***

- мы создадим новый компонент и освежим в памяти полученные знания, включая теневую модель DOM, модули, API пользовательского элемента и рефлексию;
- разобьем дизайн на несколько компонентов и сосредоточимся на возможности повторного применения компонента, чтобы использовать один, менее крупный компонент двойного назначения в более широком контексте;
- воспользуемся объектом `MutationObserver` для отслеживания изменений атрибутов;
- познакомимся со всплыванием событий внутри вашего компонента;
- изучим передовые практики работы с компонентами, связанные с использованием теневой модели DOM;
- поговорим о применении согласованных и универсальных правил дизайна, предоставляемых CSS-переменными и импортируемыми CSS-модулями.

Сейчас самое время сделать шаг назад и подвести итог. В этой книге мы много сделали с веб-компонентами. В то же время у нас было не так мно-

го замечательных примеров компонентов пользовательского интерфейса, которыми можно поделиться с другими. Один из примеров компонента пользовательского интерфейса, над которым мы работали, – это ползунок.

Рисунок 11.1 вкратце напоминает о том, на что он способен. С самого начала этот компонент создавался с целью совместного использования в более широком контексте. Он прост, поэтому применение его в качестве небольшого фрагмента более крупного, полезного и общедоступного компонента пользовательского интерфейса – отличная цель для дальнейших действий!

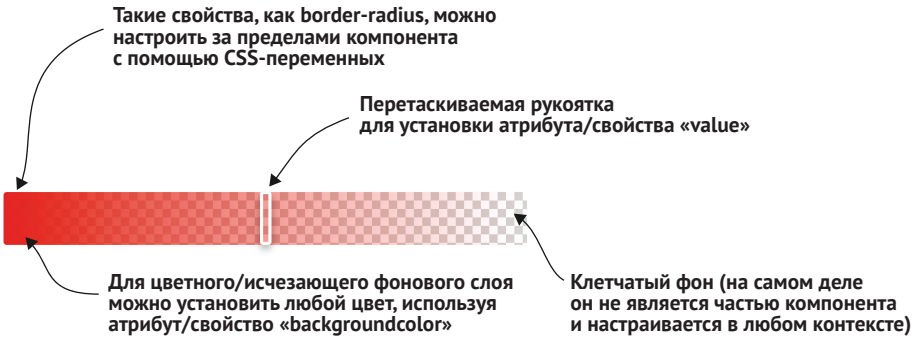



**Рис. 11.1 Компонент пользовательского интерфейса, который мы создавали на протяжении всей книги**

## 11.1 Создаем палитру цветов

Итак, чем мы займемся? Каков тот более широкий контекст, в который мы поместим наш компонент? Часто, когда мне нужен градиент в своих стилях, я использую некоторые простые, но удивительные инструменты, предлагаемые онлайн. Я думаю, что приложение для создания градиента на сайте https://cssgradient.io, показанное на рис. 11.2, хорошо спроектировано и просто в использовании. Я также считаю, что было бы прекрасно, если бы мы сократили его функциональность и убрали оттуда аспект создания градиента.

Возможно, вы знаете, что уже существует простая в использовании палитра цветов, предоставляемая тегом `<input>`. Мы можем просто добавить этот элемент в наш HTML-код:

```
<input type="color" value="#ff0000">
```

В результате мы получаем красивую, компактную, кликабельную кнопку, при нажатии на которую появляется диалоговое окно, как показано на рис. 11.3.

Как бы удобно это ни было, мне довелось работать над проектом, в котором эта палитра цветов была слишком компактной. Кроме того, мне бы хотелось, чтобы она всегда оставалась на экране, чтобы я мог постоянно настраивать объект, который раскрашиваю. Наконец, объект, кото-

рый я раскрашиваю, может быть прозрачным, поэтому было бы неплохо иметь контроль над прозрачностью в дополнение к цветам.

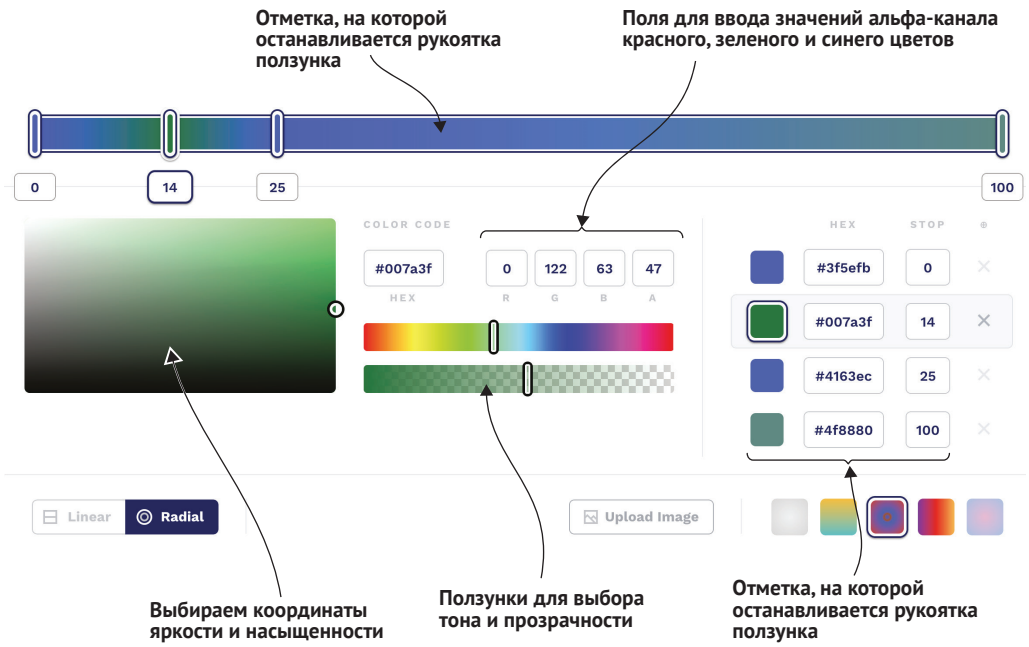



Рис. 11.2 Приложение для создания градиента на сайте cssgradient.io

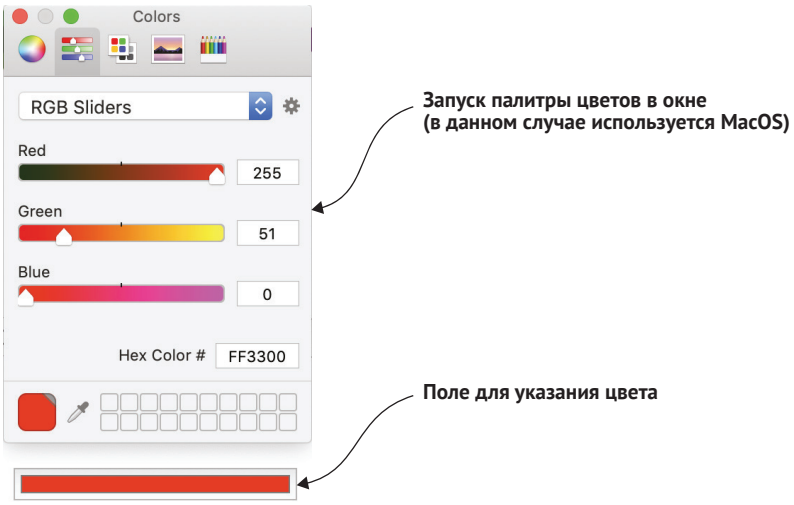



Рис. 11.3 Использование палитры цветов, созданной с помощью элемента input

Учитывая все вышесказанное, давайте создадим великолепную палитру цветов, вдохновленную сайтом cssgradient.io, с помощью веб-компонентов!

### 11.1.1 Компоненты нашего компонента

Вначале мы разобьем на части то, что нам нужно. Нам не нужно, чтобы наш компонент палитры цветов был слишком сложным, поэтому следует подумать о том, какие части пользовательского интерфейса являются его отдельными компонентами.

Если посмотреть на рис. 11.4, можно увидеть два похожих функционирующих ползунка, один для выбора тона и один для выбора прозрачности. Это замечательно, поскольку мы создавали этот простой, многократно используемый ползунок специально для выбора прозрачности! Единственное реальное различие – это изображения и цвета, используемые в качестве фона в этих двух ползунках. Напомню, что внутри компонента есть элемент `<div>`, который отображает один цвет с прозрачностью от 100 % до 0 %. Если цвет фона не установлен, этот элемент никогда не отображается, что дает нам полностью стилизуемый компонентный фон. В файле ползунка demo.html мы используем клетчатый бело-серый узор для обозначения прозрачности, но его легко можно заменить на фон с цветами радуги.

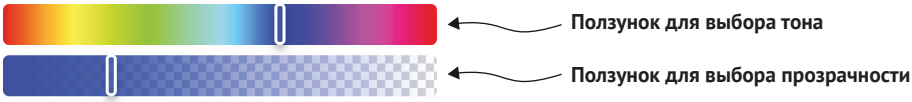



**Рис. 11.4 Два очень похожих ползунка, для которых можно создать один веб-компонент**

Здесь есть еще один элемент пользовательского интерфейса, который во многом похож на ползунок, но работает как в горизонтальном направлении, так и в вертикальном. На рис. 11.5 показан инструмент для выбора насыщенности и яркости цвета. Перетаскивая указатель слева направо, мы регулируем насыщенность или интенсивность цвета, а при перетаскивании его в вертикальном направлении регулируется яркость. Пользователь может одновременно перемещаться в обоих направлениях, выбирая идеальное сочетание двух переменных.

В отличие от ползунка, нам нужен только один из этих инструментов; но, как и в случае с ползунком, мы должны быть очень внимательны к тому, как пользователь взаимодействует с ним. Чтобы инструмент выбора и ползунок правильно реагировали на взаимодействие с пользователем, необходимо позаботиться о крайних случаях; этот двунаправленный инструмент для выбора насыщенности и яркости цвета идеально подходит для использования в качестве собственного веб-компонента, поэтому нам не нужно перегружать свой основной компонент, палитру цветов, слишком большим количеством логики.

Какие еще элементы можно выбрать? У вас могут быть собственные идеи, но я думаю, что на этом можно и остановиться. Помните, что, несмотря на то что мы ориентируемся на приложение на сайте cssgradient.io, мы не создаем градиенты, а просто выбираем цвета. Остальные соответ-

ствующие элементы пользовательского интерфейса – это поля для ввода текста и чисел. Мы, конечно, можем пойти дальше и обернуть их все или их наборы как различные компоненты.

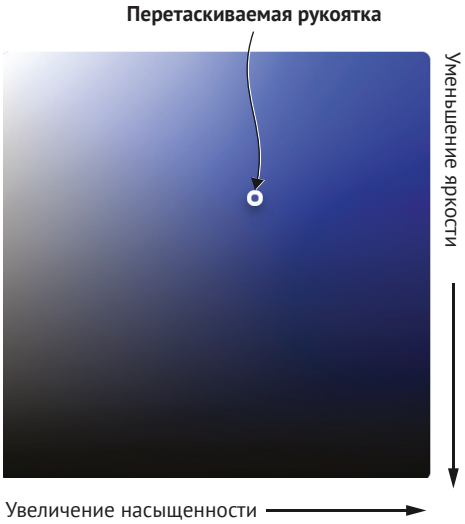



**Рис. 11.5 Инструмент для выбора насыщенности и яркости цвета как кандидат на место веб-компонента**

Например, мы могли бы сделать числовую запись для полей красного, зеленого и синего цветов одним компонентом. Я не уверен, что здесь от этого можно получить большую выгоду, особенно потому, что когда вы делаете шаг назад, каждый элемент пользовательского интерфейса имеет простой способ взаимодействия с ним и предлагает выходное значение после взаимодействия (или два, если рассматривать инструмент для выбора по горизонтали и вертикали).

Подведем итог. Вот какие части пользовательского интерфейса у нас есть:

- ползунок для выбора тона обеспечивает взаимодействие с мышью и выводит одно значение;
- ползунок для выбора прозрачности обеспечивает взаимодействие с мышью и выводит одно значение;
- инструмент для выбора насыщенности и яркости цвета обеспечивает взаимодействие с мышью и выводит два значения;
- поля для красного, зеленого, синего цветов и поля для ввода букв и цифр обеспечивают взаимодействие с клавиатурой и мышью и выводят одно значение;
- поле для ввода шестнадцатеричного кода цвета обеспечивает взаимодействие с клавиатурой и выводит одно значение.

Как мы уже говорили, мы можем объединить тон и ползунок в один, повторно используемый компонент, а также сделаем компонентом инструмент для выбора насыщенности и яркости цвета. Остальное – пользовательский интерфейс текстовой/числовой записи – будет располагаться рядом с нашими пользовательскими компонентами в главном

компоненте цветовой палитры. Учитывая этот план, давайте создадим структуру проекта, как показано на рис. 11.6.

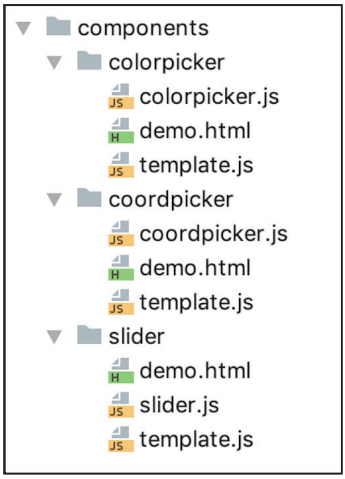



**Рис. 11.6 Начальная структура нашего проекта**

Скорее всего, по мере продвижения файлов будет больше, но можно предположить, что у каждого из этих трех компонентов – ползунка, инструмента выбора координат и основной палитры цветов – должно быть по три файла. Нам понадобится класс для определения компонента, шаблон, в котором будет содержаться весь HTML-код и стили, и, наконец, HTML-файл для демонстрации работы автономного компонента. Поскольку мы работали над ползунком на протяжении всей книги, у меня есть для вас хорошая новость: все готово! Код можно скопировать прямо сейчас в папку компонентов.

## 11.2 Компонент выбора координат

Давайте кратко рассмотрим компонент выбора координат. Причина, по которой мы не делаем это подробно, заключается в том, что, помимо того что это довольно простой компонент, он функционирует почти так же, как и ползунок. Единственное отличие состоит в том, что рукоятку можно перетаскивать как вертикально, так и горизонтально, как показано на рис. 11.5.

Я называю это инструментом выбора координат, потому что вы перетаскиваете и выбираете что-то по осям *x* (в горизонтальном направлении) и *y* (в вертикальном направлении). В частности, мы будем использовать ось *y*, чтобы в конечном итоге управлять яркостью в главном компоненте, палитре цветов, в то время как насыщенность контролируется осью *x*.

### 11.2.1 Класс инструмента выбора координат

Учитывая их сходство, давайте поговорим об одном различии между ползунком и инструментом выбора координат. Вместо атрибута `value`

теперь у нас есть два разных значения, которые используются для обозначения процентов в горизонтальном и вертикальном направлениях. Мы будем использовать имена атрибутов *x* и *y* для их обозначения. Кроме того, при обновлении визуального эскиза нам нужно принимать во внимание как вертикальное положение, так и горизонтальное. Мы будем использовать приведенный ниже листинг, чтобы выделить первую часть класса с похожими, но слегка измененными атрибутами и соответствующим отражением.

**Листинг 11.1 API инструмента выбора координат**

```
import Template from './template.js';

export default class CoordPicker extends HTMLElement {
   static get observedAttributes() {
       return ['x', 'y', 'backgroundcolor'];
   }

   attributeChangedCallback(
       name, oldVal, newValue) {
       switch (name) {
           case 'x':
           case 'y':
               this.refreshCoordinates();
               break;

           case 'backgroundcolor':
               this.style.backgroundcolor = newValue;
               break;
       }
   }

   set x(val) {
       this.setAttribute('x', val);
   }

   get x() {
       return this.getAttribute('x');
   }

// Геттеры и сеттеры для y и backgroundcolor для краткости опущены
```

Импортируем шаблонный модуль для HTML и CSS

Определяем, за какими атрибутами нужно наблюдать

Следим за изменениями этих атрибутов и отвечаем соответственно

Метод чтения и устанавливающий метод для свойств x, y и backgroundcolor

Вы, возможно, заметили, что много места занято рефлексией. Кроме того, это очень скучный, повторяющийся код, тем более если учесть, что разница между этим компонентом, последним компонентом или… любым другим компонентом очень мала. Вот почему вы, вероятно, будете все реже наблюдать явную рефлексию в компонентах, которые будете встречать на практике.

Появляются библиотеки и утилиты, стремящиеся сделать подобное отражение функцией, которую вам не нужно реализовывать самостоятельно (помимо других вещей). Этого можно добиться с помощью базового класса, который сам расширяет `HTMLElement`, или – если руководствоваться личными предпочтениями – с помощью смешанного подхода, при котором вы можете расширять свой класс, используя ключевое слово

prototype, поэтому можете свободно расширять еще один базовый класс. Здесь, однако, я думаю, что будет полезно показать все необходимое, не скрывая более скучных частей.

Еще одна сложность, которая часто скрывается при использовании последних библиотек веб-компонентов, таких как LitElement из проекта Polymer (https://lit-element.polymerproject.org/), – это первоначальная настройка компонента и HTML-рендеринг. Действительно, конструктор компонента из листинга 11.2 выглядит точно так же, как и компонент-ползунок, вплоть до слушателей событий, учитывая, что мы дублируем ту же возможность перетаскивания рукоятки. При написании большого количества собственных компонентов вы также можете попытаться скрыть эту сложность с помощью собственного базового класса или вспомогательных утилит.

**Листинг 11.2 Конструктор и обработчик событий мыши**

```
constructor() {
    super();
    this.attachShadow({mode: 'open'});
    this.shadowRoot.innerHTML = Template.render();
    this.dom = Template.mapDOM(this.shadowRoot);

    document.addEventListener('mousemove', e => this.eventHandler(e));
    document.addEventListener('mouseup', e => this.eventHandler(e));
    this.addEventListener('mousedown', e => this.eventHandler(e));
}

eventHandler(e) {
    const bounds = this.getBoundingClientRect();
    const coords = {
        x: e.clientX - bounds.left,
        y: e.clientY - bounds.top
    };
    switch (e.type) {
        case 'mousedown':
            this.isDragging = true;
            this.updateXY(coords.x, coords.y);
            this.refreshCoordinates();
            break;
        case 'mouseup':
            this.isDragging = false;
            break;
        case 'mousemove':
            if (this.isDragging) {
                this.updateXY(coords.x, coords.y);
                this.refreshCoordinates();
            }
            break;
    }
}
```

Настройка компонента: Shadow DOM, рендеринг HTML/CSS и кеширование элементов

Добавляем слушателей событий мыши для перетаскивания

Фиксируем координаты мыши относительно компонента

Обновляем атрибуты компонента x, y в процентах и перемещаем рукоятку

Оборачивая компонент, мы снова увидим очень похожий код на ползунок в приведенном ниже листинге. И снова единственное реальное

отличие заключается в обработке двух значений, x и y, вместо одного значения.

**Листинг 11.3 Обновление атрибутов x и y и положения рукоятки**

```
   updateXY(x, y) {
       let hPos =
               x - this.dom.thumb.offsetWidth/2;
       let vPos = y - this.dom.thumb.offsetHeight/2;
       if (hPos > this.offsetWidth) {
           hPos = this.offsetWidth;
       }
       if (hPos < 0) {
           hPos = 0;
       }
       if (vPos > this.offsetHeight) {
           vPos = this.offsetHeight;
       }
       if (vPos < 0) {
           vPos = 0;
       }
       this.x = (hPos / this.offsetWidth) * 100;
       this.y = (vPos / this.offsetHeight) * 100;
   }

   refreshCoordinates() {
       this.dom.thumb.style.left = (
       this.x/100 * this.offsetWidth - this.dom.thumb.offsetWidth/2) + 'px';
       this.dom.thumb.style.top =
       (this.y/100 * this.offsetHeight - this.dom.thumb.offsetWidth/2) + 'px';
   }
}

if (!customElements.get('wcia-coord-picker')) {
   customElements.define(
        'wcia-coord-picker', CoordPicker);
}
```

Получаем центрированные (относительно рукоятки) координаты для x и y

Ограничиваем x (или значение по горизонтали) внутри границ компонента

Ограничиваем y (или значение по вертикали) внутри границ компонента

Обновляем атрибуты x и y через JS API компонента

Обновляем горизонтальное и вертикальное положения рукоятки

Определяем элемент и тег из класса компонента

Вы заметите, что в верхней части класса наш метод установки фонового цвета просто обновляет цвет фона всего компонента. Мы изучим код, который будет далее, чтобы увидеть, как создать идеальный двунаправленный градиент, но установить общий цвет так же просто, как и то, что мы поместили в обработчик `attributeChangedCallback`:

```
this.style.backgroundcolor = newValue;
```

Зачем делать эту часть API, если можно установить CSS-правило для цвета фона вне компонента? Все просто: мы заставляем API инструмента для выбора координат работать так же, как API ползунка. Намного проще одновременно установить свойство `background-color` и для ползунка, и для этого компонента, используя похожий API, вместо того чтобы взаимодействовать с ними по-разному, особенно когда у них очень схожие функциональные возможности.

## 11.2.2 HTML-код и стили инструмента для выбора координат

Перейдя к нашему файлу template.js, в котором содержится HTML-код и стили для инструмента выбора координат, вы снова не заметите большой разницы между ним и ползунком. В приведенном ниже листинге показан импорт.

**Листинг 11.4 Файл template.js инструмента для выбора координат**

```
export default {
   render() {
       return `${this.css()}
               ${this.html()}`;
   },

   mapDOM(scope) {
       return {
           thumb: scope.getElementById('thumb')
       }
   },

   html() {
       return `<div id="bg-overlay-a"></div>
               <div id="bg-overlay-b"></div>
               <div id="thumb"></div>`;
   },

   css() {
       return `<style> . . . style to be continued . . . </style>`;
   }
}
```

Возвращаем весь HTML-код и стили для визуализации компонента

Класс должен получить доступ только к одному элементу: рукоятке

HTML-код компонента включает в себя двухслойный фон и рукоятку

Как обычно, стили оказались немного длинными, поэтому мы продолжим в следующем листинге:

**Листинг 11.4 Файл template.js инструмента для выбора координат (*продолжение*)**

```
:host {
    display: inline-block;
    position: relative;
}

#bg-overlay-a {
    width: 100%;
    height: 100%;
    border-radius: 3px;
    position: absolute;
    background: linear-gradient(to right, #fff 0%, rgba(255,255,255,0) 100%);
}

#bg-overlay-b {
    width: 100%;
    height: 100%;
    border-radius: 3px;
```

Первый фон содержит белый градиент, который становится прозрачным слева направо

Второй фон содержит градиент, который постепенно переходит от прозрачного к черному в вертикальном направлении

```
    position: absolute;
    background: linear-gradient(to bottom, transparent 0%, #000 100%);
}

#thumb {  ←—— Стили для рукоятки
    width: 5px;
    height: 5px;
    position: absolute;
    border-style: solid;
    border-width: 3px;
    border-color: white;
    border-radius: 6px;
    pointer-events: none;
    box-shadow: 0 4px 8px 0 rgba(0, 0, 0, 0.2), 0 6px 20px 0 rgba(0, 0, 0,
     0.19);
}
```

Как я уже говорил, общий цвет устанавливается в качестве фонового цвета компонента. Чтобы добиться эффекта увеличения насыщенности слева направо, мы используем тег `<div>` с белым цветом фона, который постепенно переходит из непрозрачного в прозрачный при перемещении слева направо. Поверх этого слоя расположен похожий градиент, который постепенно переходит от абсолютно прозрачного в черный цвет при движении сверху вниз. Поместите оба этих слоя поверх полностью сплошного фона (цвет фона нашего компонента), и у нас получится отличная небольшая карта насыщенности и яркости! Но стоит отметить: как и в случае с фоном ползунка, я заимствовал этот подход на сайте https://cssgradient.io.

### 11.2.3 Демостраницы для компонентов

Давайте закончим этот компонент, создав для него демонстрационную страницу. На самом деле учитывая, что все эти демостраницы очень похожи, давайте выберем сразу две. В приведенном ниже листинге показан код страницы для инструмента для выбора координат.

**Листинг 11.5 Демонстрационная страница инструмента для выбора координат**

```
<html lang="en">
<head>
   <title>Coordinate Picker Demo</title>
   <script type="module"  ←—— Импортируем класс нашего компонента
        src="coordpicker.js">
   </script>

   <style>
       wcia-coord-picker {  ←—— Устанавливаем определенный размер
           width: 400px;        для компонента с помощью стилей
           height: 400px;
       }
   </style>
```

```
  </head>
<body>
   <wcia-coord-picker
       backgroundcolor="#ff0000">
   </wcia-coord-picker>
</body>
</html>
```

**Помещаем инструмент для выбора координат на страницу с исходным красным цветом фона**

Я не буду приводить такой же код для палитры цветов и еще одну очень похожую демостраницу. Давайте создадим страницу для нашего последнего компонента, пока мы работаем над этим. Вспомним, как выглядит структура проекта, с помощью рис. 11.7 и создадим файл demo.html в папке components/colorpicker.

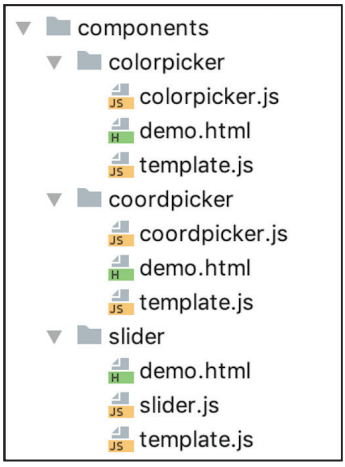



**Рис. 11.7 Вспомним, как выглядит структура файлов и папок нашего компонента при добавлении новой демостраницы**

Неудивительно, что мы можем просто скопировать и вставить содержимое из одного файла в другой, а потом просто внести некоторые изменения. Сначала содержимое тегов `<title>` и `<script>` меняется с

```
<title>Coordinate Picker Demo</title>
<script type="module" src="coordpicker.js">
</script>
```

на

```
<title>Color Picker Demo</title>
<script type="module" src="colorpicker.js">
</script>
```

Размер довольно произвольный, и размер 400×400 пикселей подойдет для демонстрации палитры цветов. Только CSS-селектор нужно будет изменить. Просто замените `wcia-ordin-picker` на `wcia-color-picker` в блоке `<style>`.

И конечно же, последнее: компонент объявляется на странице по-другому. Вместо

```
<wcia-coord-picker backgroundcolor="#ff0000"></wcia-coord-picker>
```

появляется

```
<wcia-color-picker></wcia-color-picker>
```

Однако, прежде чем двигаться дальше, важно проанализировать то, что мы имеем на данный момент. Мы сделали первые шаги для создания палитры цветов или любого количества компонентов пользовательского интерфейса, которые могут использовать ползунок или инструмент для выбора координат. Нечто столь же простое, как и любой из них, является прекрасным примером чрезвычайно простого компонента, который можно использовать во всей библиотеке.

Мы вполне можем запустить библиотеку компонентов, используя даже еще более простой компонент пользовательского интерфейса: кнопку. Однако, с чего бы вы ни начали, важно действовать предусмотрительно, чтобы все можно было использовать повторно и оно было адаптируемо к большинству сценариев. Наши стартовые компоненты продемонстрировали это здесь. В слайдере мы оставили основной фон вне компонента, чтобы иметь возможность сделать радужный градиент для ползунка с тонами или клетчатого рисунка для ползунка с выбором прозрачности. Но как насчет фона в тегах `<div>` внутри компонента? Он работает только как градиент с затуханием, хотя его цвет можно настраивать.

Возможно, нам понадобится ползунок в другом контексте, где нам нужно, чтобы этот фоновый слой делал что-то другое, или, может быть, нам нужно больше фоновых слоев! То же самое относится к инструменту для выбора координат. Возможно, вместо того чтобы сделать фон постоянной частью нашего компонента, лучше добавить фоны через слоты. Ползунок даже можно превратить в градиентный слайдер, которым мы изначально вдохновлялись, у которого будет несколько рукояток.

Запуск этих компонентов в качестве начального этапа набора компонентов пользовательского интерфейса требует долгого планирования и рефакторинга, по мере того как выявляются различные варианты использования. В этой книге мы рассматриваем только два варианта использования для ползунка и лишь один для инструмента выбора координат, но если бы мы продолжили работу с библиотекой компонентов, их могло бы быть намного больше. Учитывая это, давайте перейдем к нашему основному варианту использования: палитре цветов.

## 11.3 Палитра цветов

Теперь перейдем к третьему и последнему компоненту, который содержит два других компонента, ползунок и инструмент для выбора координат, а также поля ввода. Вместе они образуют общую палитру цветов, как показано на рис. 11.8.

Создав файл demo.html, давайте сосредоточимся на том, чтобы запустить что-то визуально, даже если мы пока не можем взаимодействовать с ним. Кроме того, позже мы займемся API компонента, чтобы сосредоточиться на его внутренней работе. Первое, что мы должны сделать, – это создать класс компонента (components/colorpicker/colorpicker.js).

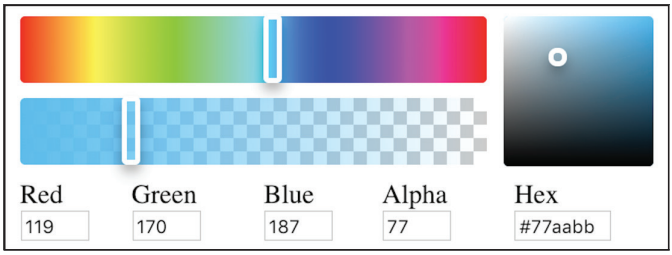



**Рис. 11.8 Палитра цветов**

**Листинг 11.6 Определение компонента для палитры цветов**

```
import Template from './template.js';

export default class ColorPicker extends HTMLElement {
   constructor() {
       super();
       this.attachShadow({mode: 'open'});
       this.shadowRoot.innerHTML = Template.render();
       this.dom = Template.mapDOM(this.shadowRoot);
   }
}
if (!customElements.get('wcia-color-picker')) {
   customElements.define('wcia-color-picker', ColorPicker);
}
```

Импортируем модуль шаблона HTML/CSS

Конструктор, который создает Shadow DOM, отображает HTML-код и стили и кеширует нужные нам элементы

Определяем компонент, отображая его в тег <wcia-color-picker>

Теперь этого достаточно, чтобы переключиться на модуль template.js для работы с HTML и CSS. Что касается HTML-кода в следующем листинге, мы продолжим и создадим разметку для ползунка, инструмента для выбора координат и полей ввода с метками.

**Листинг 11.7 HTML-код для палитры цветов**

```
import Slider from '../slider/slider.js';
import CoordinatePicker from '../coordpicker/coordpicker.js';

export default {
    render() {
        return `${this.css()}
                ${this.html()}`;
    },

    html() {
        return `<div class="container">
                    <div class="row">
                        <div class="slider-container">
                            <wcia-slider
                                id="hue-slider"
                                value="50">
                            </wcia-slider>
                            <wcia-slider
                                id="transparency-slider"
```

Импортируем ползунок и инструмент для выбора координат

Возвращаем полную строку разметки HTML/CSS

Два экземпляра ползунка, один для выбора тона и второй для выбора прозрачности

```
                value="0">
            </wcia-slider>
        </div>

        <wcia-coord-picker x="50" y="50"
          id="saturation-brightness"></wcia-coord-picker>
    </div>

    <div class="row">
        <div class="text-inputs">
```

Инструмент для выбора координат

Продолжая работать с HTML-кодом, мы сталкиваемся с полями ввода. Хотя они не являются пользовательскими веб-компонентами, мы будем обращать внимание на то, что вводит пользователь, почти так же, как и в случае с ползунком и инструментом для выбора координат.

**Листинг 11.7 HTML-код палитры цветов (*продолжение*)**

```
            <div>
                <label class="top" for="textInputR">Red</label>
                <input id="textInputR"
                       type="number" value="0"
                         max="255" size="4" min="0">
            </div>

            <div>
               <label class="top" for="textInputG">Green</label>
               <input id="textInputG"
                         type="number" value="0"
                         max="255" size="4" min="0">
            </div>

            <div>
               <label class="top" for="textInputB">Blue</label>
               <input id="textInputB"
                         type="number" value="0"
                         max="255" size="4" min="0">
            </div>

            <div>
               <label class="top" for="textInputA">Alpha</label>
               <input id="textInputA"
                         type="number" value="0"
                          max="100" min="0" size="4">
            </div>

            <div>
              <label class="top" for="textInputHex">Hex</label>
              <input id="textInputHex"
                        type="text" width="50px" size="8">
            </div>
        </div>
    </div>
</div>`
    }
}
```

Поля для ввода значений цвета: красный, зеленый и синий (принимаются значения от 0 до 255)

Поле для ввода значений альфа-канала и прозрачности (принимаются значения от 0 до 100)

Поле для ввода шестнадцатеричного кода цвета

В качестве последнего шага, чтобы увидеть, с чем мы работаем, мы добавляем стили в этот модуль импорта.

**Листинг 11.8 CSS для палитры цветов**

```
css() {
   return `<style>                    ← Стили для обычных элементов, используемых для макета
              :host {
                  width: 100%;
                  display: inline-block;
              }

              .container {
                  padding: 10px;
              }

              .text-inputs {
                  display: flex;
                  width: 100%;
                  justify-content: center;
              }

              .row {
                  display: flex;
              }

              .slider-container {
                  flex: 1;
                  padding-right: 10px;
              }
```

Продолжая работать с CSS, нужно отметить, что предыдущие стили были для макета для стандартных, повседневных элементов. Приведенные далее стили предназначены для наших пользовательских компонентов и также служат для того, чтобы ползунки можно было различать по фону.

**Листинг 11.8 CSS для палитры цветов (*продолжение*)**

```
              #hue-slider, #transparency-slider {
                  width: 100%;
                  height: 40px;
                  margin-bottom: 5px;
                  border-radius: 3px;
              }

              #saturation-brightness {
                  width: 90px;
                  height: 90px;
                  border-radius: 3px;
              }                          Специальный радужный фон
                                         для ползунка с выбором тона
              #hue-slider {
                  background: linear-gradient(to right, red 0%, #ff0 17%,
                  lime 33%, cyan 50%, blue 66%, #f0f 83%, red 100%);
```

```
            }
            #transparency-slider {
                background-image: linear-gradient(45deg, #ccc 25%,
                transparent 25%),linear-gradient(-45deg, #ccc 25%,
                transparent 25%),linear-gradient(45deg, transparent 75%,
                #ccc 75%),linear-gradient(-45deg, transparent 75%,
                #ccc 75%);
                background-size: 16px 16px;
                background-position: 0 0, 0 8px, 8px -8px, -8px 0px;
            }
        </style>`;
}
```

Специальный клетчатый фон для ползунка с выбором прозрачности

На рис. 11.9 показано, как будут выглядеть эти ползунки после подключения логики на следующих этапах для управления этими цветными слоями.

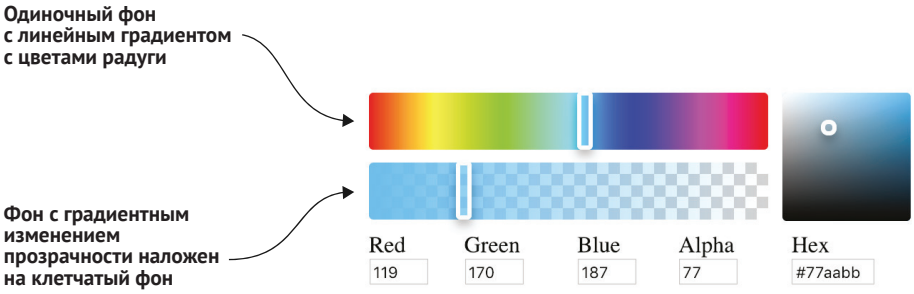



**Рис. 11.9 Фоны ползунков с возможностью выбора тона и прозрачности**

В качестве последнего шага в этом модуле импорта мы будем кешировать элементы, которые важны для организации взаимодействия. В этом листинге показаны эти восемь элементов, когда мы возвращаем ссылки на них.

**Листинг 11.9 Важные элементы для кеширования и возврата ссылок, используемых для взаимодействия**

```
mapDOM(scope) {
   return {
       hue: scope.getElementById('hue-slider'),
       transparency: scope.getElementById('transparency-slider'),
       satbright: scope.getElementById('saturation-brightness'),
       textInputR:
scope.getElementById('textInputR'),
       textInputG: scope.getElementById('textInputG'),
       textInputB: scope.getElementById('textInputB'),
       textInputA: scope.getElementById('textInputA'),
       textInputHex: scope.getElementById('textInputHex'),
   }
}
```

Три наших элемента веб-компонента кешированы

Кешируются элементы полей для ввода цвета в формате RGBA и шестнадцатеричного кода цвета

Здесь мы работаем с еще несколькими элементами, а не только с ползунком или инструментом для выбора координат. Мы хотим кешировать ссылки для этих компонентов, но нам нужно будет взаимодействовать с каждым отдельным текстовым полем, а также слушать и отвечать, когда пользователи вводят цифры и текст в качестве значений.

## 11.3.1 Наблюдение за изменениями атрибутов для взаимодействия

Множество раз при работе над таким сложным компонентом я буду добавлять слушателей событий к каждому элементу и соединять их с обработчиком, чтобы предпринять что-то, когда это событие происходит. Хотя так и можно было бы сделать, пользовательские события не были добавлены в компоненты ползунка и инструмента для выбора координат. Мы рассмотрим использование пользовательских событий в веб-компонентах ближе к концу книги, а сейчас есть другой способ, с помощью которого можно следить за изменениями и отвечать на них.

Не часто встретишь упоминание о Mutation Observer, но я думаю, что веб-компоненты – идеальный вариант для этого. При настройке объекта `MutationObserver` вы предоставляете ему определенный фрагмент HTML-кода для наблюдения, как показано на рис. 11.10. Вы также устанавливаете обработчик, который нужно вызвать, когда эти изменения происходят.

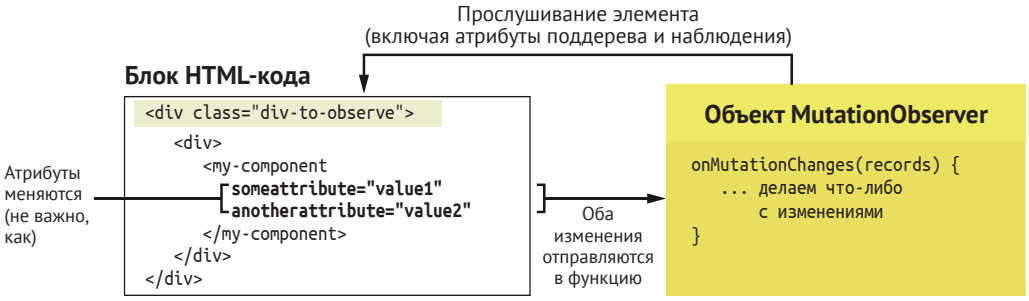



**Рис. 11.10 Mutation Observer в действии**

Хотя нам не нужно просматривать внутреннее текстовое содержимое наших HTML-тегов, атрибуты нашего веб-компонента постоянно обновляются по мере их использования. С помощью некоторых пользовательских параметров мы можем отслеживать изменения атрибутов и реагировать на них. Возвращаясь к классу компонентов (components/colorpicker/colorpicker.js) в приведенном ниже листинге, мы можем настроить `MutationObserver` для отслеживания этих изменений.

**Листинг 11.10 Прослушивание изменений атрибутов с помощью `MutationObserver`**

```
constructor() {
  super();
```

```
    this.attachShadow({mode: 'open'});
    this.shadowRoot.innerHTML = Template.render();
    this.dom = Template.mapDOM(this.shadowRoot);
    const observer = new MutationObserver( e => this.onMutationChange(e));
    observer.observe(this.shadowRoot, { attributes: true, subtree: true });
}

onMutationChange(records) {
  records.forEach( rec => {
      this.data = Handlers.update({
          model: this.data,
          dom: this.dom,
          component: this,
          element: rec.target,
          attribute: rec.attributeName,
      });
  });
}
```

**Создаем новый MutationObserver с обработчиком для прослушивания изменений**

**Наблюдаем за теневой моделью DOM компонента, определяя, как следить за атрибутами, а также наблюдать за всеми элементами внутри**

**Пользовательский обработчик еще не обсуждался**

**MutationObserver может сообщать о множестве изменений, поэтому нам нужно перебрать элементы массива, чтобы обработать их**

Теперь, каждый раз, когда атрибут значения ползунка меняется или изменяется значение `x` или `y` инструмента для выбора координат, мы будем получать уведомления! Мы также будем получать уведомления о других вещах, которые нам не нужны, таких как атрибут `class` или даже атрибуты `backgroundcolor`, но эти изменения можно игнорировать.

Чтобы обрабатывать эти события изменений, я разместил логику вне класса компонента, чтобы она не была слишком сложной и громоздкой. Функция `Handlers.update` находится в другом модуле импорта. Этой функции передаются две вещи, которые нам дает `Mutation-Observer`: ссылка на измененный элемент и то, какой атрибут изменился. Мы также передаем этому обработчику ссылку на `this` или на сам компонент палитры цветов, а также на `this.dom` или кешированные элементы, созданные в модуле template.js. Имея доступ ко всем необходимым элементам, эта функция обновления может изменять атрибуты и свойства любого из элементов. Например, при изменении тона этот новый модуль может реагировать на изменение, устанавливая цвет фона для ползунка с выбором прозрачности, как и для инструмента выбора координат, и может обновлять все поля ввода, отражающие новый цвет.

Наконец, свойство этого класса веб-компонентов `this.data` передается в функцию `Handlers.update` и возвращается. В основном оно накапливает данные.

Снова используя тон в качестве примера, значение будет сохранено и будет находиться в этих данных. При изменении яркости или насыщенности мы не хотим терять значение тона, поэтому сохраняем свойство `this.data` в качестве постоянного объекта.

Конечно, ничего из этого не будет работать, если мы не импортируем модуль. Добавив импорт вместе с модулем template.js в начало класса, мы тем самым сделаем свою работу:

```
import Template from './template.js';
import Handlers from './handlers.js';
```

При импорте `handlers` фактически импортируется еще один отдельный модуль для выполнения всех необходимых цветовых вычислений. Он предлагает преобразование из шестнадцатеричного кода в RGB, RGB в шестнадцатеричный код и другие полезные утилиты. Я не буду рассказывать о логике, находящейся внутри этих модулей. Ни один из них не учит концепциям веб-компонентов; они просто предоставляют код для контроля состояния нашего пользовательского интерфейса. Однако если вам интересно, обратитесь к репозиторию для этой книги на GitHub.

Благодаря этим дополнительным функциональным возможностям структура нашего проекта увеличилась на два файла. На рис. 11.11 показаны все модули в нашем компоненте и более подробная информация о том, как осуществляется взаимодействие.


**Рис. 11.11 Структура проекта с новыми модулями**

## 11.3.2 Реакция на изменения в полях ввода

Было бы просто чудесно, если бы при изменении значения каждого элемента `<input>` также менялся бы и атрибут. К сожалению, это работает по-другому; атрибут `value` просто не обновляется, когда пользователь

изменяет содержимое поля ввода. Поэтому нам нужен еще один слушатель – слушатель простого изменения. Вы, наверное, подумали о том, чтобы связать каждый элемент `<input>` с собственным слушателем событий, но на самом деле мы можем прослушивать события изменений, которые всплывают к нашему корню теневого дерева с помощью одного слушателя, как показано на рис. 11.12. В листинге 11.11 продемонстрирована работа одного из таких слушателей событий на практике.

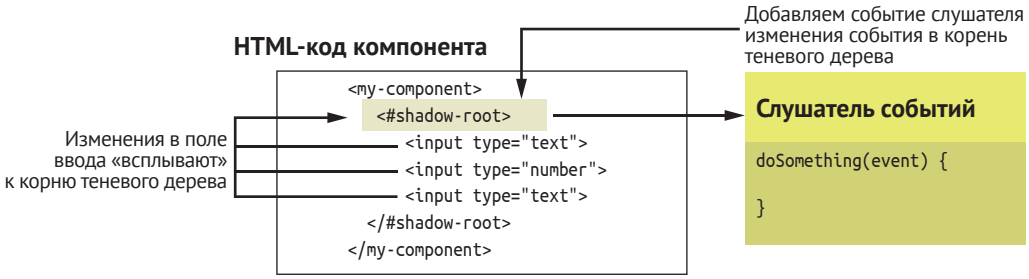



**Рис. 11.12 Всплывание событий**

**Листинг 11.11 Прослушивание событий изменения внутри теневой модели DOM компонента**

```
constructor() {
   super();
   this.attachShadow({mode: 'open'});
   this.shadowRoot.innerHTML = Template.render();
   this.dom = Template.mapDOM(this.shadowRoot);
   const observer = new MutationObserver( e => this.onMutationChange(e));
   observer.observe(this.shadowRoot, { attributes: true, subtree: true });
   this.shadowRoot.addEventListener('change', e =>
     this.onInputValueChanged(e));
}

onInputValueChanged(e) {
   this.data = Handlers.update( {
       model: this.data,
       dom: this.dom,
       component: this,
       element: e.target,
   });
}
```

Слушатель событий изменений для полей ввода

Обработчик событий изменений

Обратите внимание, что мы отправляем те же самые объекты в функцию `Handlers.update` (за исключением имени атрибута, поскольку здесь оно неприменимо). Все входящие данные из события изменения или `MutationObserver` могут обрабатываться одинаково, что делает функцию `Handlers.update` довольно простой. После всей проделанной работы по настройке неплохо было бы попытаться реализовать этот модуль самостоятельно наряду с цветовыми вычислениями (http://www.easyrgb.com/en/

math.php), который я использовал в качестве полезного ресурса для написания функций преобразования. Опять же, детали этих модулей импорта были бы несколько длинными, для того чтобы описывать их здесь, и они не являются специфичными для веб-компонентов; поэтому хотя они и не включены сюда, они доступны в репозитории для этой книги на GitHub.

### 11.3.3 Реакция на изменения атрибутов

Существует одна последняя концепция, связанная с веб-компонентами, используемая для обеспечения функциональности палитры цветов. Наш компонент должен обновлять значения цвета и прозрачности для своих собственных атрибутов. Цвет будет представлен в виде шестнадцатеричного значения, а прозрачность будет выражаться в процентах от 0 до 100. Атрибуты будут называться `hex` и `alpha` соответственно.

Чтобы обновить эти значения после изменения какого-либо аспекта цвета, будь то ползунок с выбором тона, красное поле для ввода значения цвета Red (Красный) или инструмент выбора координат, функция `Handlers.update` принимает ссылку на компонент, поэтому она может легко обновить эти атрибуты. Однако нам нужно добиться того, чтобы мы могли слушать изменения извне. Другая часть приложения, в которой находится палитра цветов, могла бы установить цвет компонента, и должен последовать ответ, при этом все применимые элементы пользовательского интерфейса будут изменены (будут обновлены поля для ввода значений, ползунки и инструмент для выбора координат). Хотя конкретная логика для достижения этой цели может находиться в функции `Handlers.update`, нам все равно нужно реагировать на изменения и отправлять информацию в модуль.

Поскольку мы уже определили, что у API есть свойства `hex` и `alpha`, очевидно, что нам нужно слушать и реализовывать для рефлексии. В приведенном ниже листинге показана реализация обратных вызовов изменения атрибутов и методов получения и установки в файле components/colorpicker/colorpicker.js:

**Листинг 11.12 Реализация API нашего компонента**

```
static get observedAttributes() {
   return ['hex', 'alpha'];
}
```

Определяем, какие атрибуты необходимо прослушивать на предмет изменений

```
attributeChangedCallback(name, oldVal, newValue) {
   switch (name) {
       case 'hex':
       case 'alpha':
           if (oldVal !== newValue) {
               this.data = Handlers.update({
                   model: this.data,
                   dom: this.dom,
                   component: this,
                   element: this,
```

Прослушиваем изменения атрибутов

```
                attribute: name,
            });
        }
        break;
    }
}

set hex(val) {
    this.setAttribute('hex', val);
}

get hex() {
    return this.getAttribute('hex');
}

set alpha(val) {
    this.setAttribute('alpha', val);
}

get alpha() {
    return this.getAttribute('alpha');
}
```

Реагируем на изменения с помощью модуля импорта Handlers, передавая имя атрибута

Геттеры и сеттеры для атрибутов hex и alpha

Хотя теперь все в порядке, нужно решить еще одну проблему, связанную с API. Палитра цветов работает неправильно при загрузке, потому что нет никакого значения цвета или прозрачности, с которого можно было бы начать, если атрибут не был указан! Чтобы завершить функциональную реализацию компонента, давайте определим значения по умолчанию. Мы можем использовать два статических метода чтения в верхней части класса `ColorPicker`:

```
export default class ColorPicker extends HTMLElement {
    static get DEFAULT_HEX() { return '#77aabb'; }
    static get DEFAULT_ALPHA() { return 100; }
```

Значения по умолчанию, которые я здесь использовал, несколько произвольны. Вы можете использовать любые значения. Затем мы можем заполнить эти значения при добавлении компонента на страницу. В приведенном ниже листинге для этой цели добавляется метод `connectedCallback`:

**Листинг 11.13 Установка цвета и прозрачности при запуске, если они не определены**

```
connectedCallback() {
    if (!this.hex) {
        this.hex = ColorPicker.DEFAULT_HEX;
    }
    if (!this.alpha) {
        this.alpha = ColorPicker.DEFAULT_ALPHA;
     }
}
```

Если атрибут шестнадцатеричного кода цвета не существует, используется значение по умолчанию

Если атрибут alpha не существует, используется значение по умолчанию

Поскольку компонент палитры цветов поддерживает рефлексию, эти свойства легко установить с помощью JS API. При прослушивании изме-

нений атрибутов при настройке этих двух параметров будут вноситься все соответствующие изменения в каждый элемент пользовательского интерфейса в компоненте. Также важно отметить, что речь идет об обработчике `connectedCallback`, а не конструкторе. При вызове конструктора атрибутам еще слишком рано находиться в жизненном цикле компонента!

## 11.4 Работаем над внешним видом палитры

Пока мы сделали немного! Три компонента готовы, и все они работают сообща, обслуживая палитру цветов, которую можно использовать в реальном проекте. Хотя было бы здорово иметь демонстрационный компонент, как во всех трех случаях, еще лучше увидеть финальный компонент в более реальном контексте. Вот почему я подумал, что было бы неплохо настроить тестовую страницу с установленным компонентом, изменить цвет фона страницы и установить степень прозрачности для текста. Приведенный ниже листинг делает это за нас, чтобы дать нам более полную демоверсию. Мы просто создадим ее в корне нашего проекта в виде файла index.html.

**Листинг 11.14 Демоверсия палитры цветов, которая может влиять на элементы на странице**

```
<html lang="en">
<head>
   <title>Color Picker Component</title>
   <script type=
       "module" src="components/colorpicker/colorpicker.js"></script>
   <style>
       h1 {
           font-family: sans-serif;
           font-weight: bolder;
           color: white;
           text-shadow:
               -1px -1px 0 #000,
               1px -1px 0 #000,
               -1px 1px 0 #000,
               1px 1px 0 #000;
       }
   </style>
</head>
<body>
   <wcia-color-picker
       hex="#7687db"
       alpha="75">
   </wcia-color-picker>

   <h1>Transparency</h1>
   <script>
       document.body.style.backgroundColor =
           document.querySelector('wcia-color-picker').getAttribute('hex');
```

Импортируем палитру цветов

Делаем черную рамку для текста на потенциальном темном фоне, когда используем палитру цветов

Добавляем компонент на страницу

Используем исходные значения палитры цветов, чтобы установить прозрачность текста и цвет фона страницы

```
        document.querySelector('h1').style.opacity =
            document.querySelector('wcia-color-picker').getAttribute('alpha');

        const observer = new MutationObserver(
            function(records) {
              records.forEach( rec => {
                switch (rec.attributeName) {
                    case 'hex':
                        document.body.style.backgroundColor = rec.target.hex;
                        break;

                    case 'alpha':
                        document.querySelector('h1').style.opacity =
                            rec.target.alpha / 100;
                }
              });
        });
        observer.observe(document.querySelector('wcia-color-picker'),
            { attributes: true });
    </script>
</body>
</html>
```

Наблюдаем за изменениями атрибутов в палитре цветов и обновляем текст и фон страницы

Хотя новая демонстрация позволяет нам более осмысленно взаимодействовать с палитрой цветов, она также подчеркивает еще кое-что – уродливый внешний вид! На рис. 11.13 видно, что не так много внимания уделено деталям дизайна.

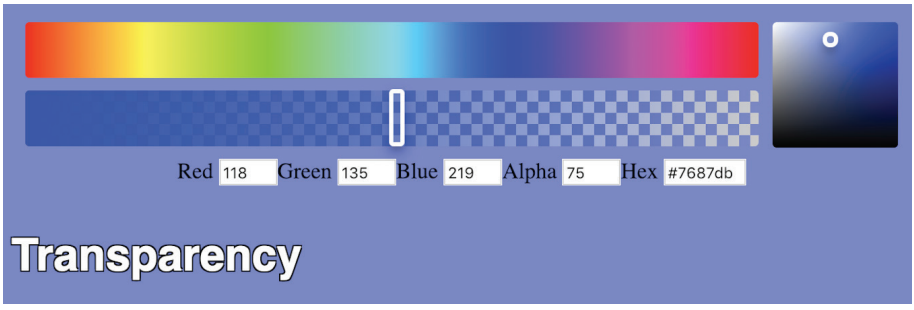



**Рис. 11.13 Новая палитра цветов, где можно изменять цвет страницы и прозрачность текста**

## 11.4.1 Загрузка CSS-переменных для улучшения дизайна

Хотя с ползунком и инструментом для выбора координат не так много проблем, поскольку движущихся частей немного, поля для ввода значений абсолютно не стилизованы, а многие более мелкие детали, например закругленные углы, довольно произвольны.

Подобные детали в действительности должны определяться на общесистемном уровне. Если рассуждать с точки зрения более крупного приложения, вы должны убедиться, что все эти детали согласованы.

Здесь может быть не только палитра цветов, но и широкий спектр компонентов. Если все эти компоненты имеют разные стили, все будет выглядеть еще хуже!

Современные дизайн-системы сейчас только догоняют веб-компоненты. Многие системы, такие как Bootstrap, сделаны таким образом, что теневая модель DOM полностью блокирует их стили от компонента. Вместо этого давайте пойдем более легким путем и создадим собственные общие стили с помощью импорта JS. Эти правила можно импортировать в любой компонент, которому они нужны!

Прежде чем сделать это, давайте изложим основные правила, используя CSS-переменные в листинге 11.15. Это может быть модуль импорта JS, если мы хотим, чтобы это был он, но, учитывая, что в нашем случае более широкий контекст приложения или дизайн-система отсутствует, мы просто поместим его в корень нашего проекта в файл vars.css.

**Листинг 11.15 CSS-переменные, которые будут использоваться в гипотетическом приложении и наших компонентах**

```
:root {
   --text-xsmall: .5em;
   --text-small: .7em;
   --text-medium: 1em;
   --text-large: 1.3em;
   --text-xlarge: 1.5em;

   --color-pureblack: black;
   --color-black: #2a2a2a;
   --color-lightblack: #4a4a4a;
   --color-darkgrey: #6a6a6a;
   --color-grey: #7a7a7a;
   --color-lightgrey: #9a9a9a;
   --color-darkwhite: #dadada;
   --color-white: #fafafa;
   --color-purewhite: #ffffff;

   --text-color: var(--color-lightblack);
   --text-inverted-color: var(--color-white);
   --border-color: var(--color-lightblack);
   --border-color-light: var(--color-darkwhite);
   --border-inverted-color: var(--color-white);
   --background-color: var(--color-white);
   --background-inverted-color: var(--color-lightblack);

   --border-radius: 6px;
   --border-width-thick: 3px;
   --border-width: 1px;
   --padding-medium: 5px;
}
```

Определяем различные размеры текста

Различные цветовые переменные

Отображаем цветовые переменные в определенные элементы, такие как текст, рамка и фон

Устанавливаем величину поля и ширину рамок

Я, вероятно, немного переборщил, определяя не только то, что может понадобиться нашим существующим компонентам, но это неплохие примеры правил, из которых можно построить общий язык дизайна для

вашего приложения. Мы даже можем начать заменять жестко закодированные значения в ползунке (components/slider/template.js), как показано в приведенном ниже листинге.

**Листинг 11.16 Замена значений CSS-переменными в ползунке**

```
css() {
   return `<style>
               :host {                          Используем CSS-переменные для самого
               ...                              компонента, чтобы управлять рамкой
                   border-radius: var(--border-radius);
               ...
               }
               #bg-overlay {                    Используем CSS-переменные для тега
               ...                              фона <div>, чтобы управлять рамкой
                   border-radius: var(--border-radius);
                   ...
               }                                Используем CSS-переменные для рукоятки,
               #thumb {                         чтобы управлять рамкой
               ...
                   border-width: var(--border-width-thick);
                   border-color: var(--border-inverted-color);
                   border-radius: var(--border-radius);
               ...
               }
           </style>`;
}
```

Теперь мы можем сделать то же самое для инструмента выбора координат (components/coordpicker/ template.js):

**Листинг 11.17 Замена значений CSS-переменными в инструменте для выбора координат**

```
css() {
   return `<style>
               #bg-overlay-a {                  Используем CSS-переменные
                   ...                          для двух фонов, чтобы управлять рамкой
                   border-radius: var(--border-radius);
                   ...
               }

               #bg-overlay-b {
                   ...
                   border-radius: var(--border-radius);
                   ...
               }                                Используем CSS-переменные для рукоятки,
               #thumb {                         чтобы управлять рамкой
                   ...
                   border-width: var(--border-width-thick);
                   border-color: var(--border-inverted-color);
                   border-radius: var(--border-radius);
```

```
                ...
            }
        </style>`;
}
```

Наконец, то же самое можно сделать и для палитры цветов (components/colorpicker/template.js).

**Листинг 11.18 Замена значений CSS-переменными в палитре цветов**

```
css() {
  return `<style>
              ...
            #hue-slider,
            #transparency-slider {
               ...
                border-radius: var(--border-radius);
            }

            #saturation-brightness {
                ...
                border-radius: var(--border-radius);
            }

            ...
        </style>`;
}
```

Используем CSS-переменные для двух ползунков, чтобы управлять рамкой

Используем CSS-переменные для инструмента выбора координат, чтобы управлять рамкой

Как видно на примере этих правил, согласованность дизайна проходит долгий путь. Больше нет риска использования несовместимых цветов или закругленных углов. Эти маленькие детали действительно имеют смысл. Хотя с помощью CSS-переменных мы достигли не очень многого, поскольку можем использовать по одному CSS-правилу за раз.

## 11.4.2 Использование импорта для более сложных стилей

Возвращаясь к импортируемым модулям, можно заняться всем остальным. На рис. 11.14 показана структура нового проекта, включая весь импорт «дизайн-системы».

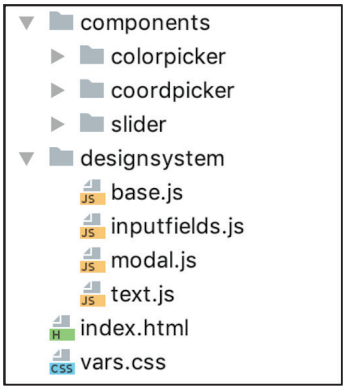



**Рис. 11.14 Файловая структура нашего проекта, включая CSS-переменные и импорт дизайн-системы**

Сперва в приведенном ниже листинге мы можем начать с того, что определим несколько универсальных правил.

**Листинг 11.19 Универсальные текстовые правила в designsystem/text.js**

```
export default {
   normal() {
       return `
           font-family: sans-serif;
           font-size: 1em;
           line-height: 1.2em;
           color: black;`;
   },
   inverted() { return `color: white;`; }
}
```

Определяем обычные текстовые правила, используя свойства color, size и font

Изменяем цвет, когда он кажется инвертированным на темном фоне

Этот небольшой набор текстовых правил можно затем импортировать в модуль, предназначенный для использования в качестве базового набора правил для любого компонента (designsystem/base.js):

```
import Text from './text.js';
export default {
   common() { return `${Text.normal()}`; }
}
```

Поскольку крайне ограниченное количество правил здесь не применимо к уже довольно солидно выглядящему ползунку или инструменту выбора координат, мы можем просто использовать этот импорт базовых стилей компонента в компоненте палитры цветов (components/colorpicker/template.js):

```
import Base from '../../designsystem/base.js';
```

Использовать его просто – это похоже на применение любого другого шаблонного литерала:

```
:host {
   ${Base.common()};
   width: 100%;
   display: inline-block;
}
```

Хотя и неплохо разобраться с основами, то немногое, что у нас есть, окажет лишь небольшое влияние на наш довольно минимальный компонент. Нет, большую часть того, что нужно стилизовать, составляют поля для ввода значений с соответствующими им метками. С этим можно справиться, определив конкретные правила в модуле designsystem/inputfields.js, как показано в приведенном ниже листинге.

**Листинг 11.20 Модуль, содержащий правила для полей ввода**

```
// designsystem/inputfields.js
import Text from './text.js';
```

Импортируем модуль text.js, поэтому мы можем использовать специфические CSS-правила

```
export default {
   css() {
       return `
           .ds-form-input {
               margin-right: 5px;
           }
           .ds-form-input
           ➥.ds-input-field-label {
               border-top-left-radius: var(--border-radius);
               border-top-right-radius: var(--border-radius);
               background-color: var(--background-inverted-color);
               padding: var(--padding-medium);
               display: block;
               font-size: var(--text-xsmall);
               ${Text.inverted()}
           }
           .ds-form-input
           ➥.ds-input-field-label.top {
               display: block;
           }
           .ds-form-input input {
               border-style: solid;
               border-width: var(--border-width);
               border-color: var(--border-color-light);
               padding: var(--padding-medium);
               font-size: var(--text-large);
           } `;
   }
}
```

Устанавливаем интервал для входного контейнера

Стили метки для поля ввода

В случае темного инвертированного цвета фона используйте функцию inverted()

Мы помещаем метку поверх поля ввода, поэтому укажите top, если в будущем хотите продолжить работу с изменениями

Стили фактического поля ввода

Как и ранее, мы должны импортировать этот модуль в модуль components/colorpicker/template.js:

```
import InputFields from '../../designsystem/inputfields.js';
import Base from '../../designsystem/base.js';
```

Мы также можем добавить его в стили в том же модуле:

```
css() {
    return `<style>
                ${InputFields.css()}
```

Обратите внимание, что мы импортируем полные CSS-правила с селекторами, поэтому вызов функции может быть встроен прямо в тег `<style>`, а не внутри селектора или блока, как это было раньше при использовании селектора `:host`. И поскольку мы используем новые селекторы, нам нужно будет добавить их в HTML-код в том же файле. Это простое добавление, которое применяется одинаково для всех наших полей ввода, поэтому я покажу только первое поле:

```
<div class="ds-form-input">
 <label class="ds-input-field-label top" for="textInputR">Red</label>
```

```
 <input id="textInputR" type="number" value="0" max="255" size="4" min="0">
</div>
```

Давайте взглянем на наши стилизованные поля ввода на рис. 11.15.

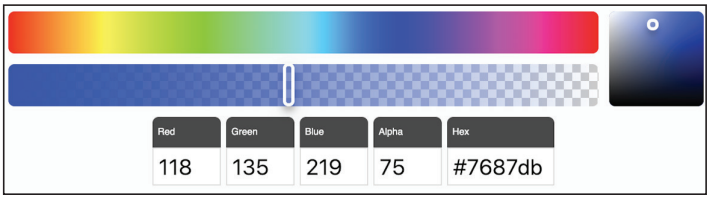



**Рис. 11.15 Стилизованные поля ввода после применения простой дизайн-системы**

Почти готово! Мне не нравится, что все элементы пользовательского интерфейса как будто плывут в пространстве. Я бы предпочел, чтобы у компонента был белый модальный фон, чтобы визуально соединять элементы и создавать ощущение зависания над страницей. Итак, давайте сделаем последний модуль импорта CSS, designsystem/modal.js, как показано в приведенном ниже листинге:

**Листинг 11.21 Модальный CSS-модуль**

```
// designsystem/modal.js

export default {
   css() {
       return `
           .ds-modal {
               ${this.rules()}
           }
       `;
   },

   rules() {
       return `
           background-color: var(--background-color);
           border-radius: var(--border-radius);
           box-shadow: 0 4px 8px 0 rgba(0, 0, 0, 0.2),
           0 6px 20px 0 rgba(0, 0, 0, 0.19);
       `;
   }
}
```

Селектор и шаблонный литерал, содержащий CSS-правила

Всего лишь CSS-правила

Тень вокруг элемента, чтобы было похоже, что он как будто висит над страницей

Здесь я немного выделяю вещи. Селектор `.ds-modal` указывает на функцию `rules()` для встраивания фактических CSS-правил. Причина в том, что в идеале я бы предпочел просто прикрепить этот селектор к палитре цветов и покончить с этим:

```
<wcia-color-picker class="ds-modal" hex="#7687db" alpha="75">
    </wcia-color-picker>
```

К сожалению, эти CSS-правила не пробьют теневую границу со стороны компонента (да, даже непосредственно на самом теге компонент все еще находится за пределами теневой границы) и не сделают ничего для стилизации компонента.

Тем не менее в других ситуациях, в которых нет теневой границы, было бы неплохо иметь этот селектор, поэтому мы оставим его в нашей дизайн-системе. Нарушение этих правил делает вещи двоякими, потому что в контексте нашего компонента теперь мы можем просто избегать вызова функции `Modal.css()` и вместо этого использовать свой собственный CSS-селектор, как в этом листинге:

**Листинг 11.22 Добавление CSS-правил**

```
css() {
   return `<style>
               ${InputFields.css()}

               :host {
                   ${Base.common()};
                   width: 100%;
                   display: inline-block;
               }

               :host(.modal) {
                   ${Modal.rules()}
               }
```

Стилизует любой компонент с классом .modal, назначенным с помощью CSS-правил

Как и во всех наших дополнительных CSS-модулях, обязательно добавьте импорт в файл components/colorpicker/template.js:

```
import InputFields from '../../designsystem/inputfields.js';
import Base from '../../designsystem/base.js';
import Modal from '../../designsystem/modal.js';
```

Обратите внимание, как мы вставили модальные правила в наш компонент. Мы сделали модальное лечение необязательным. Теперь модальный стиль применяется только в том случае, если моадльный класс присутствует в компоненте. Мы сделаем это в корневом файле index.html, чтобы увидеть его в полном контексте, когда меняем цвет фона страницы с помощью палитры цветов:

```
<wcia-color-picker class="modal" hex="#7687db" alpha="75"></wcia-color-picker>
```

Наконец, у нас есть стилизованный компонент палитры цветов, как показано на рис. 11.16.

Мало того, что мы сделали компонент более привлекательным, мы можем настраивать все глобально, чтобы придать компоненту (и окружающему приложению, если бы оно у нас было) другой внешний вид. Давайте сделаем так, чтобы он выглядел более забавно, как показано на рис. 11.17, просто увеличив значение для свойства `border-radius`. Это можно сделать прямо в файле vars.css, изменив значение в 6 пикселей до 12.

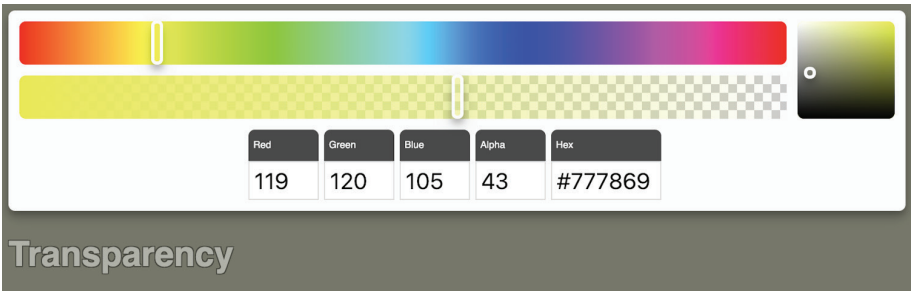



**Рис. 11.16 Стили палитры цветов, полученные с помощью наспех созданной дизайн-системы на базе модулей**

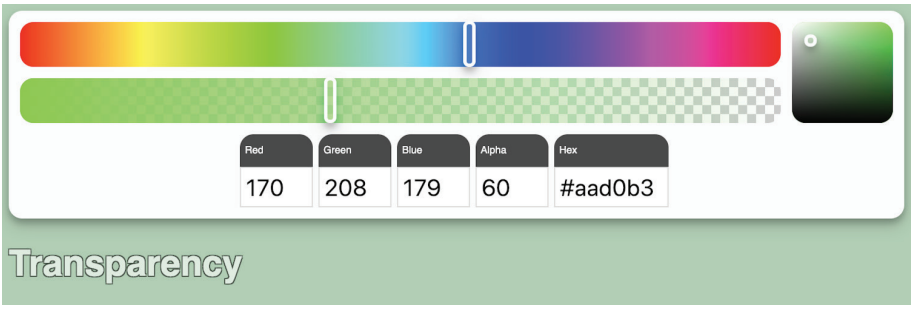



**Рис. 11.17 Дизайн палитры цветов легко настраивается путем изменения CSS-модулей или CSS-переменных**

Если вас волнуют только современные браузеры, которые поддерживают веб-компоненты, миссия выполнена! Мы создали красивую палитру цветов благодаря дизайну и техническому вдохновению, которое черпали с сайта cssgradient.io. Конечно, созданная нами дизайн-система имеет много возможностей для совершенствования, если нам необходимо создать более крупное приложение или платформу, но для компонента палитры цветов здесь достаточно пространства для стилизации снаружи, не меняя ни единой строки ни в одном из наших компонентов.

В следующей главе мы рассмотрим, что происходит, когда дело обстоит не так радужно, а целевой браузер, который нам нужен, изначально не предлагает поддержку веб-компонентов.

То, насколько браузер устарел, будет определять, как далеко мы должны зайти. Не волнуйтесь, эта палитра цветов будет работать практически везде!

## Резюме

Из этой главе вы узнали:

- как спланировать полукомплексный компонент пользовательского интерфейса, разбив наш дизайн на несколько частей, некоторые из

которых становятся собственными компонентами размером поменьше;

- как разбить компонент на входы и выходы и спланировать API вокруг них, включая использование концепции рефлексии для обеспечения общего API при использовании атрибутов тегов или JS;
- как создавать отдельные и универсальные правила стилевого оформления с помощью CSS-переменных или импортируемых модулей, которые можно использовать в своих компонентах и в более крупной системе, чтобы обеспечить согласованность дизайна.

# 12

# Сборка и поддержка старых браузеров

***Эта глава охватывает следующие темы:***

- комплектация модулей с помощью Rollup;
- транспиляция с использованием Babel, чтобы обеспечить поддержку веб-компонентов в IE11;
- запуск и объединение сценариев с помощью npm и package.json;
- использование зависимостей для разработки в package.json;
- полизаполнение CSS-переменных для IE11.

В предыдущей главе мы закончили создание компонента палитры цветов с возможностью повторного использования, состоящего из нескольких различных пользовательских компонентов. Он работает довольно хорошо, но теперь вопрос состоит в том, подходит ли этот компонент для всех ваших целевых пользователей. Конечно, это возможно, и на этом мы могли бы и остановиться. Компонент, который мы создали, работает в Chrome, Firefox и Safari. Остается только один *современный* браузер: Microsoft Edge.

На данный момент в этой книге мы рассмотрели практически все возможные концепции веб-компонентов. Наши знания позволили нам соз-

дать веб-компоненты с помощью только пользовательских элементов, в завершение мы применили удивительную теневую модель DOM.

Есть веская причина, по которой мы решаем вопросы в таком порядке, и она состоит в том, что будут ситуации, когда вы просто не сможете или у вас не будет желания использовать теневую модель DOM. Я рад сообщить, что такие ситуации случаются все реже и реже! Конец 2018 года принес нам отличные новости на этом фронте. Веб-компоненты стали поддерживаться в Firefox, в результате чего Edge является единственным популярным браузером, которого мы терпеливо ждем. Мы знали, что команда Microsoft Edge была занята работой над поддержкой веб-компонентов, но затем, в апреле 2019 года, команда выпустила предварительную версию для разработчиков на базе Chromium. Помимо разнообразия браузеров, это выглядит как отличная новость для веб-компонентов, потому что новая версия Edge поддерживает веб-компоненты так же, как и Chrome (не беспокоясь о странных вещах, которые Microsoft реализовала немного по-другому).

Общая картина здесь заключается в том, что в настоящее время существуют два популярных браузера, которые изначально не поддерживают веб-компоненты: pre-Chromium Edge и IE11. Для некоторых счастливчиков, которые занимаются веб-разработкой, эти браузеры не имеют значения. В случае с IE11 это связано с тем, что на компьютерах, где не установлена Windows10, его срок службы уже истек. На более новых компьютерах с Windows 10 Microsoft рекомендует использовать Edge, несмотря на то что доступен IE11. Что касается Edge, сейчас легко предположить, что это всего лишь вопрос нескольких месяцев, прежде чем большинству обычных пользователей будет доступна версия браузера, которая имеет те же возможности, что и Chrome.

Однако не всем из нас так повезло. IE11 по-прежнему остается проблемой для многих веб-разработчиков. Pre-Chromium Edge также может существовать некоторое время, пока пользователи медленно обновляются.

Независимо от причины, при создании компонентов полезно иметь план действий для решения этих проблем. Итак, в этой главе мы возьмем реальный компонент пользовательского интерфейса из предыдущей главы и немного поработаем над ним, чтобы он работал на Edge, используя полизаполнение и некоторые небольшие изменения. Наконец, мы поговорим о конкретных инструментах сборки, чтобы заставить наш компонент работать в IE11.

## 12.1 Обратная совместимость

Итак, вы ждете поддержки? Несмотря на то что версия Edge для разработчиков на базе Chromium доступна уже сегодня, сколько еще времени пройдет, прежде чем он будет выпущен для всех пользователей Windows? Сколько времени потребуется текущим пользователям, чтобы обновиться до последней версии? В настоящее время на эти вопросы нет подходящих ответов, поэтому стоит обсудить стратегию, которая за-

ставит палитру цветов из предыдущей главы работать в текущей версии Edge. Благодаря этому мы пройдем большую часть пути к поддержке со стороны IE11, если вам абсолютно необходим этот браузер. Для IE11 существует этап сборки/транспиляции, но сейчас давайте сосредоточимся на гипотетическом современном браузере, который не поддерживает веб-компоненты.

Одним из замечательных ресурсов, которые помогут в этом, являются различные полизаполнения, представленные на сайте www.webcomponents.org/polyfills. Честно говоря, я не такой уж и большой поклонник полизаполнения Shadow DOM. Это слишком похоже на магию, то есть за кулисами совершается множество вещей, о которых вы даже не подозреваете, например копирование и перезапись элементов DOM вашего компонента с использованием различных уникальных классов. Все было бы хорошо, если бы полизаполнение обрабатывало все без проблем и не имело ограничений. Реальность такова, что даже при использовании полизаполнения теневой модели DOM вам действительно нужно знать об ограничениях, которые возникают, когда теневая модель DOM недоступна изначально, и обходить их. Учитывая это, мы можем сделать несколько изменений, чтобы включать и выключать теневую модель DOM для нашего компонента, что будет делать его совместимым с Edge, не прибегая к полизаполнению этой функциональной возможности.

Несмотря на то что полизаполнения теневой модели DOM следует избегать, первым шагом является полизаполнение еще одного аспекта веб-компонентов: пользовательских элементов. По сути, это готовое решение. Когда мы добавляем полизаполнение в наш компонент, нам не нужно беспокоиться о предупреждениях или неподдерживаемых функциях. Пользовательские элементы будут работать только в тех браузерах, которые пока еще поддерживают их.

Полизаполнение можно найти по адресу https://github.com/webcomponents/custom-elements. Согласно документации, вы можете создать его самостоятельно, установить из NPM или, как мы это сделаем сейчас, просто использовать его из сети доставки содержимого (CDN). Чтобы действовать досконально, мы должны добавить полизаполнение во все наши три файла demo.html, чтобы все они работали. Просто добавьте в каждый из них ссылку на сценарий – в файле index.html, например:

```
<title>Color Picker Component</title>
<link rel="stylesheet" type="text/css" href="vars.css">
<script type="module" src="components/colorpicker/colorpicker.js"></script>
<script src="https://unpkg.com/@webcomponents/custom-elements"></script>
```

### 12.1.1 Включение теневой модели DOM

Если теневая модель DOM использовалась, но затем была отключена, одна из замечательных вещей заключается в том, что корень теневого дерева не был создан; вместо этого вы можете вернуться к области видимости вашего компонента (`this`). Это отлично работает, потому что со свойством `shadowRoot` можно взаимодействовать так же, как и с компо-

нентом. Это означает, что с точки зрения использования JS с целью взаимодействия с ними ни один из ваших кодов не нужно менять, если вы используете простое свойство для взаимозаменяемого представления любой из областей видимости.

Основным исключением здесь является то, что мы рассмотрели ранее. Это использование конструктора для выполнения тяжелой работы по инициализации. Помните, что при использовании теневой модели DOM вы создаете отдельную мини-модель DOM внутри своего компонента. Итак, учитывая, что вы создаете ее прямо в конструкторе, эта мини-модель становится доступной мгновенно. Когда теневая модель DOM не используется, вы полагаетесь на DOM, предоставленный HTML-страницей, на которой вы находитесь. Доступа к этой модели DOM в функции-конструкторе пока нет, поэтому функция `connectedCallback` – лучшее место для взаимодействия с DOM, например для получения и установки атрибутов и настройки значения для свойства компонента `innerHTML`.

Прежде чем мы перейдем к обходному пути, есть вероятность, что вы занимаетесь разработкой, используя браузеры Chrome, Firefox или Safari. Вместо того чтобы переходить на Edge для тестирования вещей, которые не поддерживаются веб-компонентами, вы можете выполнять бóльшую часть работы в своем любимом браузере, создав переключатель для класса, который включает и отключает теневую модель DOM. Он будет очень хорошо имитировать Edge, и вы можете просто выполнить надлежащее тестирование в этом браузере, когда закончите.

Используя ползунок в качестве исходного примера, мы добавим статический метод чтения, чтобы контролировать, подключаемся ли мы к теневой модели DOM:

```
export default class Slider extends HTMLElement {
   static get USE_SHADOWDOM_WHEN_AVAILABLE() { return false; }
```

Мы сделаем это в файле components/slider/slider.js, а также в двух других компонентах, находящихся в файлах components/coordinatepicker/coordinatepicker.js и components/colorpicker/ colorpicker.js.

С помощью этого переключателя мы можем теперь обратить наше внимание на конструктор. Помните, что здесь мы не можем взаимодействовать с DOM, если не используем теневую модель DOM, поэтому нам придется кое-что переделать. В листинге 12.1 показано, с чего мы начали, а в листинге 12.2 – как это можно изменить, чтобы включать и выключать DOM.

**Листинг 12.1 Компонент ползунок до того, как включение теневой модели DOM станет возможным**

```
constructor() {
   super();
   this.attachShadow({mode: 'open'});
   this.shadowRoot.innerHTML = Template.render();
   this.dom = Template.mapDOM(this.shadowRoot);
```

Создаем корень теневого дерева

Визуализация HTML-кода и стилей в innerHTML

Слушатели событий

```
    document.addEventListener('mousemove', e => this.eventHandler(e));
    document.addEventListener('mouseup', e => this.eventHandler(e));
    this.addEventListener('mousedown', e => this.eventHandler(e));
}
```

Чтобы изменить это, можно переместить кое-какой код в конструктор и создать (или не создавать) теневую модель DOM.

**Листинг 12.2 Активация переключателя для теневой модели DOM**

```
constructor() {
    super();

    if (Slider.USE_SHADOWDOM_WHEN_AVAILABLE &&
        this.attachShadow) {
        this.root = this.attachShadow({mode: 'open'});
    } else {
        this.root = this;
    }

    document.addEventListener('mousemove',
        e => this.eventHandler(e));
    document.addEventListener('mouseup', e => this.eventHandler(e));
    this.addEventListener('mousedown', e => this.eventHandler(e));
}

connectedCallback() {
    if (!this.initialized) {
        this.root.innerHTML = Template.render({
            useShadowDOM: Slider.USE_SHADOWDOM_WHEN_AVAILABLE &&
        this.attachShadow });
        this.dom = Template.mapDOM(this.root);
        this.initialized = true;

        if (this.backgroundcolor) {
            this.setColor(this.backgroundcolor);
        }
        if (this.value) {
            this.refreshSlider(this.value);
        }
    }
}
```

Если выбрано использование теневой модели DOM и она поддерживается, создаем корень теневого дерева; в противном случае устанавливаем ссылку на компонент (this)

Слушателям событий не нужно перемещаться, потому что компонент и документ доступны из конструктора

Функция connectedCallback может появляться несколько раз (всякий раз, когда компонент добавляется на страницу), поэтому убедитесь, что инициализация происходит только один раз

Указываем шаблонному модулю HTML/CSS, используется ли теневая модель DOM

Обновляем компонент на базе текущих атрибутов

Самое первое, что мы здесь делаем, – создаем свойство для класса под названием `this.root`. При использовании теневой модели DOM установите для этого свойства значение корня теневого дерева. В противном случае просто установите его как ссылку на наш компонент (`this`). Теперь мы можем использовать `this.root` везде, где нам нужно манипулировать содержимым нашего компонента, независимо от того, используем мы теневую модель DOM или нет.

На самом деле нам не нужно перемещать слушателей событий. Мы бы могли это сделать, если бы они были более специфичными. Например, если бы мы создали слушателя событий для рукоятки или какого-то эле-

мента, которого еще нет в DOM, здесь он бы не работал. В этом примере просто так получается, что то, что мы слушаем, – документ и сам компонент – доступно изначально. Код инициализации перемещен в новую функцию `connectedCallback`, но помните, что этот обработчик запускается каждый раз, когда компонент добавляется на страницу. Чтобы сделать компонент действительно надежным, мы должны проверить, был ли он уже инициализирован, с помощью специального свойства `this.initialized`, и запускать код, только если он еще не был запущен. Для насущных потребностей при работе с палитрой цветов эта проверка нам не нужна, но, опять же, если мы хотим, чтобы компоненты работали в различных ситуациях, это должно стать приоритетным.

Работать с нашим модулем импорта `Template` довольно просто. Вместо того чтобы установить для `shadowRoot.innerHTML` значение в виде HTML/CSS-строки, возвращаемой из импорта, мы просто используем `this.root.innerHTML`. Является `this.root` корнем теневого дерева или компонентом, он будет работать независимо от этого. Аналогично, при получении кешированных ссылок на элементы с помощью `Template.mapDOM` `this.root` работает независимо от того, какую ссылку он содержит.

Наконец, мы должны добавить еще кое-что касательно наших атрибутов. Стратегия рефлексии (атрибуты/геттеры/сеттеры) не меняется, но здесь существует проблема синхронизации. Когда мы использовали теневую модель DOM, мы могли инициализировать все, включая рендеринг всего нашего HTML-кода, получение ссылок на элементы и т. д. в конструкторе. К тому времени когда метод `attributeChangedCallback` запустится с нашими начальными атрибутами, мы будем готовы к работе. Однако теперь он срабатывает перед обработчиком `connectedCallback`, поэтому наши изменения теряются без возможности отреагировать.

На самом деле нам нужно обеспечить безотказность этого метода. Хуже, чем потеря этих изменений, может быть получение ошибки. Поскольку эта функция обратного вызова приводит к запуску кода, который изменяет рукоятку и фон, которые еще не существуют, приведенная ниже строка, например, выдаст ошибку при запуске компонента:

```
refreshSlider(value) {
    this.dom.thumb.style.left = (value/100 * this.offsetWidth -
     this.dom.thumb.offsetWidth/2) + 'px';
}
```

Чтобы решить эту проблему, можно просто проверить, был ли компонент уже установлен в `attributeChangedCallback`, и выйти, если этого не было:

```
attributeChangedCallback(name, oldVal, newValue) {
   if (!this.dom) { return; }
```

Но тогда, конечно, начальные атрибуты нашего компонента не использовались из-за этой проблемы синхронизации, поэтому мы проверили, присутствовали ли они, и работали с ними в последних нескольких строках листинга 12.2.

Хотя мы только что сфокусировались на компоненте ползунка, два других компонента можно изменить точно так же. Я не буду здесь все описывать, но вы можете заполнить их самостоятельно, и это будет неплохим упражнением. Если возникнет проблема, эти компоненты в готовом виде можно найти в репозитории для данной книги на сайте GitHub.

Тем не менее в конкретной реализации палитры цветов есть одно крошечное соображение. Я имею в виду обработчика `onMutationChange` из файла components/colorpicker/colorpicker.js:

```
onMutationChange(records) {
  records.forEach( rec => {
      this.data = Handlers.update({
```

Здесь мы обрабатываем любые изменения атрибутов наших внутренних элементов DOM. Первоначально мы наблюдали за изменениями атрибутов в `shadowRoot` и всех элементов внутри него. Теперь мы просто слушаем изменения в `this.root`. Когда теневая модель DOM не используется, мы наблюдаем за изменениями атрибутов в самом компоненте! Проблема в том, что мы уже делаем это с помощью функции `attributeChangedCallback`. Поэтому теперь мы дважды слушаем события и реагируем на них. Чтобы решить эту проблему, мы просто проигнорируем изменения атрибутов, поступающие от компонента внутри обработчика `onMutationChange`:

```
onMutationChange(records) {
  records.forEach( rec => {
      if (rec.target !== this) {
```

Здесь мы просто говорим, что если целевой элемент, идентифицируемый записью об изменении (каждая запись – это изменение, записанное MutationObserver), не является палитрой цветов, делаем все как обычно. Если целевой элемент *является* палитрой цветов, никаких действий не предпринимается.

## 12.1.2 Сравнение с полифилами

Хотя было не слишком сложно позволить компонентам работать без теневой модели DOM, это не банальность. Нельзя просто применить полифил и пойти дальше. Фактически единственная вещь, которую полизаполнение могло бы дать нам здесь, – это возможность продолжать использовать `this.shadowRoot` в компоненте, а также оно могло бы предложить инкапсуляцию, чтобы не дать стороннему JS-коду манипулировать DOM компонента, как это делала бы настоящая теневая модель DOM. Если для вас это важно, возможно, стоит обратить внимание на полизаполнение ShadyDOM (https://github.com/webcomponents/shadydom).

Остальная часть работы, которую мы проделали, особенно в том, что касается разбиения конструктора для перемещения инициализации в функцию `connectedCallback`, должна быть выполнена независимо от этого. Этот аспект, вероятно, объясняет, почему спецификация W3C рекомендует вообще не иметь подобного кода инициализации в конструкто-

ре (даже когда все остальные похоже игнорируют это правило). Гораздо проще настроить все правильно с самого начала и отключить теневую модель DOM, если это необходимо. Не имеет значения, если ваши целевые браузеры изначально поддерживают веб-компоненты, но, когда они этого не делают, полезно запускать свой компонент с учетом этих передовых методик.

### 12.1.3 Shadow CSS и дочерние элементы

Вероятно, самая раздражающая часть возвращения в мир без теневой модели DOM – это HTML-код и стили CSS. При создании нашей HTML-разметки я был слишком увлечен и использовал идентификаторы вместо классов для ссылки на элементы. Опять же, полизаполнение нас здесь не спасет. Используя шаблон компонента-ползунка в качестве примера (components/slider/template.js), нам просто нужно войти и убрать все идентификаторы. Это изменение показано в приведенном ниже листинге:

**Листинг 12.3 Изменение ID-ссылок на классы**

```
mapDOM(scope) {
   return {
       // OLD //
       overlay: scope.getElementById(
       'bg-overlay'),
       thumb: scope.getElementById('thumb')

       // NEW //
       overlay: scope.querySelector(
       '.bg-overlay'),
       thumb: scope.querySelector('.thumb')
   }
},

html() {
   // OLD //
   return `<div id="bg-overlay"></div>
           <div id="thumb"></div>`;
   // NEW //
   return `<div class="bg-overlay"></div>
   <div class="thumb"></div>`;
},
```

С теневой моделью DOM мы могли бы спокойно выполнять запросы по ID

Без теневой модели DOM это больше небезопасно, поэтому мы должны переключиться на классы

Элементы, использующие идентификаторы для ссылки ранее

Переходим на использование классов, если теневая модель DOM отсутствует

Для ограниченного контекста нашего компонента на демонстрационной странице нам вообще-то не нужен этот этап. Просто так получилось, что ни один из идентификаторов, которые мы использовали, не конфликтовал – все они были уникальными. Поэтому, если вы пропустили этот шаг, ничего страшного; все будет работать нормально. Проблема состоит в том, что если бы мы продолжали ссылаться по идентификатору и забыли об этом, это стало бы бомбой замедленного действия в наших руках. Использование данного компонента в более крупном приложе-

нии с другими ID-ссылками может переопределить то, какой элемент здесь возвращается, если два и более элементов используют один и тот же идентификатор, что может иметь серьезные (и таинственным образом действующие) последствия.

И снова у нас есть пример передовой методики, и нам нужно беспокоиться, только если мы планируем использовать наши компоненты в контексте, где нет теневой модели DOM. Если это возможно, лучше всего вообще избегать использования идентификаторов. Если это невозможно – ну, честно говоря, я наслаждаюсь роскошью использования идентификаторов, чтобы делать то, для чего они и были придуманы: ссылаться на уникальные элементы!

Последнее препятствие, которое нужно преодолеть, – это CSS. В этом случае полифил ShadyCSS действительно помогает, но в нем есть много всего, и мне кажется, что в таких случаях это того не стоит. Проблема состоит в том, что селектора `:host` не существует. На самом деле в Edge он фактически ломает ваши стили, если вы даже пытаетесь его использовать! Кроме того, простые автономные селекторы, такие как `.thumb`, которые раньше работали только в вашей теневой модели DOM, теперь могут влиять на все ваше приложение.

Полифил ShadyCSS подходит для этого как нельзя лучше. Вы, будучи разработчиком, несете ответственность за размещение своей разметки и стилей в теге `<template>`. Затем вступает в дело полифил и переписывает ваши элементы и стили для использования уникальных селекторов, поэтому кажется, что теневая модель DOM по-прежнему работает. Я склонен думать, что требуемая здесь настройка представляет собой то же самое или даже большее усилие, чем простое решение проблем. Да, теневая модель DOM обеспечивает защиту от проникновения стилей в наш компонент, но полифил этого не делает. Поэтому в действительности его использование не несет особой пользы, когда можно сделать что-то более простое.

Здесь нам пригодится применение шаблонных литералов. Вернитесь в класс компонента, где мы вызываем метод `Template.render`:

```
this.root.innerHTML = Template.render({ useShadowDOM:
   Slider.USE_SHADOWDOM_WHEN_AVAILABLE && this.attachShadow });
```

Здесь передача логического значения указывает функции `render` на то, используем мы теневую модель DOM или нет, а затем мы можем изменить стили, чтобы использовать соответствующие селекторы. Например, если мы изначально применяли `:host` в качестве селектора, теперь мы должны использовать имя компонента. В частности, в случае с ползунком

```
:host { . . . } becomes wcia-slider { . . . }

:host .thumb { . . . } or .thumb { . . . } becomes wcia-slider .thumb { . . . }
```

Учитывая это и сосредоточив внимание на шаблонном модуле ползунка (components/slider/template.js), мы можем создать в приведенном ниже листинге код, чтобы использовать тот или иной селектор.

**Листинг 12.4 Переключение между селекторами теневой и нетеневой моделей DOM**

```
render(opts) {
   return `${this.css(opts.useShadowDOM)}
           ${this.html()}`;
},

createHostSelector(useshadow, host) {
   if (useshadow) {
       return ':host';
   } else {
       return host;
   }
},

css(useShadowDOM) {
   const comp = 'wcia-slider';
   return `<style>
               ${this.createHostSelector(
                      useShadowDOM, comp)} {
                  display: inline-block;
                  position: relative;
                  border-radius: var(--border-radius);
               }

               ${this.createHostSelector(useShadowDOM, comp)} .bg-overlay {
                   width: 100%;
                   height: 100%;
                   position: absolute;
                   border
```

Передаем логическое значение функции css, чтобы указать, используется ли теневая модель DOM

Возвращаем соответствующую строку селектора для использования теневой или нетеневой модели DOM

Объявляем тег компонента, чтобы использовать его при генерации селектора

Динамически создаем селектор в зависимости от того, используется ли теневая модель DOM и название компонента

Точно то же самое можно сделать в инструменте для выбора координат и палитре цветов. Однако в палитре цветов есть один селектор, который немного отличается:

```
:host(.modal)
```

Помните, что этот селектор просто утверждает, что если у палитры цветов есть класс с именем `modal`, фон и будет стилизован соответствующим образом. Чтобы получить то, что нам нужно, без теневой модели DOM, нам понадобится следующий селектор:

```
wcia-color-picker.modal
```

В этом случае мы добавим дополнительную функцию для обработки этого случая в файле components/colorpicker/template.js, как видно из приведенного ниже листинга.

**Листинг 12.5 Обработка особого случая класса в компоненте**

```
createHostContextSelector(
       useshadow, host, clazz) {
   if (useshadow) {
       return `:host(${clazz})`;
```

Новая функция, которая принимает логическое значение Shadow DOM, имя компонента и класс для использования в компоненте

```
    } else {
        return host + clazz;
    }
},

css(useShadowDOM) {
    const comp = 'wcia-color-picker';
    return `<style>
                ...
                ${this.createHostContextSelector(useShadowDOM, comp, '.modal')}
                {
                    ${Modal.rules()}
                }
```

Создаем селектор :host(.modal) или wciacolor-picker.modal в зависимости от того, используется ли ShadowDOM

В качестве неплохого домашнего задания по JS можно придумать единую функцию, которая обрабатывает все варианты селектора `:host`, а затем встроить ее в базовый класс, из которого мы можем расширить шаблонный модуль любого веб-компонента. Опять же, когда мы смотрим в будущее, говоря о веб-компонентах, именно с этими видами оптимизации будет проделываться много интересной работы, и нам не понадобятся для этого новые функциональные возможности браузера!

## 12.2 Наименьший общий знаменатель

Как видите, при создании компонента, который потенциально может быть использован, когда нативные веб-компоненты недоступны, нужно много чего учесть. Хорошо, что с API пользовательского элемента так легко использовать полизаполнение, но на этом простота не заканчивается. Вероятно, становится очевидным, что компоненты и, собственно, веб-разработка в целом играют по разным правилам, используя или не используя теневую модель DOM.

При создании, будь то применение полифила или нет, вам нужно будет разработать свой компонент в качестве наименьшего общего знаменателя. Если вы не используете теневую модель DOM или не уверены, что применяете ее, то должны планировать свой компонент так, как если бы вы его не использовали. Также необходимо принять во внимание, что в отношении полифилов существует ряд серьезных оговорок. Самый захватывающий аспект теневой модели DOM – это инкапсуляция CSS, но полифилы не решают эту проблему. CSS-правила по-прежнему могут просачиваться. Они могут вытекать и из вашего компонента, если ваши селекторы не настроены должным образом, чтобы предотвратить подобную ситуацию, сделав их специфичными для вашего компонента. Повторюсь: не используйте просто `.thumb`; используйте `my-component .thumb`.

При подготовке компонента к переходу на нетеневую модель DOM было много сходства и повторяющегося кода. При рассмотрении этого кода в сочетании с повторяющимся кодом для рефлексии атрибутов и свойств в ваших компонентах может возникнуть искушение попробовать фреймворк или библиотеку.

LitElement (https://lit-element.polymer-project.org), созданная командой Polymer, обещает стать мощным базовым классом веб-компонентов, обеспечивающим все эти функции. Это определенно заставляет вас использовать несколько шаблонов разработки и расширяет API веб-компонентов с помощью дополнительных функциональных возможностей. Возможно, вы пытаетесь забыть о некоторых из этих проблем и ограничений, поэтому LitElement может быть прекрасным вариантом, особенно потому, что она обещает поддержку вплоть до IE11. StencilJS (https://stenciljs.com) от команды Ionic предлагает несколько иной подход. Разработчик может создать компонент с помощью этого фреймворка, и он будет скомпилирован в чистый веб-компонент.

Я уверен, что в будущем мы увидим еще больше решений и устойчивые версии LitElement и StencilJS. Лично я бы предпочел избегать этих решений в своих попытках уберечься от сложностей, связанных с фреймворками и библиотеками, используя только то, что нужно. Мне также нравится разрабатывать компоненты без шага сборки и компиляции до их выпуска, который оба этих решения применяют в процессе разработки.

В конце концов, вы должны просто использовать то, что подходит для вашего проекта. Тем не менее вся сложность, которую мы рассмотрели, не является необходимой при выполнении разработки для современных браузеров со встроенной поддержкой веб-компонентов. Надеюсь, что в скором времени Edge будет меньше беспокоить вас, если принять во внимание то, что все больше разработчиков не учитывают IE11, когда речь идет о требованиях к браузеру.

Но что происходит, когда нам нужно продвигать и поддерживать IE11? Полизаполнение пользовательских элементов по-прежнему работает, поэтому создание собственных элементов, как мы это делали, не является проблемой. Основной проблемой остается отсутствие поддержки новых функций JS, таких как `Class`. Чтобы позабыть об этом, необходимо выполнить транспиляцию и сборку! Мы поговорим об этом далее.

Конечно, всегда будут несоответствия, связанные с браузерами, которые необходимо устранить. На самом деле наша палитра цветов пока еще не очень хорошо работает в Edge. Чтобы закончить, давайте исправим эту проблему, чтобы палитра цветов прекрасно работала во всех современных браузерах. Вернитесь к классу компонента ползунка из файла components/slider/slider.js:

```
setColor(color) {
   this.dom.overlay.style.background = `linear-gradient(to right, ${color}
     0%, ${color}00 100%)`;
}
```

В этой функции можно использовать шестнадцатеричный код цвета прямо в линейном градиенте, когда прозрачность исчезает. Все другие современные браузеры поддерживают добавление двух дополнительных цифр для обозначения кода цвета из восьми символов. Последние две цифры обозначают прозрачность 0 %. К сожалению, в Edge это не

поддерживается. Нам нужно будет использовать цветовую модель RGBA и получить справочную информацию по преобразованию из модуля утилит `Color`, который мы можем импортировать.

**Листинг 12.6 Исправление линейного градиента для Edge**

```
import Template from './template.js';
import Color from '../colorpicker/color.js';

export default class Slider extends HTMLElement {

...

setColor(color) {
   const rgb = Color.hexToRGB(color);
   this.dom.overlay.style.background = `linear-gradient(to right,
     rgba(${rgb.r}, ${rgb.g}, ${rgb.b}, 1) 0%, rgba(${rgb.r}, ${rgb.g},
     ${rgb.b}, 0) 100%)`;
}
```

Изменяем правило стиля для IE/Edge

После добавления этих изменений наш компонент можно протестировать в любом современном браузере, включая Edge!

## 12.3 Процессы сборки

До сих пор мы все делали без использования какого-либо фреймворка и сложных рабочих процессов, которые выполняют кучу разной работы под капотом, о которой вы и не знаете. Был только браузер и немного HTML- и JS-кода и стили.

В случае со многими современными рабочими процессами в сети дело обстоит иначе. Вы не будете запускать один и тот же код в своем браузере, как вы пишете его в редакторе, и такое будет происходить не раз. Между ними может находиться шаг сборки. Существует множество причин для выполнения сборки, начиная от использования таких инструментов, как Sass и LESS для компиляции стилей, до генерации большого HTML-файла из различных фрагментов, которые вы рассортировали по нескольким разным файлам.

Я мог бы продолжить объяснять причины использования одного или нескольких шагов сборки, даже не говоря о JS. Подобные задачи внешнего интерфейса, будь то HTML, CSS или JS, почти всегда выполняются с помощью Node.js. Но какую конкретную систему нужно использовать? Среди основных систем, которые обещают сделать все это, можно упомянуть Grunt и Gulp, но даже более специфичные системы, которые обещают сделать что-то одно, имеют тенденцию перекрываться. Например, Webpack предназначен для компоновки ресурсов, но в случае со множеством задач он может перекрываться с теми задачами, которые Grunt и Gulp могут выполнять самостоятельно.

Когда существует сообщество веб-разработчиков, выпускающее новые инструменты каждый день, и множество действительно надежных систем сборки, которые могут все это делать, может возникнуть путаница отно-

сительно того, какие инструменты нужно включить в свой набор и какие системы использовать для управления всем этим. В последнее время все же наблюдается уклон в сторону простого варианта, когда это возможно.

### 12.3.1 Использование сценариев NPM

Прежде чем углубляться и выяснять, почему у нас может возникнуть желание включить процесс сборки в свой рабочий процесс веб-компонента, давайте поговорим о простом способе запуска задач. Вы, вероятно, использовали Node.js, даже если только для установки чего-либо. Чтобы освежить вашу память, напомню, что `npm` – это часть экосистемы Node.js для установки выбранного вами пакета JS.

Например, если вы хотите установить полизаполнения веб-компонента, перейдите в корневой каталог вашего проекта, запустите терминал и выполните команду

```
npm install @webcomponents/webcomponentsjs
```

Этот пакет будет установлен в корне вашего проекта в папке node_modules. Конечно, по мере того как вы добавляете все больше и больше пакетов, можно легко запутаться, поэтому вам понадобится какая-то запись, которая будет отслеживать ваши зависимости, подобно этой, равно как и другие детали вашего проекта. Вот для чего существует файл package.json. Создать новый такой файл с нуля легко. Снова в терминале, в корне своего проекта, выполните команду

```
npm init
```

Вам нужно будет ответить на несколько вопросов, чтобы предоставить информацию по своему проекту, такую как имя, адрес электронной почты, название пакета и т. д. Когда файл package.json будет готов, если вам нужно будет выполнить предыдущую команду для установки `webcomponentsjs`, она будет добавлена в список зависимостей в JSON.

Или, если это зависимость, предназначенная только для рабочего процесса разработчика вашего проекта, и это не часть вашей рабочей версии, нужно выполнить команду

```
npm install @webcomponents/webcomponentsjs --save-dev
```

Помимо зависимостей, у файла package.json есть еще один довольно мощный аспект. Мы можем добавить объект `script` для запуска всего, что нам нужно. Можно попробовать запустить что-то простое, и это довольно легко.

**Листинг 12.7 Простой скрипт для файла package.json**

```
{
 "name": "wcia",
 "version": "1.0.0",
 "scripts": {
   "test": "echo 'Hello from package.json'"  <-- Скрипт для запуска
 }
}
```

По сути, все, что можно запустить в терминале, можно добавить сюда. В простом тесте, приведенном в листинге 12.7, используется команда Linux `echo`, которая выводит любое сообщение, которое вы ей даете, в виде строки в вашем терминале. Пользователям Windows также не нужно чувствовать себя обделенными благодаря Windows Subsystem for Linux (WSL; https://docs.microsoft.com/en-us/windows/wsl/install-win10). При этом пользователи Windows могут выполнять те же команды Linux, что и пользователи Mac или Linux. Даже до появления WSL, которая определенно не идеальна, простая установка Git для Windows (https://gitforwindows.org) позволяла использовать ограниченный набор команд Bash, которого вполне могло быть достаточно.

Причина этого заключается в том, что сценарии `npm` все чаще и чаще становятся частью рабочего процесса разработчика вместо больших, сложных систем сборки, таких как Grunt или Gulp. Когда речь идет о сложных и многочисленных задачах как части рабочего процесса, в таких системах сборки нет ничего плохого. Однако при выполнении нескольких простых задач вся эта сложность не нужна. Системами сборки легко овладеть. Выполнение множества различных задач потребует изучения необходимых плагинов и устранения недостатков, когда они не работают вместе, но это также означает, что вам не нужно прописывать каждую небольшую задачу, такую как копирование файлов, запуск препроцессоров CSS, загрузка на сервер, конкатенация файлов, HTML-шаблонирование и т. д. Но если вам нужно всего несколько задач, и их очень легко написать самостоятельно, нет никаких причин, по которым вы можете отказаться от простого пути.

В следующих двух главах мы рассмотрим несколько основных способов выполнения сборки и тестирования. Хотя сами инструменты сборки и тестирования имеют различные уровни сложности, команды для их запуска невероятно просты. Даже если вы работаете в Windows без вышеупомянутого WSL и просто используете эмуляцию Git Bash, команды, которые мы запускаем, будут работать с одним предостережением при запуске тестов, о чем я упомяну в следующей главе. Следовательно, здесь мы будем избегать систем сборки при изучении соответствующих процессов, что позволит нам сосредоточиться на конкретных задачах, которые мы выполняем, избегая при этом множества настроек, не имеющих прямого отношения к тому, что нам нужно для запуска. Самое главное – вам решать, какую систему сборки выбрать, если вы захотите ее использовать.

## 12.4 Сборка компонентов

На самом деле веб-компоненты ничем не отличаются от других понятий с точки зрения того, как и почему мы будем использовать этап сборки для нашего JS-кода. И так же, как и во всех остальных случаях, сложность может возрастать по мере роста потребностей нашего проекта или компонента. Что нам пока не ясно, так это зачем вообще выполнять сборку.

### 12.4.1 Почему мы выполняем сборку

Существует множество причин для запуска процесса сборки JS. Одна из получающих все большее распространение причин заключается в том, что разработчик может предпочесть другой язык, помимо JS, для написания кода. Много лет назад CoffeeScript был популярным языком для написания веб-приложений, хотя в наши дни TypeScript от компании Microsoft является наиболее популярным языком для создания веб-приложений. Однако TypeScript – это не полностью другой язык – это надмножество JS с добавлением типизированных переменных. Он также предлагает новейшие функции JS, которые еще не вошли в браузеры. На самом деле у издательства Manning есть две действительно солидные книги по TypeScript, которые недавно вышли в свет:

- Angular Development with TypeScript (https://www.manning.com/books/angular-development-with-typescript-second-edition);
- TypeScript Quickly (https://www.manning.com/books/typescript-quickly).

Также TypeScript становится все более и более актуальным языком для работы веб-компонентов. Помимо того что это популярный язык для работы в целом, нужно отметить, что на нем написаны проекты LitElement и lit-html от команды Polymer. Хотя написание кода с использованием новых языковых функций, таких как декораторы, не обязательно, это настоятельно рекомендуется, поскольку большинство примеров написаны именно так.

Есть не только CoffeeScript и TypeScript – также существует огромное количество языков, которые разработчики используют для запуска кода в интернете. Однако все эти языки имеют одну общую черту. В действительности они не запускаются в браузере. Ваш код пишется на выбранном вами языке, но затем *транспилируется* в JS, чтобы он мог работать.

Если слово «транспиляция» звучит как чужеродная концепция, можно сказать, что она очень похожа на компиляцию. И то, и другое позволяют писать код и преобразовывать его в нечто, работающее на нужной вам платформе, например в сети. Компиляция означает, что вы нацелены на более низкий уровень абстракции, такой как байт-код. Вывод компилятора в основном нельзя прочитать.

Компиляцию можно рассматривать почти как если бы мы взяли устную речь и сохранили ее в виде звукового файла. Невозможно разобраться в том, кто что говорит, посмотрев на запись в своем любимом аудиоредакторе, но вы, безусловно, можете воспроизвести ее и абсолютно прекрасно понять, что говорят.

Транспиляцию можно рассматривать как перевод, например, с испанского на английский. Если вы ими владеете, на испанском и английском можно прекрасно читать тексты, а если нет, перевод поможет вам прочитать текст на вашем родном языке.

Транспиляция – это даже не написание чего-либо на совершенно новом языке. За прошедшие годы в JS было добавлено много новых и интересных функций, особенно в 2015 году, когда был выпущен стандарт ES6/

ES2015. Однако разработчики не могли сразу использовать эти функции. Даже если их любимый браузер поддерживал их, это относилось не ко всем браузерам. Даже сейчас, хотя основные современные браузеры имеют отличную поддержку функций ES6/ES2015, у разработчиков может возникнуть желание использовать более старые браузеры, такие как IE. Даже если это не так, существуют замечательные, совершенно новые функции языка JS или даже экспериментальные функции, которые разработчики хотят использовать и которые пока не имеют никакой поддержки со стороны браузеров. Для таких случаев Babel (https://babeljs.io), вероятно, является наиболее широко используемым на сегодняшний день JS-транспилятором.

Еще одна важная причина для изменения JS-кода – взять много разных исходных файлов и поместить их в один больший. Когда исходный код вашего приложения начинает расти, превращаясь в сотни или тысячи строк кода, не рекомендуется помещать весь свой JS-код в один большой файл. Во-первых, при работе в команде легче наступать друг другу на пальцы при изменении одного и того же файла. Во-вторых, ваш проект лучше организован, когда JS-код разделен правильно. Фрагменты функционала легче найти, когда вам не нужно отправляться на охоту, используя один огромный файл. Кроме того, когда они организованы в менее крупные файлы с подходящим названием, легче взглянуть на файловую структуру проекта и понять, что он делает и как все работает.

Несмотря на улучшения рабочего процесса разработчиков из небольших файлов, для браузера лучше, чтобы все было вместе в одном файле или, что еще лучше, было разумно объединено в файлы, которые загружаются, когда нужны функциональные возможности. Когда все объединено, браузеру нужно обрабатывать меньше сетевых запросов. Это важно, потому что у браузеров имеется максимальное количество запросов, которые можно выполнить одновременно. Кроме того, из-за состояния сети сценарии могут загружаться медленно. Вы можете начать представлять себе, какие странные вещи могут происходить с вашим приложением, когда для его загрузки имеется какой-то функционал, но не доступен другой скрипт, потому что сетевой запрос занимает слишком много времени.

До появления модулей ES6/ES2015 и игнорирования аналогичных решений, таких как RequireJS, JS-код в отдельных файлах просто связывался воедино с помощью средств конкатенации. По сути, конкатенация – это просто помещение содержимого каждого JS-файла в более крупный файл в указанном вами порядке. Мы по-прежнему делаем нечто подобное, но в случае с модулями все должно быть немного умнее. Автоматизированные инструменты должны пройти через ваш код, отследить модули, на которые вы ссылаетесь, используя ключевое слово `import`, и связать их, чтобы представить окончательный вывод. Есть и более умный способ. Некоторые инструменты используют метод под названием *встряхивание дерева*. Если вы импортируете модуль и не используете его где-либо в своем коде, он не будет объединять этот конкретный модуль. Встряхивание

дерева – умный способ гарантировать наличие более мелких наборов, которые включают в себя только тот код, который вам нужен.

Такие инструменты, как Webpack (https://webpack.js.org), отличаются друг от друга еще больше, позволяя создавать несколько выходных наборов и компоновать больше типов файлов, нежели просто JS. Эти пакеты организованы по функциональности, которая необходима для запуска определенных областей вашего приложения. Веб-приложения могут быть огромными, и, возможно, стоит подумать над тем, чтобы разделить свое приложение на разные секции.

Например, если вы работаете над веб-приложением для банковского сектора, пользователь может просматривать свои последние транзакции в одном разделе, но никогда не станет посещать другой раздел, чтобы увидеть информацию о своем аккаунте. При таком сценарии нет причин заставлять пользователя загружать набор, содержащий JS-код, связанный с информацией об учетной записи. Поэтому хотя банковское приложение может представлять собой один большой модуль, разумнее разделить его на несколько наборов для каждого раздела приложения. На рис. 12.1 показаны основные различия между более простыми инструментами, такими как Rollup, и более сложными, такими как Webpack.

И снова мы вернулись ко множеству инструментов, которые мы можем использовать! В любом случае, и транспиляция, и компоновка являются двумя основными мотиваторами процесса сборки для вашего JS-кода.

## 12.4.2 Компоновка модулей с помощью Rollup

Хотя существует множество инструментов для компоновки модулей, Webpack традиционно был довольно сложным в настройке для самых простых задач, в то время как Rollup был простой, но не настолько настраиваемой альтернативой. Недавние версии Webpack изменили то, насколько крутой является кривая обучаемости для выполнения простых вещей, в то время как новичок Parcel.js (https://parceljs.org) также приобрел популярность!

Нам просто нужно выбрать что-то одно, чтобы двигаться дальше; учитывая три этих замечательных варианта, я бы хотел выбрать Rollup (https://rollupjs.org), поскольку с ним у меня больше опыта и я ценю его за простоту, благодаря которой его можно быстро запустить и приступить к работе. Как и в случае с любой командой `npm install`, обязательно создайте файл package.json в корне своего проекта. Затем в терминале с помощью команды `cd` перейдите в корневой каталог вашего проекта и выполните команду

```
npm install --save-dev rollup
```

Обратите внимание, что здесь мы использовали параметр `--save-dev`. Rollup будет добавлен в ваш пакет.json в качестве dev dependency, т. е. зависимости, которая нужна вам только во время разработки. Это означает, что вы не собираетесь ничего делать с Rollup, кроме того что он поможет вам в процессе разработки и сборки. Это не код, который предна-

значен для поставки вместе с вашим компонентом. Когда вы закончите, ваш файл package.json будет выглядеть так, как показано в приведенном ниже листинге (конечно, в зависимости от того, как вы назвали свой проект и какую версию указали).

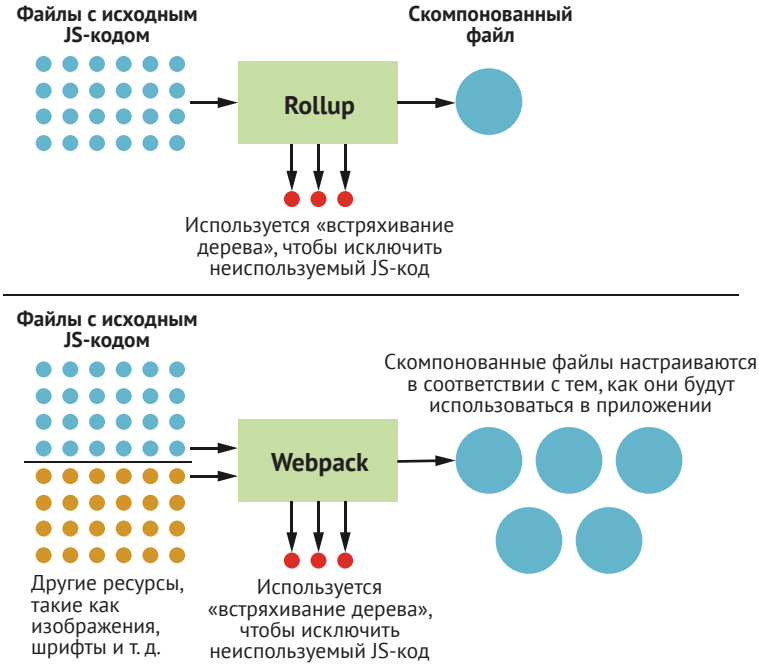



**Рис. 12.1 Rollup в сравнении с Webpack**

**Листинг 12.8 Файл package.json после установки Rollup**

```
{
 "name": "wcia",
 "version": "1.0.0",
 "dependencies": {},
 "devDependencies": {
   "rollup": "^1.0.2"
 }
}
```

Зависимость, необходимая во время разработки

Интересно отметить, что можно установить Rollup (или любой другой пакет в целом) глобально с помощью параметра `-g`:

```
npm install rollup -g
```

При глобальной установке Rollup можно запускать непосредственно со своего терминала, где угодно на компьютере, просто выполнив команду `rollup` с некоторыми параметрами. Здесь мы установили его локально, как часть проекта. При локальной установке Rollup по-прежнему можно запускать на терминале с помощью сокращенной команды `rollup`,

так как путь установки, вероятно, будет добавлен к вашим переменным среды. Я до сих пор не верю этому! Если бы у вас было несколько разных установок Rollup в разных проектах, вы бы гадали, где вы на самом деле используете его. Вместо этого мне нравится действовать немного точнее и выполнять его из моего проекта: node_modules/.bin/rollup. Выглядит немного сложнее, но такой вариант более распространен, нежели глобальная установка.

Причина, объясняющая, почему так будет лучше, состоит в том, что если бы вы захотели, чтобы члены команды работали с вашим проектом и инструментами глобально, вам нужно было бы предоставить им рукописный список всего, что нужно установить для работы с вашим проектом, и они должны были бы устанавливать все это по одному. При наличии большого числа зависимостей некоторые вещи легко позабыть, и отлаживать причину неисправности в процессе сборки будет сложно. При локальной установке все, что нужно, находится непосредственно в файле package.json и может быть установлено за один раз с помощью команды `npm install`.

Сложно набирать на клавиатуре этот путь каждый раз, когда вы хотите запустить сборку. Команда становится еще длиннее, когда мы добавляем параметры, чтобы указать, где находится основная точка входа JS, где должен находиться выходной файл и какое у него имя. Вот почему мы можем сделать запись в нашем объекте сценариев package.json и добавить команду туда.

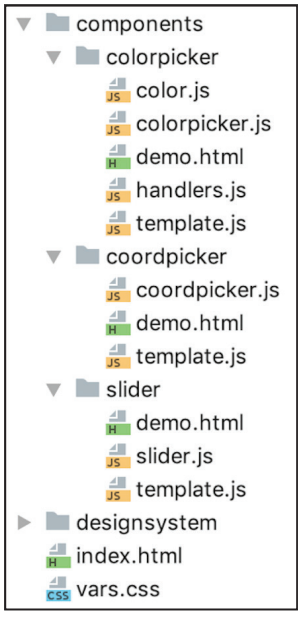



**Рис. 12.2 Файлы ползунка**

Однако, прежде чем мы это сделаем, мы должны совсем немного изменить структуру нашего веб-компонента. В качестве примера давайте начнем с ползунка из предыдущей главы, который был небольшой частью компонента палитры цветов. На рис. 12.2 показана его простая файловая структура вместе с другими компонентами и модулями дизайн-системы.

Опять же, хотя ползунок отлично работал в нашей локальной среде разработки (честно говоря, он настолько мал, что, вероятно, отлично работает и в сети), нам нужно создать набор, чтобы конечный пользователь загружал все модули (slider.js, template.js и все соответствующие куски нашей дизайн-системы). Эти файлы теперь следует рассматривать как исходные файлы, которые не используются напрямую конечными пользователями. Таким образом, мы создадим папку src в каталоге каждого компонента и поместим туда файлы slider.js и template.js. Мы сделаем это и для других компонентов. На рис. 12.3 показана новая структура папок.

Благодаря новой структуре папок входной файл для Rollup теперь находится по адресу component/slider/src/slider.js. Ничего в коде внутри этого файла не меняется, кроме одной маленькой детали. Хорошая новость заключается в том, что наши пути импорта в основном связаны с компонентом, поэтому их не нужно менять. Когда мы импортируем

модуль `Template`, он по-прежнему находится по адресу./template.js. Раздражает то, что когда мы исправили прозрачность для Edge, то использовали модуль Color из компонента палитры цветов. Поэтому теперь вместо

```
import Color from '../../colorpicker/color.js';
```

нужно использовать

```
import Color from '../../colorpicker/src/color.js';
```

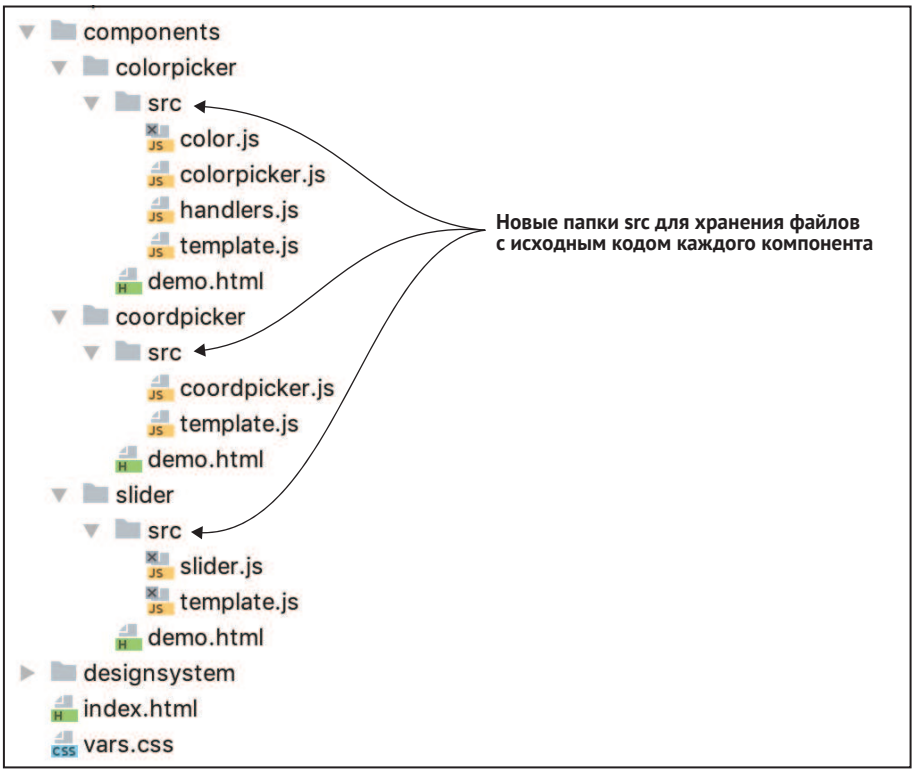



**Рис. 12.3 Ползунок и другие файлы компонентов с папкой src**

В конце концов, вывод может быть создан там, где когда-то был исходный файл slider.js. Эти два параметра являются основными для работы Rollup! Вот команда, которую мы будем выполнять:

```
./node_modules/.bin/rollup chapter12and13/components/slider/src/slider.
   js --file chapter12and13/components/slider/slider.js --format umd
   --name slider -m
```

Полный путь к каталогу включает в себя «chapter12and13», чтобы соответствовать репозиторию для этой книги на GitHub. Самый первый параметр – это расположение файла компонента ползунка с исходным кодом. В качестве единственного обязательного параметра этот первый параметр также является единственным, для которого флаг не требуется.

Во-вторых, нам нужно указать выходной файл, передаваемый перед параметром `--file`. Далее идет выходной формат, обозначаемый с помощью параметра `--format`. Здесь нет правильного ответа, но я предлагаю использовать универсальное определение модуля (UMD). При компоновке в виде UMD JS-код можно загружать различными способами. Два из них – CommonJS и асинхронное определение модуля (AMD), которые можно использовать в различных сценариях, в том числе и с RequireJS. Последний метод, который активирует UMD, заключается в простом глобальном определении, где не предполагается никаких механизмов загрузки JS. UMD прикрепляет ползунок к `window` в качестве глобальной переменной, доступной из любого места на вашей странице.

Какое имя у этой глобальной переменной? На этот вопрос легко ответить, используя параметр `--name`. Назовем нашу переменную `slider`. Теперь у нас есть `window.slider` в качестве глобальной переменной, но мы, вероятно, никогда не будем ее использовать, поскольку наш компонент настраивается автоматически. Возможно, вы захотите действовать более осторожно и используете имя, с которым в вашем приложении никогда не будет конфликтов. Пространство имен вашего компонента может быть хорошим кандидатом на включение в этот код, например `MyNamespaceSlider`, или можно использовать имя вашего приложения – просто что-то, чтобы сделать его уникальным.

Очевидный вопрос заключается в том, отказываемся ли мы от возможности использовать ползунок в качестве обычного модуля ES6/ES2015, как это было раньше. Нет! Если более крупное приложение, содержащее ползунок, хочет импортировать модуль, оно может с легкостью импортировать файл src/slider.js и использовать его, игнорируя сгенерированный набор. Это более крупное приложение может затем скомпоновать само приложение и все компоненты внутри него, используя Rollup или любой другой компоновщик модулей, который он предпочитает.

Самый последний флаг `-m` включает генерацию «карт кода». Если вы незнакомы с картами кода, они связывают сгенерированный вывод с оригинальными файлами с исходным кодом. Фрагмент «карты» представляет собой нечитабельный файл с расширением.map. Он содержит информацию для поиска, позволяющую осуществить это связывание. Это может звучать бессмысленно, пока не увидишь это в действии. Вы можете попробовать сами после запуска сборки, но на рис. 12.4 показаны карты кода в действии. В моем коде есть ошибка. Хотя мы используем выходной набор, который будем генерировать далее, мы видим точную строку, где произошла ошибка.

### 12.4.3 Запуск сборки с помощью npm

Теперь, когда мы знаем, как выполнять сборку с помощью Rollup, когда распланировали более подходящую файловую структуру компонента и знаем, чего ожидать от скомпонованного вывода, давайте сделаем процесс компоновки с помощью Rollup более простым. Как уже говорилось ранее, мы можем легко добавить команду компоновки в наш файл

package.json. Обычно достаточно чего-то простого. Можно просто назвать задачу `build` и двигаться дальше, как показано в приведенном ниже листинге.

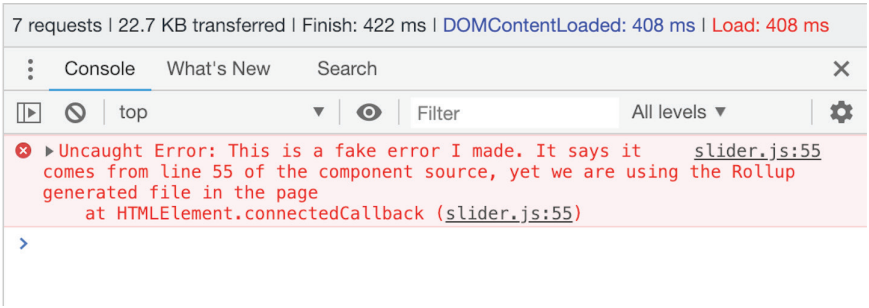



**Рис. 12.4 Карты кода показывают, где произошла ошибка в ваших исходных файлах, даже при компоновке вывода**

**Листинг 12.9 Добавляем скрипт Rollup в файл package.json**

```
{
 "name": "wcia",
 "version": "1.0.0",
 "dependencies": {},
 "devDependencies": {
   "rollup": "^1.0.2"
 },
 "scripts": {
"build": "./node_modules/.bin/rollup chapter12and13/components/slider/src/
   slider.js --file chapter12and13/components/slider/slider.js -format
   umd --name slider -m"
}
```

Скрипт сборки Rollup

Поэтому теперь, вместо того чтобы вводить длинную и сложную команду сборки, мы можем просто выполнить новую команду `build` в терминале в корне проекта:

```
npm run build
```

В идеале весь проект должен быть одним компонентом ползунка. Затем мы можем выполнить команду `npm install` и использовать его в любом проекте (например, в палитре цветов).

Однако, как я настроил проект палитры цветов для этой книги, все компоненты находятся в одном проекте (и в одной и той же папке главы 12). Таким образом, планирование стратегии, позволяющей учесть это, может показаться немного странным вызовом, но на самом деле это предоставляет удобный способ запуска сценариев.

Мы можем начать с добавления еще двух сценариев сборки в файл package.json, как показано в приведенном ниже листинге. Поскольку теперь их три, мы должны быть более точными относительно того, как мы их называем, а не просто «build».

**Листинг 12.10 Скрипты для запуска каждой сборки компонента**

```
{
 "name": "wcia",
 "version": "1.0.0",
 "dependencies": {},
 "devDependencies": {
   "rollup": "^1.0.2"
 },
"scripts": {
   "build-slider": "./node_modules/.bin/rollup
     chapter12and13/components/slider/src/slider.js --file
     chapter12and13/components/slider/slider.js --format umd
     --name slider -m",
   "build-coordpicker": "./node_modules/.bin/rollup
     chapter12and13/components/coordpicker/src/coordpicker.js --file
     chapter12and13/components/coordpicker/coordpicker.js --format umd
     --name coordpicker -m",
   "build-colorpicker": "./node_modules/.bin/rollup
     chapter12and13/components/colorpicker/src/colorpicker.js --file
     chapter12and13/components/colorpicker/colorpicker.js --format umd
     --name colorpicker -m"
}
```

Задача Rollup для инструмента выбора координат

Задача Rollup для палитры цветов

Теперь вы, наверное, думаете, что у нас есть три команды для запуска вместо одной, но мы можем комбинировать сценарии! Знак одиночного или двойного амперсанда не является чем-то, чему необходимо строго следовать при работе с `npm`. Это всего лишь стандартный Linux, и мы можем использовать амперсанды для объединения команд в скриптах package.json. Одиночный амперсанд запускает команды параллельно, а двойной запускает их одну за другой. Кроме того, мы можем ссылаться на другие скрипты по имени во всех новых командах. Мы добавим еще одну задачу сборки после того, как закончим с Rollup, поэтому давайте пока не будем вызывать этот новый скрипт `build`. Вместо этого мы назовем его `build-rollup`:

```
"build-rollup": "npm run build-slider && npm run build-coordpicker && npm
    run build-colorpicker"
```

Теперь все три компонента можно собрать, просто выполнив команду

```
npm run build-rollup
```

Однако обратите внимание, что если вы работаете в Windows, этот подход с амперсандами не будет работать без использования WSL, эмулятора Git Bash или чего-то подобного.

## 12.5 Транспиляция для IE

Я упомянул о дополнительном шаге сборки для наших компонентов. На данный момент палитра цветов и два дочерних компонента работают во всех популярных браузерах, включая Edge, если мы отключаем тене-

вую модель DOM. Как упоминалось ранее, Edge скоро будет обновлен до уровня Chrome и будет нативно поддерживать веб-компоненты.

У нас остается один проблемный браузер: IE11. Нас беспокоит его возраст и отсутствие обновлений. Современные браузеры обновляются автоматически, и веб-разработчикам обычно приходится беспокоиться только по поводу последних версий каждого браузера. Таким образом, обычно мы используем самые последние функциональные возможности в довольно короткие сроки, при условии что все браузеры идут в ногу друг с другом. IE здесь как бельмо на глазу. Поскольку IE11 является последней версией, которая когда-либо будет выпущена, мы застряли на функциях, имеющихся у него в настоящее время. Некоторые из нас, веб-разработчиков, оказались в состоянии игнорировать его в качестве необходимого требования, потому что его использование очень невелико, и сейчас компания Microsoft рекомендует пользователям Windows браузер Edge. Но не всем веб-разработчикам так повезло, и им он по-прежнему необходим.

IE не только не поддерживает веб-компоненты, в отличие от текущей версии Edge, но также и не поддерживает языковые функции ES6/ES2015, такие как классы и жирные стрелки. Ранее в этой главе мы обсуждали транспиляцию как способ делать что-то вроде перевода с одного языка на другой, например переводя TypeScript или CoffeScript в JS, но сейчас мы можем использовать ее для решения проблемы, связанной с IE, транспилируя более новый JS-код в более старый.

### 12.5.1 Babel

Самым популярным инструментом для решения этих проблем является Babel (https://babeljs.io). Нам понадобится установить несколько пакетов с помощью команды `npm install`, чтобы заставить Babel работать:

- `@babel/core` – основной набор функций Babel;
- `@babel/cli` – инструментарий для использования Babel в командной строке;
- `@babel/preset-env` – работа с Babel может стать сложной; пресет, который позволяет указать среду и автоматически активирует необходимые плагины.

Давайте продолжим и установим их как dev-зависимости в корне проекта, потому что, как и в Rollup, это всего лишь инструментарий для сборки, и они не будут частью выпуска компонента:

```
npm install --save-dev @babel/core

npm install --save-dev @babel/cli

npm install --save-dev @babel/preset-env
```

После установки, поскольку они были сохранены, эти зависимости добавляются в файл package.json. В приведенном ниже листинге показано, как он выглядит на данный момент.

**Листинг 12.11 Самая последняя версия файла package.json, включая зависимости Babel**

```
{
 "name": "wcia",
 "version": "1.0.0",
 "dependencies": { },
 "devDependencies": {
   "@babel/cli": "^7.2.3",
   "@babel/core": "^7.2.2",
   "@babel/preset-env": "^7.2.3",
   "rollup": "^1.0.2",
 },
 "scripts": {
   "build-slider": "./node_modules/.bin/rollup
     chapter12and13/components/slider/src/slider.js --file
     chapter12and13/components/slider/slider.js --format umd
     --name slider -m",
   "build-coordpicker": "./node_modules/.bin/rollup
     chapter12and13/components/coordpicker/src/coordpicker.js --file
     chapter12and13/components/coordpicker/coordpicker.js --format umd
     --name coordpicker -m",
   "build-colorpicker": "./node_modules/.bin/rollup
     chapter12and13/components/colorpicker/src/colorpicker.js --file
     chapter12and13/components/colorpicker/colorpicker.js --format umd
     --name colorpicker -m",
   "build-rollup": "npm run build-slider && npm run build-coordpicker && npm
     run build-colorpicker",
 }
}
```

Инструменты для работы с Babel в командной строке

Пакет @babel/core

Пакет @babel/preset-env для легкой настройки

После этой установки Babel очень прост в использовании. Опять же, как и в Rollup, поскольку мы установили локальную, а не глобальную зависимость, исполняемый файл Babel можно найти в файле node_modules/.bin/babel.

Однако Babel не решает вопрос компоновки модулей. Для этого нам нужен дополнительный этап. Для того чтобы позаботиться об этом в рамках Rollup, существуют плагины. Однако здесь мы осмеливаемся зайти на территорию, где все не так просто и зависит от потребностей вашего проекта. Я считаю, что для этих компонентов мы должны создать для IE сборку, отличную от той, что используется для современных браузеров. Причина, по которой я думаю, что у нас должны быть разные сборки, заключается в том, что нет необходимости перегружать современные браузеры громоздким транспилируемым кодом, когда для этого нет причин. Но, возможно, наличие нескольких сборок вредит вам и вашей команде, когда речь идет о простоте доставки компонентов. В конечном счете выбор за вами, но сейчас я принимаю решение о поставке двух версий.

Поскольку набор Rollup уже существует, мы можем просто использовать его в качестве предварительно скомпонованного источника, который подается в Babel, при условии что будем осторожны и сначала

выполним его сборку. Если вы придерживаетесь другого мнения и считаете, что эти компоненты будут лучше обслуживаться одним выходным файлом, можно настроить Rollup для добавления этого шага с дополнительными настройками. В действительности все зависит от вашего варианта использования и от того, как будет применяться ваш компонент. На рис. 12.5 показан наш вариант конвейера сборки.

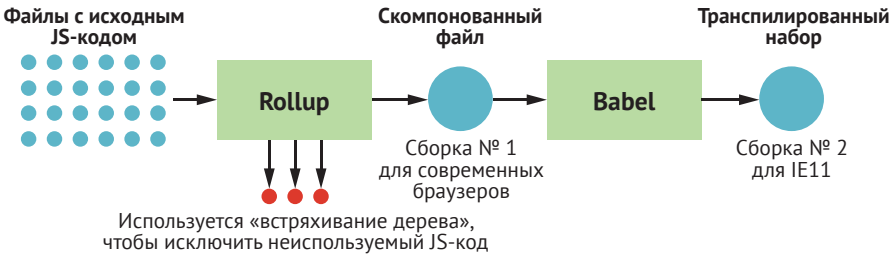



**Рис. 12.5 Конвейер сборки палитры цветов включает в себя две сборки: одну для современных браузеров и другую для IE11**

Однако для использования настроек `preset-env` необходим конфигурационный файл Babel. Хотя он и довольно простой. В корне проекта просто создайте файл.babelrc следующего содержания:

```
{
 "presets": ["@babel/preset-env"]
}
```

Эта последняя часть – настройки, необходимые для запуска транспиляции Babel. Мы просто говорим ему использовать предустановленные настройки Babel в одной строке. Затем, чтобы выполнить команду с этим параметром, вам нужно просто запустить команду Babel, используя входной и выходной файлы:

```
./node_modules/.bin/babel chapter12and13/components/slider/slider.js
   --out-file chapter12and13/components/slider/slider.build.js
```

Первый параметр – это входные данные и, опять же, это скомпонованный вывод из Rollup. Мы поместим вывод в то же самое место, у которого просто будет немного другое название, например slider.build.js. Удивительно, но, в отличие от многих команд, которые вы можете выполнять, это не приведет к появлению вывода на вашем терминале. Вы можете легко убедиться в том, что она работает по файлу, который создает.

Как и в случае с тремя сценариями для Rollup в файле package.json, мы можем добавить сценарии для транспиляции с помощью Babel. В приведенном ниже листинге показаны три новых сценария сборки.

**Листинг 12.12 Шаг транспиляции Babel для каждого компонента**

```
"build-slider-ie": "./node_modules/.bin/babel
  chapter12and13/components/slider/slider.js --out-file
  chapter12and13/components/slider/slider.build.js",
```

```
"build-coordpicker-ie": "./node_modules/.bin/babel
  chapter12and13/components/coordpicker/coordpicker.js --out-file
  chapter12and13/components/coordpicker/coordpicker.build.js",
"build-colorpicker-ie": "./node_modules/.bin/babel
  chapter12and13/components/colorpicker/colorpicker.js --out-file
  chapter12and13/components/colorpicker/colorpicker.build.js",
```

Опять же, как это было с Rollup, эти команды можно объединить в один шаг с помощью знаков амперсанда:

```
"build-ie": "npm run build-slider-ie && npm run build-coordpicker-ie &&
npm run build-colorpicker-ie"
```

Конечно, чтобы транспилировать все три, мы могли бы использовать терминал и выполнить команду

```
npm run build-ie
```

Более того, давайте создадим один скрипт для компоновки и транспиляции. В приведенном ниже листинге показан файл package.json целиком с новым скриптом «build».

**Листинг 12.13 Текущий файл package.json с компоновкой от Rollup и транспиляцией от Babel**

```
{
 "name": "wcia",
 "version": "1.0.0",
 "dependencies": { },
 "devDependencies": {
   "@babel/cli": "^7.2.3",
   "@babel/core": "^7.2.2",
   "@babel/preset-env": "^7.2.3",
   "rollup": "^1.0.2",
 },
 "scripts": {
   . . . предварительно добавленные скрипты . . .
   "build": "npm run build-rollup
   ➥&& npm run build-ie"
 }
}
```

Новый скрипт сборки, который компонует и транспилирует все компоненты

Теперь мы вернулись к нормальному и легко запоминающемуся процессу сборки. Просто используйте команду `npm run build` в своем терминале, и все три компонента будут скомпонованы и транспилированы, чтобы не было никаких проблем с IE!

Поскольку я принял решение, что у меня будет два разных вывода, есть смысл иметь в наличии два разных HTML-файла, один для IE, а другой для всего остального. Конечно, после добавления папки с исходным кодом файловая структура изменилась. Лично я думаю, что имеет смысл использовать оригинальные исходные файлы вместо скомпонованного вывода Rollup, чтобы мы могли получать мгновенный отклик во время разработки. Если добавить задачу «наблюдения» Rollup, это также может

сработать в случае с более сложной системой, которая постоянно работает во время разработки, но, чтобы ничего не усложнять, мы просто немного изменим путь в файле demo.html:

```
<script type="module" src="src/slider.js"></script>
```

Чтобы запустить демонстрацию с IE, тег `<script>` необходимо дополнительно изменить. Поскольку модули не поддерживаются, в нем больше не может содержаться фраза `type = "module"`. Мы создадим другой демонстрационный файл для IE, который будет называться demo-ie.html. Пока тег `<script>` будет единственным, что изменится:

```
<script src="slider.build.js"></script>
```

Конечно, мы повторим этот шаг и для двух других компонентов. На рис. 12.6 показана структура одного из компонентов с выходными файлами.

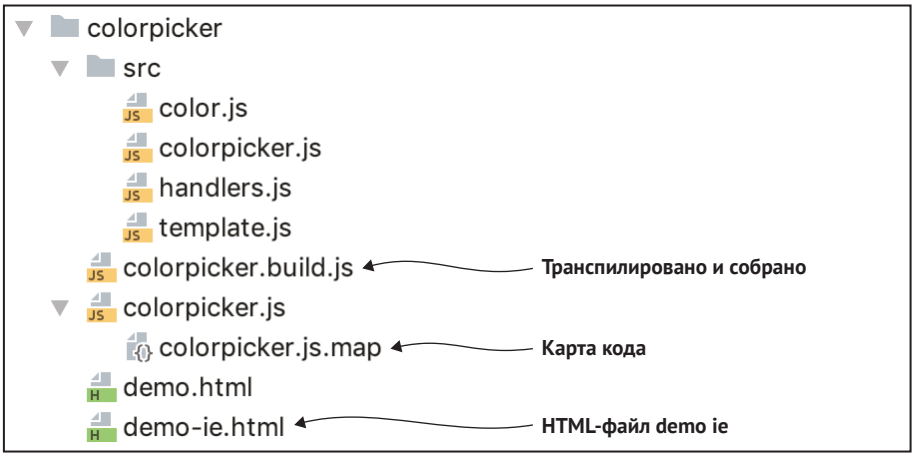



**Рис. 12.6 Файловая структура проекта со скомпонованными и транспилированными данными. Такие инструменты, как Webstorm, изображенные здесь, делают JS-файл похожим на каталог, чтобы скрыть сложность сгенерированных файлов, таких как исходные карты, даже если на самом деле это плоская файловая структура**

## 12.5.2 CSS-vars-ponyfill

При дальнейшем рассмотрении компонентов во время тестирования в IE11 с использованием нового демофайла видно, что дела обстоят немногим менее чем идеально. На рис. 12.7 показаны некоторые визуальные расхождения. В остальном все работает просто отлично.

Это отнюдь не проблема веб-компонентов, но мы использовали CSS-переменные, чтобы сделать компоненты гибкими с точки зрения стилей. Эти переменные позволили нам настроить глобальное округление границ, цвет текста и т. д. и влиять на все, что есть на странице. Недостаток в том, что это более новая функция. Даже при широко распространенной

поддержке CSS-переменных со стороны браузеров IE11 просто не добавил соответствующие функции, поэтому он не сможет их использовать. Значит ли это, что нам нужно отказаться от CSS-переменных? Нет, мы справимся. Обычно я бы сказал «полифил», но в этом случае я буду использовать термин «понифил».

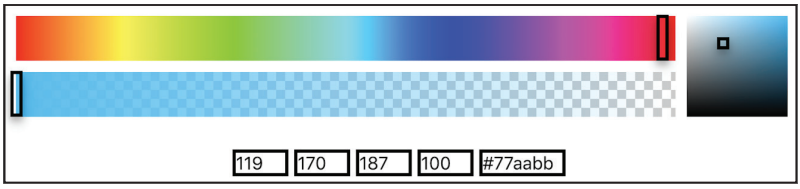



**Рис. 12.7 В браузере IE11 палитра цветов выглядит немного иначе**

Честно говоря, я не слышал о понифилах до того, как приступить к изучению обработки CSS-переменных в IE11. Полифилы имеют тенденцию изменять среду выполнения браузера. Например, при полизаполнении пользовательских элементов веб-компонента создается глобальный объект, `customElements`, чтобы соответствовать современным браузерам, в которых он уже присутствует. Добавление этого объекта означает, что мы модифицируем браузер, в частности добавляем функции, предоставляемые его глобальному пространству. Понифилы обещают, что не будут изменять среду браузера, когда заставляют работать неподдерживаемые функции.

CSS-vars-ponyfill не является полностью готовым решением, а это значит, что для его запуска нам потребуется вызвать функцию, чтобы запустить его. Во-первых, теперь, когда у нас есть файл package.json, давайте установим понифил с помощью `npm`. Поскольку это зависимость на стороне клиента, мы сохраним ее, но не как dev-зависимость, как в случае с другими инструментами сборки:

```
npm install css-vars-ponyfill
```

После установки понифил можно добавлять в каждый файл demo-ie.html:

```
<script src="https://unpkg.com/@webcomponents/custom-elements"></script>
<script src="https://cdn.jsdelivr.net/npm/css-vars-ponyfill@1"></script>
<script src="slider.build.js"></script>
```

Отмечу, что в своем теге `<script>` я использую онлайн-версию просто для того, чтобы предоставить информацию о том, что он существует, но вы можете использовать ее или заменить на ту, которая только что была установлена на node_ modules/css-vars-ponyfill/dist/css-vars-ponyfill.js.

Как уже упоминалось, `css-vars-ponyfill` не является полностью готовым решением. Нам все еще нужно вызвать функцию, чтобы он приступил к работе. Это работает путем обработки тегов `<style>` на странице и замены стилей, что будет понятно для IE. Поскольку стили компонента

недоступны до тех пор, пока в каждом из них не будет настроено свойство innerHTML, мы запустим понифил после этого.

В приведенном ниже листинге показана функция connectedCallback компонента ползунка с установленным понифилом.

**Листинг 12.14 Добавляем CSS vars ponyfill, чтобы позволить существующим CSS-переменным работать в IE**

```
connectedCallback() {
   if (!this.initialized) {
       this.root.innerHTML = Template.render({ useShadowDOM:
         Slider.USE_SHADOWDOM_WHEN_AVAILABLE && this.attachShadow });
       this.dom = Template.mapDOM(this.root);
       if ( typeof cssVars !== 'undefined') {
           cssVars();
       }
       this.initialized = true;

       if (this.backgroundcolor) {
           this.setColor(this.backgroundcolor);
       }

       if (this.value) {
           this.refreshSlider(this.value);
       }
   }
}
```

Вызываем функцию cssVars для замены CSS-переменных в браузере

Проверяем, существует ли понифил и был ли он добавлен с помощью тега <script> на демонстрационной странице

Поскольку мы только что поместили скрипт на страницу, наше использование просто диктует, что функция cssVars присоединена к глобальному пространству (в противоположность тому, как я описывал понифил). Однако это решение существует в виде модуля, который мы могли бы импортировать и запускать таким образом. Здесь мы даем пользователям компонентов возможность использовать или не использовать понифил, основываясь на том, добавили они скрипт или нет. Обратите внимание, что синтаксис проверки *немного* странный. Если бы я просто использовал !cssVars, в случае если она не существует, мы бы получили сообщение о том, что функция cssVars не определена, поскольку она не является свойством чего-либо и может быть просто неопределенной переменной в области видимости, которую мы проверяем. Поэтому мы действуем более осторожно, чтобы избежать появления ошибки при рассматривании ее типа.

## Резюме

Из этой главе вы узнали:

- о простом способе запуска сценариев с использованием npm и файла package.json без необходимости полагаться на более сложные системы сборки, которые требуют долгой настройки;

- о причинах этапа сборки, будь то компоновка кода для промышленной эксплуатации или транспиляция, чтобы новые функции JS работали в старых браузерах;
- что компоновка полезна для объединения вашего кода в один или несколько файлов, в то же время разумно исключая неиспользуемый импорт.

# 13

# Тестирование компонентов

***Эта глава охватывает следующие темы:***

- выполнение тестов с помощью тестировщика веб-компонентов (WCT);
- использование фреймворка Mocha и библиотеки Chai для создания тестов;
- альтернативный запуск тестов с использованием Karma и Karma Web Components.

Прежде чем считать палитру цветов законченной, необходимо сделать еще кое-что, что следует принять во внимание. Это не тот шаг, к которому все прилагают усилия, но тестирование может иметь большое значение с точки зрения того, насколько можно доверять компоненту и насколько легко его поддерживать. То же самое можно сказать почти обо всем, что вы делаете, занимаясь разработкой программного обеспечения.

Тестирование можно делить по-разному, но один из способов – это функциональное тестирование в сравнении с модульным. Границы между ними могут быть довольно размытыми, но модульное тестирование, как правило, включает в себя взятие фрагмента кода, который делает что-то одно, или модуля, и выполнение серии тестов над ним, чтобы убедиться, что он не рушится в каком-либо крайнем случае, который не был учтен при разработке. С другой стороны, функциональное тестирование

включает в себя тестирование определенной части функционала, ожидаемого пользователем, – это не гарантирует, что код делает все делает правильно, только то, что делает приложение.

## 13.1 Модульное тестирование и разработка через тестирование

В палитре цветов утилиты для конвертации цветов из файла component/colorpicker/sr/color.js являются идеальными кандидатами для модульного тестирования. Например, в этом модуле существует функция для конвертации цвета в виде RGB в шестнадцатеричное значение. Одиночный тест может гарантировать, что объект, который выглядит как `{r: 255, g: 0, b: 0}`, на выходе дает `# ff0000`. Он может работать идеально всеми правильными способами, но не исправляет ошибку, когда передаются недопустимые значения (например, если это число больше 255 или числа отрицательные). Практика написания модульных тестов – отличный способ подумать об этих крайних случаях.

Модульные тесты можно использовать в ряде случаев, и если какой-либо из них окажется неудачным, вы как разработчик будете знать, что нужно что-то исправить. Конечно, если вы все исправили и внесли для этого много изменений, возможно, вы захотите удостовериться, что больше ничего не сломали. Поэтому вы перезапускаете модульные тесты. Если они все прошли успешно, вы уверены в том, что этот фрагмент функционала работает как обычно.

Во время тестирования также можно отслеживать покрытие кода. Например, если в вашем коде был блок `if/then`, а ваши модульные тесты не охватили случай, который имел место при таких условиях, будет создан отчет, указывающий на то, что вы не покрыли эти конкретные строки кода.

Когда обычно идет речь о модульном тестировании, особенно вне веб-разработки, модульные тесты часто не включают в себя пользовательский интерфейс. Современная веб-разработка – то место, где эти границы склонны быть расплывчатыми. Если взять, к примеру, какой-нибудь компонент, будь то веб-компонент или компонент в React, Vue, Angular и т. д., у него будет API. Этот API можно рассматривать как модуль, который можно протестировать. Более того, значительная часть функциональных возможностей JS, подобных этой, должны быть протестированы, и их нельзя запускать без DOM.

Появившееся недавно и ставшее популярным решение этой проблемы состоит в полной виртуализации DOM. JSDOM (https://github.com/jsdom/jsdom) предлагает полностью виртуальную модель DOM, которая работает без браузера или даже графического интерфейса прямо в Node.js или браузере. К сожалению, веб-компоненты пока еще не поддерживаются в JSDOM, поэтому это не то решение, которое можно использовать, если вы не разобрали и не протестировали части своего веб-компонента, фактически не запустив его в качестве компонента.

Из-за этого, особенно при тестировании веб-компонентов, нам необходимо будет использовать браузеры для запуска тестов. Несмотря на введение браузера и пользовательского интерфейса в процесс тестирования, мы по-прежнему можем тестировать отдельные «модули» функциональности.

Еще один момент, связанный с размытыми границами, возникает, когда речь идет о функциональном тестировании. Данные типы тестов можно рассматривать с точки зрения пользователя. Когда пользователь нажимает кнопку, происходит нечто значимое для него, и результат можно протестировать. Иногда, при желании, эти тесты можно смешивать с модульными, и если для запуска теста используется браузер, сделать это гораздо проще.

Причина этого заключается в том, что существует множество различных методик и инструментов тестирования. Мы будем обсуждать здесь инструменты и методы, которые обычно рассматриваются как модульные тесты, или тесты, которые разработчик чаще пишет с точки зрения разработки через тестирование (TDD). Это тесты, которые разработчик пишет при создании кода. В идеальной ситуации разработчик создал бы какую-то часть функционала и написал бы тесты для резервного копирования этой части.

Тестирование – это обширная тема, и о его различных аспектах написано множество книг. Однако, что касается веб-компонентов, я думаю, что разработка через тестирование и модульные тесты являются наиболее актуальными темами для обсуждения, учитывая нюанс, что в настоящее время мы должны полагаться на браузер, хотя можно ожидать использования какого-то решения, подобного JSDOM.

## 13.2 Web Component Tester

Еще одна причина для изучения этого типа тестирования заключается в том, что команда Polymer создала собственный инструмент тестирования для веб-компонентов, который носит название Web Component Tester (WCT). Его можно найти по адресу https://github.com/Polymer/tools/tree/master/packages/web-component-tester. По умолчанию в него встроено множество вещей, и с ним действительно легко начать работать.

Подобные инструменты тестирования часто разбиты на несколько разных частей. В WCT автоматизация браузера выполняется Selenium. Автоматизация браузера просто означает, что браузеры, в которых вы собираетесь разместить свои тесты, должны автоматически запускаться из терминала с помощью HTML- и JS-кода и стилей, и эти браузеры должны докладывать в ваш терминал о результатах.

Фреймворк для тестирования – Mocha, в случае с WCT это то, что вы используете как разработчик для организации и написания своих тестов. С помощью Mocha вы создадите *наборы*, или группы тестов, в которых каждая группа заполнена фактическими единичными тестами. Mocha предо-

ставляет хуки для настройки перед тестами, хуки для того, чтобы сносить все по окончании тестов, поэтому вы можете запускать следующий тест с чистого листа, и множество других функциональных возможностей.

Последним важным элементом WCT является библиотека для проверки утверждений, в данном случае Chai. Библиотеки для проверки утверждений – это небольшая, но важная часть любого решения для тестирования. По сути, утверждение – это вопрос, который вы задаете и ожидаете, что ответ будет верным. Простой пример: «Я ожидаю, что 1 + 2 будет равно 3». Это утверждение можно перефразировать с помощью Chai, если написать:

```
assert.equal(1 + 2, 3);
```

Конечно, 1 + 2 в сумме всегда дает 3, поэтому это утверждение никогда не будет ошибочным. На практике вы не увидите жестко закодированные значения (по крайней мере, с обеих сторон утверждения). Скорее всего, вам придется проверять, что переменная или результат функции равна другой переменной или другому результату. Например, у вас может быть простая функция, которая удваивает числа. Ваша функция `doubleNum(num)` может принимать значение и удваивать его. Чтобы знать, что это работает, вам нужно выполнить ряд утверждений, таких как:

```
assert.equal(doubleNum(2), 4);
```

Более сложные функции могут и не сработать по разным причинам, и тестирование – отличный способ выявить такие случаи.

Chai предлагает различные способы делать утверждения, но если говорить в двух словах, эта библиотека делает что-то одно, и делает это хорошо. На рис. 13.1 показан весь поток WCT.

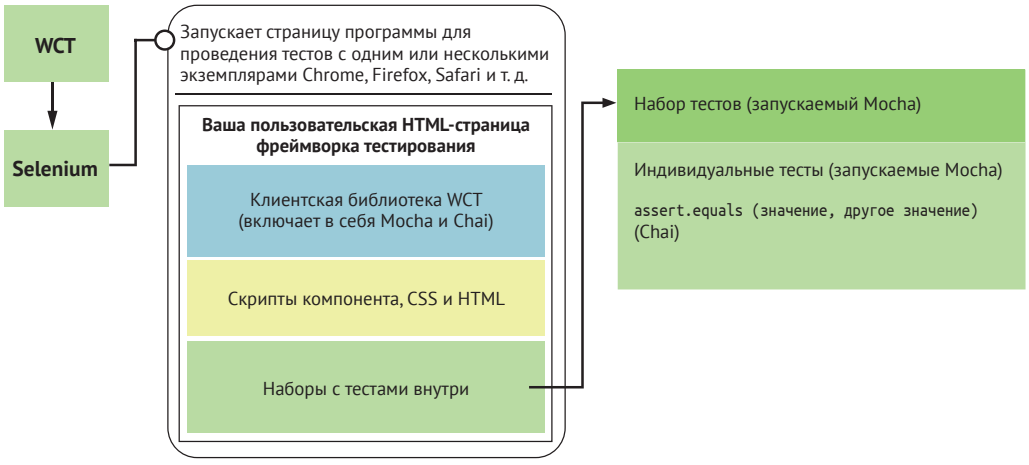



**Рис. 13.1 Поток WCT**

Установить WCT просто:

```
npm install --save-dev web-component-tester
```

Это еще одна зависимость, используемая только в период разработки, которую нам понадобится запускать локально из папки node_modules/bin. Однако я должен отметить, что поскольку Selenium является зависимостью и использует Java, один из наблюдательных технических рецензентов этой книги обнаружил, что на его компьютере с Windows 10 запуск WCT был невозможен, пока он не воспользовался версией Java 8. У меня есть ощущение, что запуск разных версий Java будет своего рода движущейся целью на разных платформах при обновлении нескольких зависимостей, когда выходят новые версии WCT и Selenium. В идеале вам повезет, как и мне, и вам даже не нужно будет думать о Java при установке WCT – в противном случае нужно будет обратить внимание на установленную у вас версию Java.

Однако нам потребуется запускать тесты для файлов, поэтому сейчас самое время создать тестовую папку с тестовым HTML-файлом для каждого компонента. На рис. 13.2 показана новая структура папок с новыми тестовыми файлами.

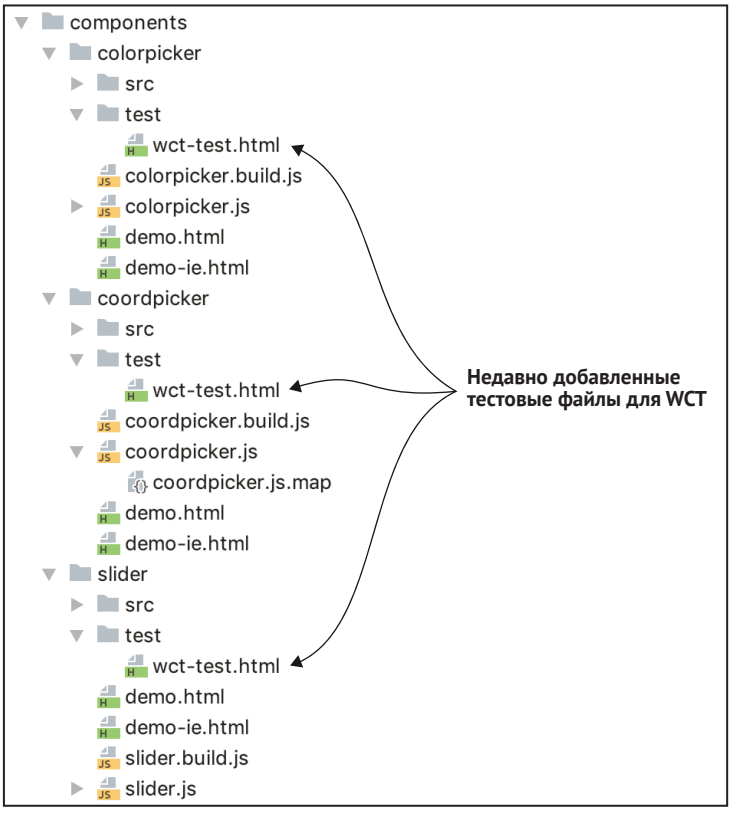



**Рис. 13.2 Структура проекта с тестовыми файлами**

HTML-файл в новой тестовой папке обычно называется как-то вроде index.html или index.test.html. Но в этой главе мы будем рассматри-

вать несколько разных способов тестирования; чтобы было понятно, какой именно, я назвал этот первый HTML-файл wct-test.html. Прежде чем приступить к созданию реальных тестов в файле, давайте добавим скрипт в файл package.json. В приведенном ниже листинге показана последняя версия файла package.json, после того как мы установили WCT и добавили скрипт.

**Листинг 13.1 Добавляем WCT в файл package.json нашего проекта**

```
{
 "name": "wcia",
 "version": "1.0.0",
 "dependencies": {
   "css-vars-ponyfill": "^1.16.2"
 },
 "devDependencies": {
   "@babel/cli": "^7.2.3",
   "@babel/core": "^7.2.2",
   "@babel/preset-env": "^7.2.3",
   "mocha": "^5.2.0",
   "rollup": "^1.0.2",
   "rollup-plugin-babel": "^4.2.0",
   "web-component-tester": "^6.9.2"   <-- Пакет WCT
 },
 "scripts": {
   "wcttest": "./node_modules/.bin/wct   <-- Сценарий WCT
--npm chapter12and13/components/**/test/wct-test.html",
   "build-slider": "./node_modules/.bin/rollup
    chapter12and13/components/slider/src/slider.js --file
    chapter12and13/components/slider/slider.js --format umd
    --name slider -m",
   "build-coordpicker": "./node_modules/.bin/rollup
    chapter12and13/components/coordpicker/src/coordpicker.js --file
    chapter12and13/components/coordpicker/coordpicker.js --format umd
    --name coordpicker -m",
   "build-colorpicker": "./node_modules/.bin/rollup
    chapter12and13/components/colorpicker/src/colorpicker.js --file
    chapter12and13/components/colorpicker/colorpicker.js --format umd
    --name colorpicker -m",
   "build-rollup": "npm run build-slider && npm run build-coordpicker &&
    npm run build-colorpicker",
   "build-slider-ie": "./node_modules/.bin/babel
    chapter12and13/components/slider/slider.js --out-file
    chapter12and13/components/slider/slider.build.js",
   "build-coordpicker-ie": "./node_modules/.bin/babel
    chapter12and13/components/coordpicker/coordpicker.js --out-file
    chapter12and13/components/coordpicker/coordpicker.build.js",
   "build-colorpicker-ie": "./node_modules/.bin/babel
    chapter12and13/components/colorpicker/colorpicker.js --out-file
    chapter12and13/components/colorpicker/colorpicker.build.js",
   "build-ie": "npm run build-slider-ie && npm run build-coordpicker-ie &&
```

```
    npm run build-colorpicker-ie",
   "build": "npm run build-rollup && npm run build-ie"
 }
}
```

WCT – чрезвычайно простая команда для запуска. Просто запустите исполняемый файл WCT, используя путь к одному или нескольким тестам. В случае с нашей тестовой конфигурацией HTML-файлы всегда находятся в определенном месте в каждой папке компонента. Поскольку нам нужно запустить все компоненты с помощью одной команды, мы поменяем имя компонента, используя символы подстановки: `components/**/test/wct-test.html`. Наконец, поскольку для запуска используется `npm`, WCT нужен флаг `--npm`.

## 13.2.1 Пишем тесты

Каждый тестовый HTML-файл будет иметь очень знакомую структуру. Сейчас, когда тестов нет, структура в приведенном ниже листинге ничем не отличается от любого другого HTML-файла. Единственной зависимостью от нашего компонента являются файлы browser.js, которые предоставляют все функции и загрузку WCT на стороне клиента.

**Листинг 13.2 Структура тестового файла для WCT**

```
<html>
<head>
  <script src="../../../../node_modules/web-component-tester/browser.js">
  </script>   <-- Необходимые сценарии тестирования
  <script     <-- Импорт компонента ползунка
       type="module"
       src="../src/slider.js">
  </script>
  <style>
      wcia-slider {
         width: 500px;   <-- Устанавливаем ползунку ширину для запуска зависимых от размера тестов
      }
   </style>
</head>
<body>
<wcia-slider value="50"></wcia-slider>   <-- Компонент ползунка
<script>
// Здесь идут тесты;   <-- Место для тестов
</script>
</body>
</html>
```

Когда мы начинаем писать тесты, помните, что здесь мы используем конкретный фреймворк для тестирования и библиотеку утверждений, Mocha и Chai. У Mocha на самом деле имеется два разных стиля: разработка через тестирование и разработка через поведение (или функциональный стиль). Стиль по умолчанию в WCT – разработка через тестиро-

вание. В идеале это модульные тесты, которые вы пишете при создании компонента. Теперь давайте определим группу или набор тестов для ползунка.

**Листинг 13.3 Начало тестового набора компонента ползунка**

```
suite('slider value getting/setting', function() {
  const sliderWidth = 500;
  const thumbCenterOffset = 5/2 + 3; // width/2 + left border
  const slider = document.body.querySelector('wcia-slider');
```

Определяем ширину ползунка при подготовке к тестам

Определяем центр ползунка, что поможет нам в будущих тестах

Первым параметром, передаваемым в функцию `suite` в Mocha, является имя набора тестов. Здесь нужно быть конкретным, что очень удобно. Чем более подходящим будет название теста, тем легче его будет найти, когда тест и набор сообщат о сбое в вашем терминале.

Второй параметр – это функция, содержащая тесты. Хотя мы пока еще не добрались до определения одиночного теста, нужно сделать еще кое-что. Это хорошая возможность сделать шаг назад и подумать о том, какой функционал нужно протестировать. Компонент ползунка на самом деле не так уж и много делает. Учитывая, что это веб-компонент и мы потратили время на поддержку рефлексии, у нас должна быть возможность установить значение ползунка с помощью атрибута или JS API. Кроме того, компонент в действительности связывает только свое визуальное состояние (положение рукоятки) с числовым процентным значением. Мы можем проверить этот аспект, но положение рукоятки (в пикселях) будет зависеть от размера компонента.

Это то, что активируют эти две переменные. Сначала мы указываем ширину ползунка, которая уже была определена в таблице стилей на странице HTML. Во-вторых, мы определим, насколько смещен ползунок, чтобы отцентрировать его по положению, вычитая половину его ширины и размер левой границы. Наконец, мы возьмем ссылку на ползунок для проведения тестов.

Мы пометим первый тест как «slider get initial value». Компонент, как установлено на странице, имеет атрибут `value`, равный 50:

```
<wcia-slider value="50"></wcia-slider>
```

Таким образом, если 50 % – это начальное значение ползунка, рукоятка должна появиться в центре. В первом тесте, показанном в приведенном ниже листинге, можно утверждать три вещи:

**Листинг 13.4 Одиночный тест для ползунка**

```
test('slider get initial value', function () {
  assert.equal(slider.value, 50);
  assert.equal(
```

Проверяем, что значение ползунка в соответствии с JS составляет 50

Проверяем, что значение ползунка в соответствии с атрибутом равно 50

```
    slider.getAttribute('value'), 50);
  assert.equal(slider.root.querySelector('.thumb').style.left, sliderWidth *
    50/100 - thumbCenterOffset + 'px');
});
```

Проверяем, что рукоятка находится в центре компонента

Сначала мы проверяем, что при получении значения с помощью JS возвращается 50. Нам также нужно, чтобы Chai подтвердила, что мы получаем то же значение из атрибута, чтобы доказать, что рефлексия работает. Далее мы проверим положение ползунка. Учитывая значение 50, мы можем вычислить, где должна быть рукоятка, учитывая компонент, рукоятку и размер границы. Поскольку мы знакомы с внутренней работой ползунка, нам известно, что значение свойства `left` должно составлять 500 * 50/100 – (5/2 + 3), или 244,5 пикселя.

Здесь есть кое-что очень интересное. Вспомните тот момент, когда мы изучали теневую модель DOM. Мы обсуждали режимы корня теневого дерева «open» и «closed». Вы помните, что когда корень закрыт, как бы мы ни старались, нам так и не удалось добраться до компонента и работать с DOM. Открытый режим был немного более щадящим, потому что мы могли войти с помощью свойства компонента `shadowRoot`, зная, что эта задняя дверь не входила в планы разработчика компонента. Такая дверь здесь весьма кстати. Если мы не можем пробить теневую границу веб-компонента, то не можем использовать метод `querySelector` для рукоятки и протестировать ее.

В приведенном ниже листинге даны оставшиеся тесты для этого компонента.

**Листинг 13.5 Набор тестов для ползунка**

```
suite('slider value getting/setting', function() {
    const sliderWidth = 500;
    const thumbCenterOffset = 5/2 + 3; // width/2 + left border

    const slider = document.body.querySelector('wcia-slider');

    test('slider get initial value',
        function () {
            assert.equal(slider.value, 50);
            assert.equal(slider.getAttribute('value'), 50);
            assert.equal(slider.root.querySelector('.thumb').style.left,
              sliderWidth * 50/100 - thumbCenterOffset + 'px');
    });

    test('set slider value with JS',
        function () {
            slider.value = 20;
            assert.equal(slider.value, 20);
            assert.equal(slider.getAttribute('value'), 20);
            assert.equal(slider.root.querySelector('.thumb').style.left,
              sliderWidth * 20/100 - thumbCenterOffset + 'px');
    });
```

Проверяем начальное значение ползунка

Проверяем установку нового значения с помощью JS API

```
    test('set slider value with attributes',
        function () {
            slider.setAttribute('value', 30);
            assert.equal(slider.value, 30);
            assert.equal(slider.getAttribute('value'), 30);
            assert.equal(slider.root.querySelector('.thumb').style.left,
              sliderWidth * 30/100 - thumbCenterOffset + 'px');
    });
});
```

**Проверяем установку нового значения с помощью атрибутов**

Теперь мы можем запустить эти тесты с помощью команды `npm run wct-test`. На рис. 13.3 показан пример того, что вы увидите в терминале при запуске еще нескольких тестов, которые мы добавим немного. Обратите внимание, что проходящие тесты хороши и зеленые!

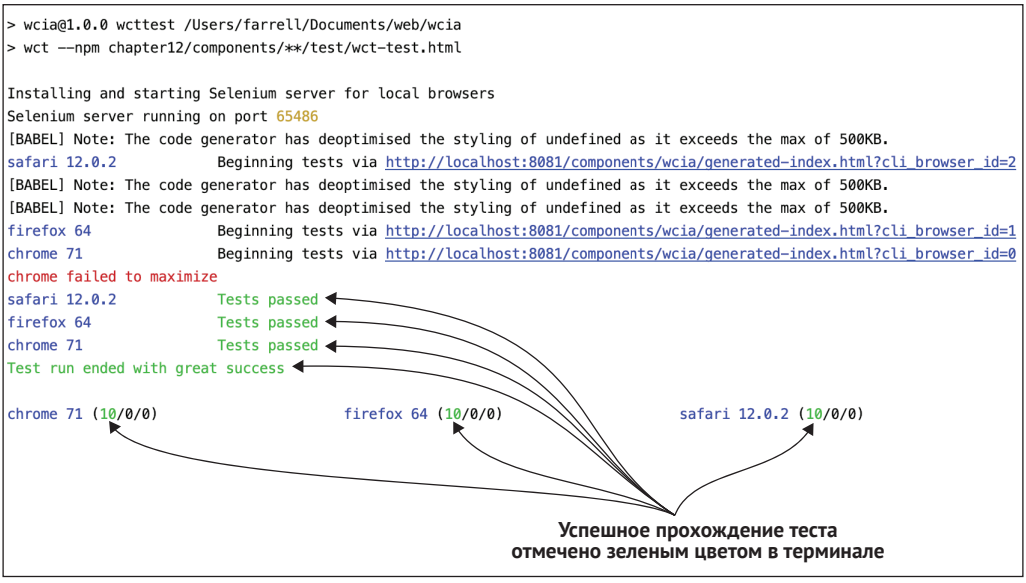



**Рис. 13.3 Прохождение тестов для компонента ползунка**

Также полезно показать неудавшиеся тесты! Практика написания этих тестов в процессе разработки заставляет задуматься о тестировании странных крайних случаев. Ползунок – простой случай, но существует несколько легких способов сделать так, что он потерпит неудачу. Подумайте, что случилось бы, если бы мы установили для ползунка значение больше 100 или меньше 0. Это не имеет смысла с точки зрения визуального отображения ползунка, поэтому в идеале мы должны ограничить ползунок максимальными и минимальными значениями. Давайте добавим еще два теста в приведенный ниже листинг, чтобы тесты не прошли, при условии что это ограничение уже действует.

**Листинг 13.6 Неудачные результаты тестирования, поскольку максимальные и минимальные значения еще не реализованы**

```
test('set slider value too big', function () {
   slider.setAttribute('value', 110);
   assert.equal(slider.value, 100);
   assert.equal(slider.getAttribute('value'), 100);
   assert.equal(slider.root.querySelector('.thumb').style.left, sliderWidth *
     100/100 - thumbCenterOffset + 'px');
});

test('set slider value too small', function () {
   slider.setAttribute('value', -10);
   assert.equal(slider.value, 0);
   assert.equal(slider.getAttribute('value'), 0);
   assert.equal(slider.root.querySelector('.thumb').style.left, sliderWidth *
     0/100 - thumbCenterOffset + 'px');
});
```

Значение ползунка больше 100, поэтому оно должно быть приведено к 100

Значение ползунка меньше 0, поэтому оно должно быть приведено к 0

Поскольку тесты провалились, как показано на рис. 13.4, мы определили функционал, который нам необходимо реализовать. В качестве упражнения можете попробовать настроить компонент таким образом, чтобы все эти и предыдущие тесты прошли успешно.

Помимо домашнего задания, которое я только что вам дал, есть еще три компонента, которыми можно заняться! В случае если вы застряли, я написал несколько тестов. Если подобное произошло, не стесняйтесь зайти в репозиторий для этой книги на сайте GitHub.

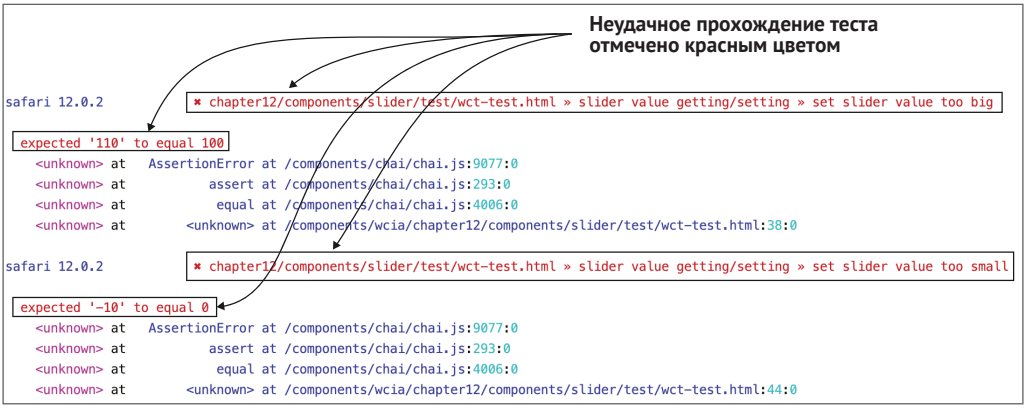



**Рис. 13.4 Тесты завершились неудачно**

## 13.3 Сравнение со стандартной тестовой конфигурацией при использовании Karma

С WCT довольно приятно работать! Настройка была чрезвычайно минимальной и позволила нам сосредоточиться на написании тестов, не перебирая сложные конфигурации, хотя какую-нибудь конфигурацию и можно было бы добавить, при наличии значений по умолчанию, которые вас не волновали. Более подробную информацию можно найти на странице https://github.com/Polymer/tools/tree/master/packages/web-component-tester.

Суть в том, что WCT предназначен для веб-компонентов и компонует некоторые ключевые вещи для их тестирования. Например, полифилы веб-компонента входят в комплект и автоматически включаются, если они необходимы в ваших тестовых приспособлениях HTML. WCT тоже ожидает готовности ваших компонентов, ожидая события браузера `WebComponents-Ready`. Также предоставляется помощник для использования тегов `<template>` в тестах.

Тем не менее WCT пока еще является новичком и находится в стадии разработки. Если вам это подходит, отлично! Если нет, и вы предпочитаете использовать что-то другое, это тоже нормально. Хорошо то, что у современных браузеров, поддерживающих веб-компоненты, нет никаких проблем с простыми тестами веб-компонентов. Они просто работают, как и любая другая веб-функция.

Учитывая это, давайте попробуем поменять инструменты тестирования. Мы заменим Selenium на Karma, но оставим Mocha и Chai. Таким образом все наши тесты останутся теми же, и это даст нам всю ту гибкость и плагины, которые поставляются с экосистемой Karma. На рис. 13.5 показана новая цепочка действий с использованием Karma.

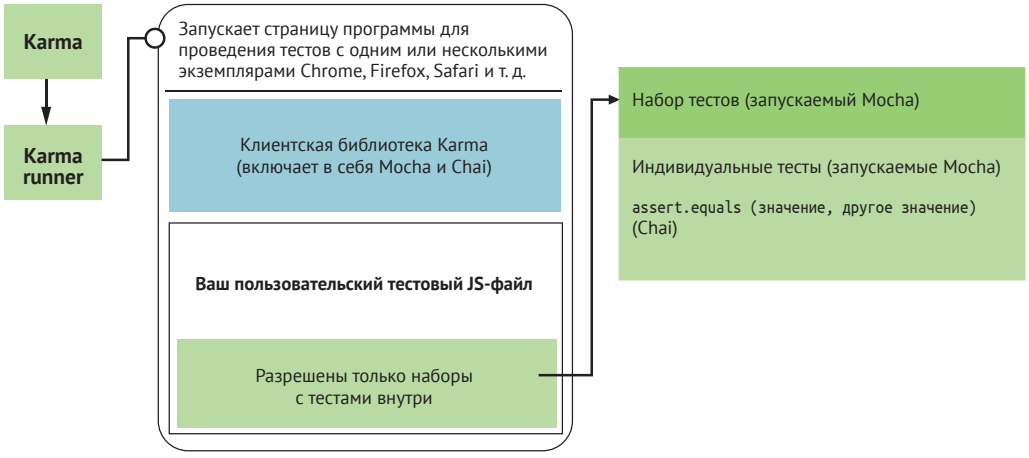



**Рис. 13.5 Новая цепочка действий с использованием Karma**

Недостаток установки на базе Karma состоит в том, что она немного сложна. Для начала давайте установим несколько вещей с помощью `npm`:

```
npm install --save-dev karma
npm install --save-dev mocha
npm install --save-dev chai
```

Mocha и Chai не будут работать в Karma без плагина, чтобы устранить разрыв, поэтому мы также установим и их:

```
npm install --save-dev karma-mocha
npm install --save-dev karma-chai
```

Karma тоже нужны плагины для запуска браузеров и выполнения тестов:

```
npm install --save-dev karma-chrome-launcher
npm install --save-dev karma-firefox-launcher
```

По мере продвижения будут и другие зависимости, а это основные. Последнее, что нужно сделать, – снова установить Karma, но уже глобально, и я объясню, для чего:

```
npm install -g karma
```

Эта глобальная установка не имеет ничего общего с запуском ваших тестов. Она предоставляет утилиту командной строки для создания конфигурационного файла. При выполнении команды `karma init` из корневого каталога вашего проекта после установки вам предлагается ряд подсказок и вопросов, как показано на рис. 13.6.

Не обязательно делать все в точности так, как делал я, потому что мы будем менять некоторые параметры по ходу дела. Хорошо, что у нас есть базовый файл karma.conf.js для работы. В приведенном ниже листинге показана начальная конфигурация.

**Листинг 13.7 Начальная конфигурация Karma (сокращенный вариант; пустые строки и комментарии были удалены)**

```
module.exports = function(config) {
 config.set({
   basePath: '',
   frameworks: ['mocha'],
   files: [],
   exclude: [],
   preprocessors: {},
   reporters: ['progress'],
   port: 9876,
   colors: true,
   logLevel: config.LOG_INFO,
   autoWatch: false,
   browsers: ['Chrome', 'Firefox'],
   singleRun: false,
   concurrency: Infinity
 })
}
```

```
Which testing framework do you want to use ?
Press tab to list possible options. Enter to move to the next question.
> mocha

Do you want to use Require.js ?
This will add Require.js plugin.
Press tab to list possible options. Enter to move to the next question.
> no

Do you want to capture any browsers automatically ?
Press tab to list possible options. Enter empty string to move to the next question.
> Chrome
> Firefox
>

What is the location of your source and test files ?
You can use glob patterns, eg. "js/*.js" or "test/**/*Spec.js".
Enter empty string to move to the next question.
>

Should any of the files included by the previous patterns be excluded ?
You can use glob patterns, eg. "**/*.swp".
Enter empty string to move to the next question.
>

Do you want Karma to watch all the files and run the tests on change ?
Press tab to list possible options.
> no
```

**Рис. 13.6 Вопросы и подсказки**

Первое, о чем нужно беспокоиться, – это использование модулей в наших веб-компонентах. WCT позволил нам игнорировать эту часть настройки, но когда мы сами запускаем тестовую конфигурацию, это уже наша проблема. С работой модулей и импорта не все так гладко, потому что за кулисами для нас работает Node.js, чтобы справиться с большим количеством тестов. Сама Node пока не поддерживает модули. Итак, нам понадобится выполнить шаг «предварительной обработки», прежде чем компоненты будут загружены на страницу, и запустить тесты.

Мы говорили о Rollup в предыдущей главе, поэтому давайте снова воспользуемся им! Когда я пишу эти строки, номер последней версии Rollup уже перевалил за 1.0. Обычно я бы рекомендовал установить `karma-rollup-preprocessor`. К сожалению, у нас возникли неудобства: этот модуль не поддерживает последнюю версию Rollup. Подобное может случаться время от времени, когда пакеты не синхронизируются друг с другом, особенно при таком количестве рабочих частей.

К счастью, мне удалось поохотиться и обнаружить, что кто-то разветвлял этот исходный проект и создал кое-что, что работает с последней

версией. Возможно, скоро нам не придется использовать это ответвление, но до тех пор можно установить это:

```
npm install --save-dev @metahub/karma-rollup-preprocessor
```

Из-за пространства имен пакета `@metahub` загрузка по умолчанию любого плагина с именем, начинающимся с «karma-», здесь не подходит. Как бы досадно это ни было, это знакомит нас с нестандартной настройкой, что представляет собой обычное явление при работе с конфигурацией Karma с нуля. Учитывая это, в конфигурационный файл нужно добавить запись `plugins`:

```
plugins: ['@metahub/karma-rollup-preprocessor', 'karma-*']
```

Поскольку здесь мы переопределяем значения по умолчанию, `karma-*` также необходимо добавить обратно в список. Мы добавим в список препроцессоров и запись, чтобы отображать JS-файлы в Rollup:

```
preprocessors: {
   './chapter12and13/components/**/*.js': ['rollup']
}
```

Здесь мы, вероятно, немного переусердствовали, поскольку JS-файлов несколько, а точка входа для каждого компонента только одна. Путь мог бы быть более точным, но я не слишком беспокоюсь о том, чтобы сократить время предварительной обработки на несколько микросекунд.

Rollup или плагины в целом также необходимо настроить. В приведенном ниже листинге показана конфигурация Rollup, которая подойдет нам, в файле karma.conf.js.

**Листинг 13.8 Конфигурация плагина Rollup в Karma**

```
rollupPreprocessor: {
 options: {
     output: {
         // Чтобы включить встроенные карты кода в качестве URI данных
         sourcemap: true,                    <-- Включаем карты кода
Формат iife --> format: 'iife',
         name: 'testing'                     <-- Название пакета
     }
 }
},
```

Карты кода здесь могут показаться необязательными, но только если ваши тесты прошли успешно. Если они провалились и нуждаются в отладке, вам понадобится узнать, какая строка в вашем первоначальном, нескомпонованном коде привела к такому результату. IIFE буквально означает «функция-выражение, вызываемая сразу после создания» (*immediately invoked function expression*). Мы хотим, чтобы наш набор вызывался сразу же после загрузки и создания определения веб-компонента? Да, пожалуйста. Это прекрасно подходит для тестирования и включает в себя то, как компонент был собран ранее с помощью Rollup. Прежде при использовании компоновки в стиле UMD (universal module defini-

tion) эта опция и многое другое были разрешены (отсюда и слово «universal» в названии). Имя набора здесь не имеет большого значения, но оно обязательно, и «testing» отлично подходит.

Два последних простых дополнения – это Chai для фреймворков, которые нам нужно использовать:

```
frameworks: ['mocha', 'chai'],
```

а также указание для Mocha использовать тестирование в стиле TDD:

```
client: { mocha: { ui: 'tdd' } }
```

Теперь, когда мы возвращаемся не к таким простым вещам, необходимо составить план для файлов, которые Karma будет обслуживать. В случае с WCT было просто отлично, что тесты можно запускать из HTML-файла. Мы вернемся к этому чуть позже, но, как таковая, Karma поддерживает только HTML-тесты, подобные этому. Проблема состоит в том, что Karma загружает HTML-файлы с помощью HTML-импорта. Поскольку Chrome – единственный браузер, который пока поддерживает эту устаревшую функцию (а скоро он перестанет это делать), это также единственный браузер, который может запускать наши тестовые HTML-страницы. Учитывая это, если никакие другие плагины не используются, нам понадобятся тесты в виде JS-файлов, и шаблоны файлов, используемые для обслуживания, будут выглядеть так:

```
files: [
 './chapter12and13/components/**/test/karma-test.js',
 './chapter12and13/components/**/*.js'
],
```

Данный шаблон файла обслуживает все JS-файлы компонентов, а также тесты компонентов с именем karma-test.js, которые нам еще нужно создать. Несмотря на то что мы используем другой инструмент, у нас по-прежнему есть Mocha и Chai, поэтому все ранее сделанные тесты можно скопировать. В приведенном ниже листинге показан тестовый файл только для JS, чтобы программным путем присоединить скрипты компонента, создать компонент и добавить все это в тело страницы.

**Листинг 13.9 Тестовый файл только для JS, создаваемый в тестовой папке для каждого компонента**

```
suite('slider value getting/setting', function() {
   const sliderWidth = 500;
   const thumbCenterOffset = 5/2 + 3; // width/2 + left border
   const container = document.createElement('div');
   container.innerHTML = `<script type="module" src="../src/slider.js">
                          </script>
                          <wcia-slider style="width: ${sliderWidth}px"
                           value="50"></wcia-slider>`;
   document.body.appendChild(container);
   const slider = container.querySelector('wcia-slider');

   test('slider get initial value', function () {
       assert.equal(slider.value, 50);
```

Помещаем компонент и скрипт компонента в контейнер

Добавляем все в тело страницы для тестирования

```
      assert.equal(slider.getAttribute('value'), 50);
      assert.equal(slider.root.querySelector('.thumb').style.left,
    sliderWidth * 50/100 - thumbCenterOffset + 'px');
  });
});
```

Вы можете обратиться к репозиторию для этой книги на GitHub, чтобы увидеть все эти новые JS-тесты на месте для всех компонентов, но в листинге 13.9 показаны только реальные различия с использованием ползунка в качестве примера.

Сейчас самое время попробовать запустить тесты! Как и прежде, имя `test` для скрипта, добавленного в файл package.json, будет более подходящим именем, но поскольку мы имеем дело с несколькими типами тестов, его можно назвать `karmatest`:

```
"karmatest": "./node_modules/karma/bin/karma start karma.conf.js"
```

Этот скрипт просто дает Karma указание начать тестирование для конфигурационного файла, который мы только что создали. Нам нужно поменять кое-что в конфигурации, чтобы все работало нормально. Прежде чем мы это сделаем, выполняя команду `npm start karmatest`, мы запускаем браузеры в записи `browsers` в конфигурации Karma. Появляется изображение, подобное тому, которое мы видим на рис. 13.7, когда браузер останавливается после запуска тестов.

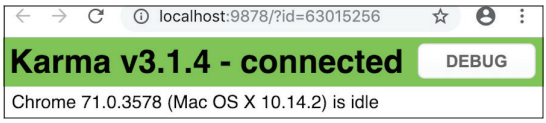



**Рис 13.7 Страница Karma**

Причина приостановки заключается в том, что это дает нам возможность нажать кнопку **Debug** (Отладка) и увидеть тесты, запущенные в контексте. Мы можем открыть инструменты разработчика в браузере, как обычно, и увидеть результаты теста, просмотреть элементы на странице и отладить все обнаруженные ошибки. Этот режим отладки показан на рис. 13.8, хотя, опять же, в действительности это просто браузер с открытыми инструментами разработчика.

Предположим, что все работает и нам не нужно ничего отлаживать. Желательно, чтобы Karma запустила браузеры, выполнила тест и на этом все. Для этого нам просто нужно исправить значение записи `singleRun` в конфигурации Karma и вместо `false` указать `true`:

```
singleRun: true
```

Более того, у нас есть возможность вообще не видеть, как браузеры всплывают на экране, если «headless»-версии поддерживаются средствами запуска Karma, такими как Chrome и Firefox. Обратите внимание, что не только Karma поддерживает это. Обе обычным образом установленные версии этих браузеров предлагают режим «headless», а средства запуска Karma просто используют его:

```
browsers: ['FirefoxHeadless', 'ChromeHeadless'],
```

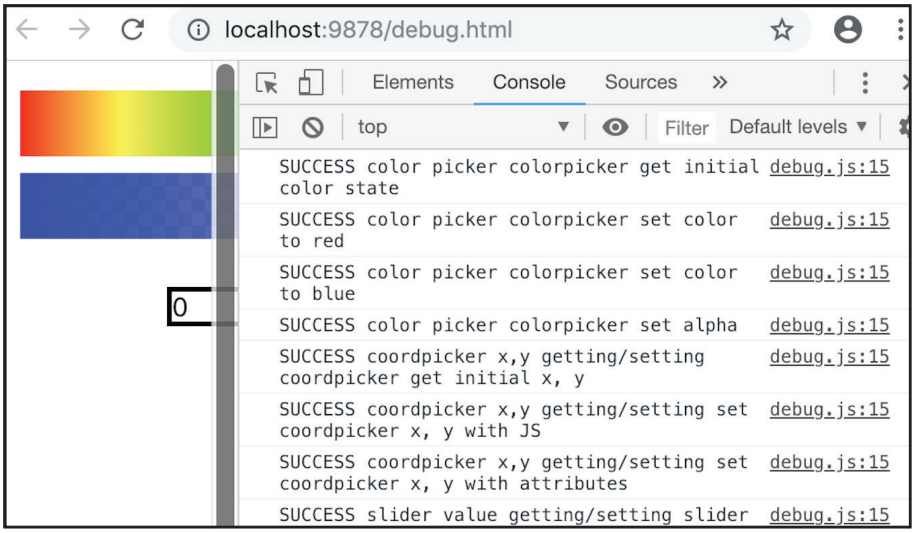



**Рис. 13.8 Страница отладки Karma**

В этом листинге рассматриваются все параметры, которые мы изменили в конфигурации Karma, чтобы провести тесты с использованием Karma, Mocha и Chai.

**Листинг 13.10 Конечная конфигурация Karma**

```
module.exports = function(config) {
 config.set({
   basePath: '',
   plugins: [
       '@metahub/karma-rollup-preprocessor',
       'karma-*'],
   frameworks: ['mocha', 'chai'],
   files: [
     './chapter12and13/components/**/test/karma-test.js',
     './chapter12and13/components/**/*.js'
   ],
   exclude: [],
   preprocessors: {
       './chapter12and13/components/**/*.js': ['rollup']
   },

   rollupPreprocessor: {
     options: {
         output: {
             sourcemap: true,
             format: 'iife',
             name: 'testing'
         }
     }
   },
```

Добавлены плагины для Rollup, и повторно добавлено karma-* по умолчанию

Добавлена библиотека Chai

Добавлены конкретные файлы для нашей настройки

Добавлены препроцессор и конфигурация Rollup

```
    reporters: ['progress'],
    port: 9876,
    colors: true,
    logLevel: config.LOG_INFO,
    autoWatch: false,
    browsers: [                          <-- Версии браузеров заменены на headless
         'FirefoxHeadless',
         'ChromeHeadless'],
    singleRun: true,                     <-- Значение изменено на true
    concurrency: Infinity,
    client: {                            <-- Добавлено тестирование в стиле TDD
        mocha: {
            ui: 'tdd'
        }
    }
  })
};
```

Теперь будем надеяться, что при запуске команды `npm start karmatest` везде в вашем терминале будет зеленый цвет – это значит успешные результаты! Здесь было довольно много подвижных частей, и, действуя методом проб и ошибок, мы настроили все это самостоятельно; но преимущество по сравнению WCT состоит в том, что у вас гораздо больше контроля и значительное количество совместимых плагинов с такой настройкой тестирования, которая уже давно существует.

Единственное, что немного огорчает, – это отсутствие возможности использовать тестовый HTML-файл, как в WCT. Лично для меня это моя любимая часть экосистемы WCT. К счастью, существует специальный плагин Karma для веб-компонентов!

## 13.3.1 Плагин karma-web-components

Плагин `karma-web-components` делает разные вещи, но в основном он позволяет нам снова использовать тестовые HTML-файлы, как это делает WCT. До сих пор мы успешно справлялись с проблемами синхронизации нагрузки в тестах; но данный плагин также слушает событие `WebComponentsReady` в вашем браузере перед запуском тестов, чтобы убедиться, что все готово для успешного тестирования.

Первое, что нужно сделать, – это установить плагин:

```
npm install --save-dev karma-web-components
```

Затем мы можем добавить в тестовую папку каждого компонента дополнительный файл karma-test.html. Для компонента-ползунка приведенный ниже листинг показывает, что находится внутри.

**Листинг 13.11 Тестовый HTML-файл, который будет использоваться плагином karma-web-components**

```
<html>
<head>
  <script                            <-- Импорт модуля компонента
```

```
        type="module"
        src="../src/slider.js">
    </script>
    <script src="../../../../node_modules/karma-web-components/framework.js">
    </script>
    <style>
        wcia-slider {
            width: 500px;
        }
    </style>
</head>
<body>
<wcia-slider value="50"></wcia-slider>
<script>
// Те же самые тесты и набор тестов, которые у нас уже были ранее;
</script>
</body>
</html>
```

Фреймворк, предоставляемый плагином

Компонент ползунок на странице

Как и WCT, плагин `karma-web-components` должен загружаться на стороне клиента. Но этот единственный файл скрипта – единственное, что нужно изменить на этой странице, в отличие от тестовой страницы WCT. Это другая библиотека для загрузки, но вся тестовая конфигурация может остаться прежней. На рис. 13.9 показан обновленный рабочий поток плагина `karma-web-components`. Очень напоминает WCT.

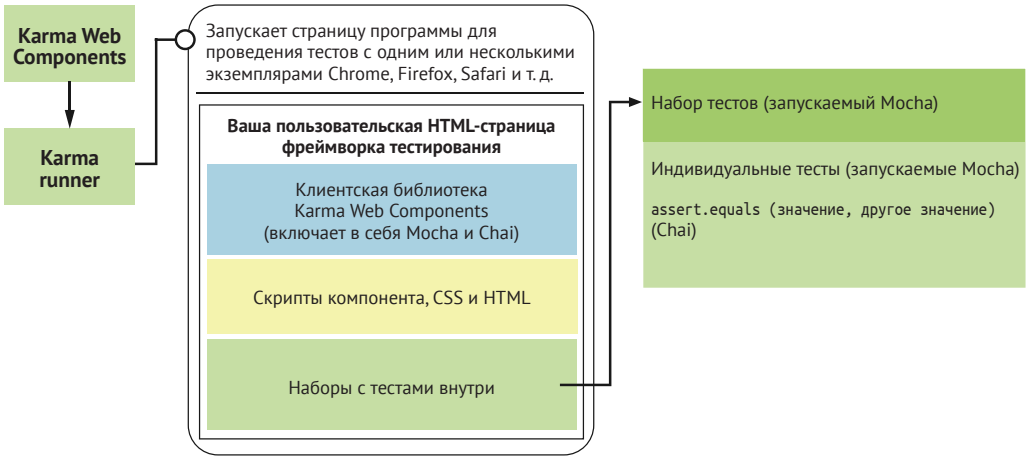



**Рис. 13.9 Рабочий поток karma-web-components**

Вернемся в конфигурационный файл Karma. Нам понадобится добавить плагин в существующий список:

```
frameworks: ['mocha', 'chai', 'web-components'],
```

Единственное другое отличие состоит в том, что файлы нужно будет обслуживать немного иначе, как показано в приведенном ниже листинге.

**Листинг 13.12 Конфигурирование Karma для использования плагина karma-web-components**

```
files: [
 './chapter12and13/components/**/src/*.js',
 './node_modules/karma-web-components/framework.js',
   {
       pattern: './chapter12and13/components/**/test/karma-wc-test.html',
       watched: true,
       included: false
   }
],
```

Конечно, существующие файлы веб-компонентов еще нужно обслуживать. Также требуется фреймворк с `karma-web-components` на стороне клиента. Наконец, необходимо обслуживать HTML-файл, содержащий тесты, но нам также нужно будет настроить пару параметров. Для флага `included` должно быть установлено значение `false`, чтобы HTML-файлы не загружались в браузер автоматически. Предыдущие примеры были немного медлительными и неточными, включая файлы, которые не нужно загружать раньше. Отличие здесь состоит в том, что если файл включен, запуск теста будет прерван.

## 13.3.2 Несколько тестов в одном проекте

Последний пример с `karma-web-components` подразумевает незначительное изменение конфигурации Karma. Вместо того чтобы изменить настройку по умолчанию, я хотел оставить обе конфигурации. В репозитории для этой книги на GitHub вы сможете запускать WCT, Karma и Karma с плагином `karma-web-components` из одного и того же проекта.

Эту тему еще долго можно обсуждать, и, хотя одного из приведенных примеров будет достаточно, единого стандартного способа настройки тестирования не существует. У каждого проекта свои потребности, и, скорее всего, вам понадобится поработать, чтобы настроить все по своему вкусу. Рассказывая о различных методах, описанных в этой главе, я надеюсь, что ваши настройки будут достаточно продолжительными, чтобы вы, по крайней мере, смогли изучить все изменения, которые вам нужно сделать.

Вот почему в моем репозитории конфигурация `karma-web-components` находится в файле с именем karma.conf.webcomponents.js. У сценария `npm` для его запуска – другое имя с указанием на новую конфигурацию в параметрах:

```
"karma-wc-test":
  "./node_modules/karma/bin/karma start karma.conf.webcomponents.js",
```

У вас не будет всех этих похожих тестов в одном проекте, но у вас могут быть разные типы тестов с разными вещами. Например, в случае только модульных тестов с чистым JS, без необходимости полагаться на браузер, мне нравится использовать Tape и JSDOM. И я мог бы использовать

Karma, Mocha и Chai для тестов, которые мне нужно запускать в браузере. Моя точка зрения заключается в том, что хотя я изо всех сил старался включить в этот проект избыточные тесты, наличие нескольких отдельных тестовых прогонов в одном проекте вполне нормально.

### 13.3.3 Замечание относительно Safari

И последнее, на что здесь стоит обратить внимание. Разработчики, работающие на Windows, не смогут запускать тесты в Safari в любом случае, а вот у пользователей MacOS это должно получиться. В своих примерах я не устанавливал `karma-safari-launcher`. Как правило, «Safari» – это еще один браузер, который можно добавить в конфигурационный файл Karma. В настоящее время это средство запуска Karma работает не так идеально, если запускать его на компьютерах под управлением ОС macOS Mojave. Safari запустится, но, для того чтобы дать разрешение на загрузку Karma, потребуется вмешательство пользователя. На момент написания этой главы данный вопрос открыт (https://github.com/karma-runner/karma-safari-launcher/issues/29). В репозитории для этой книги на GitHub сейчас вместо `karma-safari-launcher` я использую `karma-safarinative-launcher` в качестве обходного пути. Чтобы использовать этот пользовательский плагин, единственное, что необходимо добавить в файл karma.conf.js, – это следующее:

```
customLaunchers: {
  Safari: {
      base: 'SafariNative'
  }
},
```

Теперь вы можете проводить тестирование с Safari в своем списке браузеров в файле karma.conf.js:

```
browsers: ['FirefoxHeadless', 'ChromeHeadless', 'Safari'],
```

В идеале, однако, этот обходной путь не понадобится, поскольку проблема будет решена в исходном пакете плагина. До тех пор можно использовать это!

## Резюме

Из этой главе вы узнали:

- о различных стилях тестирования и преимуществах TDD при написании компонентов;
- о трех разных способах запуска тестов, чтобы продемонстрировать разнообразие доступных опций;
- о способах «думать модулями» и о том, как разрабатывать тесты для каждого из них.

# 14

# *События и поток данных приложения*

***Эта глава охватывает следующие темы:***

- создание собственных пользовательских событий в сравнении с использованием нативных событий DOM;
- всплывание событий для двух типов событий, включая использование параметра `composed` для прохождения сквозь теневую модель DOM;
- событие `WebComponentsReady` и `customElements.isDefined` для обработки синхронизации;
- использование централизованной модели данных и шины событий для обработки потока данных в приложении.

По мере того как мы приближаемся к концу нашего путешествия по веб-компонентам, остается не так много тем для освещения с точки зрения возможностей веб-компонентов. Тем не менее если сравнивать то, что мы уже узнали, с современным фреймворком, может показаться, что в некоторых областях их недостаточно.

## 14.1 Использование фреймворков

Хотя функции веб-компонентов теперь являются частью стандартных веб-спецификаций, такие вещи, как привязка данных, маршрутизация и шаблоны проектирования приложений в стиле «модель–представление–контроллер» (MVC), таковыми не являются! Если честно, было бы глупо, если бы это было так. Сеть – это огромное пространство, и не все мы делаем приложения. Даже если бы это было так, разработчики приложений обычно выбирают подходящий шаблон проектирования для конкретного проекта. Встроенные средства, которые поддерживают определенные способы разработки приложений, вряд ли будут приветствоваться.

Фактически у нас почти были до невероятия базовые основы для связывания данных, когда мы использовали метод `Object.observe`, позволявший прослушивать изменения в JS-объекте (рис. 14.1). Однако популярные фреймворки в итоге не приняли его, потому что он не соответствовал их конкретным решениям для связывания данных и управления состоянием приложений.

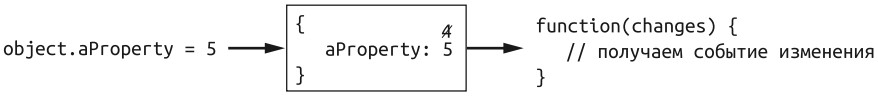



**Рис. 14.1 Устаревшая функция связывания данных: `Object.observe`**

Несмотря на то что эти типы функций не являются частью веб-компонентов, написанных на чистом JS, и это имеет смысл, фреймворк может и будет выбирать излюбленный способ предлагать их, особенно когда типичному пользователю, ориентированному на приложения, нужны все или некоторые из них. Это позволяет создавать отличные учебные пособия и удивительные примеры, которые тесно связаны друг с другом и отлично подходят для начинающих.

Учитывая разнородный характер использования веб-компонентов, единого шаблона проектирования приложения, который бы управлял ими всеми, вероятно, никогда не будет. Вы не увидите, как огромное количество блогов использует те же шаблоны проектирования, которые применяются для веб-компонентов, по крайней мере в ближайшее время. Это отсутствие кристально чистого направления – причина, по которой веб-компоненты могут чувствовать, что им чего-то не хватает, особенно при принятии проектного решения и неизбежном сравнении фреймворка с любым решением, не использующим фреймворки, включая веб-компоненты.

Однако есть и хорошие новости. Общие функции современных фреймворков теперь более взаимозаменяемы, чем когда-либо. Взять, к примеру, Redux, библиотеку, предназначенную для управления состоянием приложения. Redux настолько сильно связана с React, что вы мо-

жете поверить, что их нельзя разделять. Некоторые считают, что если вы используете React, то должны использовать и Redux. И наоборот, можно предположить, что Redux нельзя использовать нигде, кроме React. Это – артефакт тех удивительных демонстраций, постов в блогах и учебных пособий, созданных сообществом React, соединяющих их. На самом деле некоторые разработчики React начали применять другие библиотеки для управления состоянием, такие как MobX.

Я хочу сказать, что существует широкий спектр взаимозаменяемых решений, которые популярные фреймворки уже используют; просто это не сразу видно при изучении фреймворка. Постоянная попытка заставить библиотеку вроде Redux работать в React, Angular, Vue и чистом JS только поможет нам, разработчикам веб-компонентов.

Более того, вам может и не понадобиться сложная библиотека, наподобие Redux, для управления приложением. В этой главе мы усовершенствуем планировщик упражнений из главы 10, чтобы использовать чрезвычайно простые и специально разработанные шаблоны проектирования приложений. Вам, наверное, интересно, зачем вообще заморачиваться шаблонами проектирования или с чего начать. Как и любое решение, все начинается с проблемы, которую нужно решить.

## 14.2 События

Я думаю, было бы справедливо сказать, что в любой сложной архитектуре приложения или фреймворке существует некий особый обмен сообщениями. Поверх этого может быть построено намного больше движущихся частей, но на самом базовом уровне все начинается с какого-то события, такого как нажатие кнопки или изменение поля ввода. Чтобы сообщить об этом изменении, генерируется сообщение, а затем в результате этого события что-то происходит. Итак, начнем с самого начала.

### 14.2.1 Нативные события и WebComponentsReady

Не так много нужно сказать о нативных событиях. Мы использовали их на протяжении всей книги, когда слушали нажатия кнопок, изменения элементов ввода и т. д. Это события, которые браузер генерирует сам с помощью DOM.

Поскольку вы уже знакомы с нативными событиями, я хочу привести сомнительный пример такого события, относящийся к коду, который можно найти при изучении веб-компонентов. Я не упоминал о нем, потому что он не особо нужен для ежедневного использования; официально это даже не функция веб-компонента, и он доступен только через полизаполнение. Тем не менее он используется как нативное событие, поэтому давайте рассмотрим его! В приведенном ниже листинге приводится чрезвычайно простой пример для тестирования события `WebComponentsReady`.

**Листинг 14.1 Событие `WebComponentsReady`**

```
<html>
<head>
   <title>Web Components Ready</title>
   <script src="https://unpkg.com/@webcomponents/webcomponentsjs@2.0.0/ ◄— Полифил
               webcomponents-loader.js"></script>
   <script>
       document.addEventListener(  ◄———————— Слушатель событий WebComponentsReady
               'WebComponentsReady', function(e) {
           console.log('components ready');
       });
   </script>
</head>
</html>
```

Это событие просто позволяет вам получать уведомления, когда веб-компоненты в целом готовы на странице. На самом деле готовность означает, что любые созданные определения веб-компонентов могут быть применены к элементам на странице. Например, созданный вами пользовательский элемент `<sample-component>` – это `HTMLUnknownElement` без определения с помощью метода `customElements.define`.

Однако я сказал, что это сомнительный пример. По-настоящему нативные события генерируются браузером. Хотя это событие используется так, как если бы оно было сгенерировано браузером, это не так. Оно генерируется полифилом. Можно ли назвать его нативным, если оно используется как таковое и выглядит так же? Более того, оно так похоже на базовую функцию веб-компонента, когда используется, и это может сбивать с толку – можно ли ожидать, что оно будет срабатывать даже при отсутствии полифила. Почему оно доступно только при полизаполнении?

Когда браузер нативно поддерживает веб-компоненты, в частности API пользовательских элементов, компоненты готовы сразу. Однако когда такой поддержки нет и используется полифил, требуется время, чтобы загрузить полифил и разрешить веб-компонентам определять себя. Ситуация была еще хуже, когда HTML-импорт был предпочтительным способом создания веб-компонентов. Для загрузки необходимого HTML-импорта также требуется время. В ожидании всего этого, возможно, было бы неразумно взаимодействовать с компонентами на странице. Событие `WebComponentsReady` позволяет узнать, когда это безопасно.

Звучит полезно, но на самом деле это редко используется. С одной стороны, при использовании современного браузера, где полифилы не нужны, веб-компоненты будут готовы незамедлительно. Во-вторых, такие проблемы с синхронизацией, похоже, возникают нечасто. В этой книге мы организовали наши веб-компоненты, создав один большой компонент, который представляет приложение и включает в себя дочерние компоненты. Хотя мы взаимодействуем с этими дочерними веб-компонентами и предполагаем, что они готовы с самого начала, они даже не существовали бы, если бы основной компонент приложения еще не был

создан. И конечно же, этот основной компонент создается только тогда, когда готовы веб-компоненты.

Кроме того, событие `WebComponentsReady` не является частью спецификации веб-компонентов. Без включения полифила готовое событие не будет инициировано просто потому, что обычно его не существует!

Несмотря на то что оно не несет особой пользы при повседневном использовании (хотя оно было полезно для некоторых настроек тестирования), я хотел рассказать о нем по нескольким причинам. Во-первых, это событие довольно часто встречается при поиске в сети и в некоторой степени сбивает с толку, когда нужно определить, является ли оно событием реального веб-компонента или оно просто из полифила. Во-вторых, это демонстрирует «нативное событие». Хотя было бы немного странно называть его нативным из-за того, что оно было сгенерировано из полифила, оно по-прежнему исходит из DOM, в частности из объекта `document` в нашем примере. Далее мы сопоставим его с пользовательскими событиями. Наконец, это дает нам возможность перейти к связанной функции веб-компонента (на этот раз реальной).

### 14.2.2 Когда определяются пользовательские элементы

Я немного растягиваю, чтобы включить тему «промисов» в раздел о событиях. Но если вы знаете, что такое промис, он выполняет то же самое, что и событие; это позволяет нам говорить о более подходящем способе решения проблем синхронизации веб-компонентов, если мы с ними столкнемся.

Промисы – весьма базовая функциональность JS. Создать их с нуля легко, а использовать уже существующий еще проще. В этой книге мы рассмотрели две из трех функций API пользовательских элементов:

- `customElements.define(<имя тега>, <класс>, <параметры>)` позволяет нам дать жизнь и назначить поведение имени пользовательского элемента;
- `customElements.get(<имя тега>)` возвращает класс, ассоциированный с существующим пользовательским тегом. Мы использовали его, чтобы установить, был ли пользовательский элемент уже определен. Подсказка: если возвращено undefined, он не определен;
- `customElements.whenDefined` – последний метод, и сейчас мы его обсудим.

Вместо того чтобы получить готовое событие к тому моменту, когда веб-компоненты в целом готовы, возможно, более целесообразно слушать, когда конкретный веб-компонент определен, особенно когда можно отложить загрузку JS-модулей или целых наборов, поскольку пока мы не используем определенный набор компонентов. Возможно, мы захотим получить уведомление, когда определенные компоненты будут готовы, даже если другие компоненты работали в течение достаточно долгого времени.

Учитывая это, можно посмотреть на `customElements.whenDefined` в действии в приведенном ниже листинге.

**Листинг 14.2 Использование промиса, чтобы установить, когда определен конкретный компонент**

```
<html>
<head>
   <title>Custom Elements When Defined</title>
   <script>
       class SampleComponent                    <-- Очень простой класс компонента
               extends HTMLElement {
           constructor() {
               super();
               this.attachShadow({mode: 'open'});
               this.shadowRoot.innerHTML = 'my component';
           }
       }

       setTimeout( function() {                 <-- Ждем 2 секунды, а затем определяем компонент
           if (!customElements.get('sample-component')) {
               customElements.define('sample-component', SampleComponent);
           }
       }, 2000);

   customElements.whenDefined(                  <-- Создаем промис, который предупреждает
     'sample-component').then( ()=>{                нас об определении веб-компонента
       console.log('defined now!');
   });
   </script>
</head>
<body>
<sample-component></sample-component>
</body>
</html>
```

Благодаря этой последней функции `customElements` у нас есть реальный способ предпринять или отложить действие, пока мы не узнаем, что определенный веб-компонент работает должным образом.

### 14.2.3 Пользовательские события

Нативные события – отличная вещь, но если вы занимаетесь веб-разработкой, то, вероятно, уже давно используете их в повседневной работе для таких простых вещей, как прослушивание события щелчка мыши. Поэтому в них нет ничего захватывающего.

Пользовательским событиям уже несколько лет отроду, и они ни для кого не секрет; но если они не являются частью вашей веб-разработки, вероятно, следует воспользоваться ими! Прежде чем перейти к изучению обмена сообщениями в приложении, где мы будем использовать пользовательские события, давайте кратко рассмотрим некоторые основные способы использования.

Что именно они позволяют вам делать? Равно как и нативные события, они позволяют получать сообщения и принимать меры. Как и событие щелчка мышью, пользовательское событие можно слушать и что-то

делать, когда оно срабатывает. В отличие от нативных событий, мы можем сами генерировать и инициировать событие. Например, в палитре цветов, с которой мы работали в предыдущей главе, вместо того чтобы наблюдать за изменениями атрибутов, как мы это делали, мы могли бы создать собственное событие, которое запускается, когда цвет или альфа-канал компонента изменяются в результате использования нами компонента. С помощью пользовательских событий мы контролируем содержимое события, имя события, которое нужно слушать, и синхронизацию при запуске.

При работе с пользовательскими событиями необходимо выполнить три основных действия. Первое – создать событие. Мы будем делать это с помощью конструктора `CustomEvent`. Конструктор принимает имя события в качестве первого параметра, а затем еще один, необязательный объект с параметрами события и подробностями в качестве второго параметра:

```
const event = new CustomEvent('myevent', { detail: { message: 'hi', number: 5 } });
```

Так же, как и в нативных событиях, у нас имеется тот же самый объект `detail` для пользовательских пар типа «ключ/значение». Свойство `detail`, по-видимому, нечасто используется большинством людей, когда речь идет о нативных событиях. Поскольку браузер генерирует нативные события, все планируется заранее с помощью конкретных пар типа «ключ/значение» в самом объекте события. Например, можно получить местоположение клика из свойств `clientX` и `clientY` с помощью обычного нативного события клика:

```
document.addEventListener('click', function(e) {
   console.log(e.clientX, e.clientY)
})
```

Мы также можем получить свойство `e.detail`. Это свойство, особенно в случае со щелчком мышью, содержит количество нажатий на элемент. Кажется случайностью, что это значение находится в переменной с именем `detail`, но это именно то, чем оно и является: случайным. Может показаться, что свойство `detail` содержит все необходимые данные о пользовательских свойствах и, возможно, не было должным образом спланировано с самого начала.

В случае с нативными событиями это кажется немного странным, ведь тут все контролирует браузер. Но свойство `detail` является основной концепцией для пользовательских событий, где у нас есть пользовательские данные, которые необходимо передать, как, например, наши предыдущие данные.

После того как событие было создано, оно никуда не денется, пока мы не запустим или, выражаясь более формально, не *отправим* его. Как и нативные события, пользовательские события отправляются из DOM (документа или элементов внутри). Учитывая это, давайте отправим событие из документа:

```
document.dispatchEvent(event);
```

Последний базовый этап для работы с пользовательскими событиями – это прослушивание события. Здесь его можно рассматривать как любое нативное событие, с которым вы когда-либо работали, только с уникальным именем и пользовательскими данными в объекте `detail`. Мы должны настроить слушателя до того, как отправим событие, чтобы он уже был готов, когда событие будет инициировано. В приведенном ниже листинге показан весь пример целиком.

**Листинг 14.3 Создание, диспетчеризация и прослушивание пользовательского события**

```
<html>
<head>
   <title>Custom Events</title>
   <script>
       document.addEventListener(                    <-- Слушатель событий и функция обратного вызова
               'myevent', function(e) {
          console.log('The message', e.detail.message, 'with number',
                       e.detail.number);
       });

       const event = new CustomEvent(                <-- Создаем новое пользовательское событие
               'myevent', {
                   detail: {                         <-- Пользовательский объект detail, содержащий
                    message: 'hi',                       свойства, через которые мы проходим
                    number: 5 }
       });

       setTimeout( function() {                      <-- Инициируем/отправляем событие, но ждем
           document.dispatchEvent(event);                2 секунды, чтобы доказать, что обратный
       }, 2000);                                         вызов ждет его

   </script>
</head>
<body>

</body>
</html>
```

Поскольку в этом примере мы регистрируем выходные данные в нашу консоль, если вы попытаетесь запустить этот код, откройте его в инструментах разработчика вашего браузера. Пользовательские свойства `message: 'hi'` и `number: 5` переносятся вплоть до console.log, когда инициируется обратный вызов события. Мы также можем отправить событие, используя таймер. Просто в этом примере он лучше доказывает нам, что обработчик события фактически запускается из самого события.

## 14.2.4 Всплытие пользовательского события

Отправка и прослушивание события из одного элемента – это прекрасно, но один нюанс, касающийся пользовательских событий, заключается в том, что по умолчанию они не «всплывают». Когда мы говорим *всплыва-*

*ем*, это означает, что событие проходит через множество уровней DOM, и любой из них можно прослушать. Например, при нажатии на кнопку щелчок проходит через кнопку, а затем к родителю, к родителю родителя, вплоть до корня страницы. Каждый элемент, через который проходит щелчок, может генерировать собственное событие щелчка, если вы решите его прослушать.

Нативные события, такие как событие щелчка, делают это по умолчанию. Пользовательские события – нет. Поэтому нужно помочь им в этом. Мы продемонстрируем это на практике в приведенном ниже листинге, добавив дополнительный элемент для пользовательского события, через который нужно пройти.

**Листинг 14.4 Всплывание пользовательского события**

```
<html>
<head>
   <title>Custom Event Bubbling</title>
   <script>
       document.addEventListener('myevent', function(e) {
           console.log('The message', e.detail.message, 'with number',
                       e.detail.number);
       });

       const event = new CustomEvent('myevent', {
           bubbles: true,                          <-- Включаем всплывание
           detail: {
               message: 'hi',
               number: 5
           }
       });

       setTimeout( function() {                    <-- Отправляем элемент через другой тег <div>
           document.getElementById('target')
                       .dispatchEvent(event);
       }, 2000);

   </script>
</head>
<body>
   <div id="target"></div>                         <-- Дополнительный тег <div> для передачи события
</body>
</html>
```

Код в листинге 14.4 будет работать, но только потому, что мы добавили опцию `bubbles: true` ко второму параметру пользовательского события. Без этого, если вы решите закомментировать это в данном примере, событие запустится и остановится на теге `<div>` с идентификатором `"target"`. Если бы мы добавили слушателя событий к этому тегу `<div>` вместо документа, все бы работало прекрасно, потому что событию не нужно было всплывать, чтобы мы его перехватили.

## 14.3 Передача событий через веб-компоненты

Хотя нативные и пользовательские события – понятия довольно простые, если рассматривать их на уровне функций, стратегии передачи событий в целом могут быть многочисленными и сложными. Одна из этих сложностей возникает при работе с веб-компонентами и теневой моделью DOM.

Приведенный ниже листинг начинается с простого примера веб-компонента, в котором не используется теневая модель DOM. Здесь есть только кликабельная кнопка.

**Листинг 14.5 Веб-компонент со слушателем щелчков**

```
<html>
<head>
   <title>Web Component Events</title>
   <script>
       class SampleComponent extends HTMLElement {
           connectedCallback() {
               this.innerHTML =                    <-- Кнопка внутри веб-компонента
                   '<button>Click me</button>';
           }
       }

       customElements.define('sample-component', SampleComponent);

       document.addEventListener(                  <-- Слушатель событий щелчка
               'click', function(e) {
           console.log('was clicked', e.target, e.currentTarget,
                       e.composedPath());          <-- Консоль регистрирует источник и путь события
       });
   </script>
</head>

<body>
<sample-component></sample-component>
</body>
</html>
```

Как правило, в этой книге мы слушали события щелчка и подобные вещи внутри самого компонента. Однако в листинге 14.5 мы слушаем событие клика для всего документа, но это событие возникло внутри компонента. Мы полагаемся на способность события щелчка проходить через компонент к документу.

Нажатие кнопки мыши в любом месте страницы инициирует обратный вызов. Но если событие может прийти откуда угодно, может ли этот крайне неспецифичный слушатель событий быть полезным, если вы не знаете, откуда пришел щелчок?

Оказывается, можно точно узнать, где он был инициирован. При нажатии на кнопку в нашем веб-компоненте свойство `event.target` регистрируется как `<button>Click me</button>`. `target` – первая остановка клика, прежде чем событие всплывет в остальных местах.

Также не всегда полезно знать, откуда пришел щелчок, – скорее, нам нужно знать фактический элемент, к которому мы добавили событие. Мы можем получить это, используя свойство `event.currentTarget`. В нашем примере мы добавляем событие в документ, и это именно то, что мы получаем, когда регистрируем `event.currentTarget`.

Метод `event.composedPath()` также представляет интерес. Обратите внимание, что это функция, а не свойство. При вызове этой функции мы вернем полный путь, через который проходит событие. В нашем примере мы получим это:

```
[button, sample-component, body, html, document, Window]
```

Обратите внимание, что оно начинается с кнопки, которую мы нажали, проходит через веб-компонент, через элементы `body` и `html` и объект `document` и заканчивается в корне всего: объекте `Window`. Опять же, обратите внимание, что мы пока еще не используем теневую модель DOM!

## 14.3.1 Распространение нативных событий с помощью теневой модели DOM

Вспомните два режима теневой модели DOM, упомянутых ранее, закрытый и открытый. Если вы помните, закрытый режим не рекомендуется, потому что он только все усложняет, не предлагая при этом никакой реальной безопасности, как предполагалось. Несмотря на это, стоит упомянуть еще одну небольшую разницу между закрытым и открытым режимами.

В приведенном ниже листинге мы немного изменили простой веб-компонент, чтобы добавить теневую модель DOM. Все остальное то же самое, включая кнопку и слушателя событий.

**Листинг 14.6 Веб-компонент с теневой моделью DOM и слушателем события щелчка**

```
<html>
<head>
   <title>Shadow DOM Events</title>
   <script>
       class SampleComponent extends HTMLElement {
           connectedCallback() {
               const shadow = this.attachShadow(
                       {mode: 'open'});
               shadow.innerHTML = '<button>Click me</button>';
           }
       }

       customElements.define('sample-component', SampleComponent);

       document.addEventListener('click', function(e) {
           console.log('was clicked', e.target, e.currentTarget,
                       e.composedPath());
       });
   </script>
```

На этот раз используется теневая модель DOM

```
</head>

<body>
<sample-component></sample-component>
</body>
</html>
```

Теперь, глядя на журнал консоли, видно, что `event.currentTarget` остался тем же: это документ, к которому мы добавили слушателя событий. А вот `event.target` выглядит иначе. Он отображается как элемент веб-компонента `<sample-component>`. Теневая модель DOM скрывает тот факт, что щелчок исходил от кнопки внутри компонента, хотя и делает это не полностью. Когда мы смотрим на функцию `event.composedPath()`, то видим это:

```
[button, document-fragment, sample-component, body, html, document, Window]
```

За исключением дополнительного слоя фрагмента документа, который представляет теневую границу, все то же самое. Мы видим, что событие происходит на теге `<button>` внутри компонента и распространяется через него.

Однако если мы изменим режим теневой модели DOM на `closed`, то даже не получим полный вариант метода `composedPath`. Он начинается с элемента `<sample-component>`:

```
[sample-component, body, html, document, Window]
```

Когда мы имеем дело с закрытым, а не открытым режимом Shadow DOM, это кажется совершенно нормальным, верно? Открытый режим все закрывает, чтобы случайно не сделать ничего плохого, но он достаточно открыт, чтобы иметь обходные пути, если вы знаете, что идете против намеченного рабочего процесса. Напомним, что у нас нет возможности использовать метод `querySelector` для элементов внутри теневой модели DOM извне, кроме случаев, когда мы проходим через свойство `shadowRoot`. Выглядит довольно похоже. Можно посмотреть на `event.target`, чтобы увидеть, где происходит событие, но когда теневая модель DOM не позволяет нам получить полную картину, мы можем посмотреть на функцию `event.composedPath()`. Закрытый режим усложняет все это, хотя на самом деле он не является безопасным.

## 14.3.2 Распространение пользовательских событий с помощью теневой модели DOM

У пользовательских событий есть еще одна вещь, чтобы скрыться от теневой модели DOM. Помимо необходимости установить для `bubbles` значение `true`, чтобы событие всплывало через DOM, нам также нужно установить для свойства `composed` значение `true`. Без этого событие просто не всплывет из теневой модели DOM и не будет обращаться к родительским компонентам или элементам.

Приведенный ниже листинг основан на последнем примере, где мы передали событие щелчка через теневую модель DOM. Вместо того что-

бы передавать событие напрямую, мы будем слушать его в компоненте и генерировать пользовательское событие, которое и будем передавать.

**Листинг 14.7 Передача пользовательского события через теневую модель DOM**

```
<html>
<head>
   <title>Shadow DOM Custom Events</title>
   <script>
       class SampleComponent extends HTMLElement {
           connectedCallback() {
               const shadow = this.attachShadow({mode: 'open'});
               shadow.innerHTML = '<button>Click me</button>';
               shadow.querySelector('button').addEventListener(
                   'click', e => {
                       const customEvent = new CustomEvent('myclick', {
                       bubbles: true,
                       composed: true,
                       detail: {
                              message: 'hi',
                              number: 5
                       }
                   });
                   shadow.dispatchEvent(customEvent);
               })
           }
       }

       customElements.define('sample-component', SampleComponent);

       document.addEventListener(
               'myclick', function(e) {
           console.log('was clicked', e.target, e.currentTarget,
                       e.composedPath());
       });
   </script>
</head>

<body>
<sample-component></sample-component>
</body>
</html>
```

При нажатии кнопки генерируется новое пользовательское событие для отправки

Прослушиваем пользовательское событие myclick

Хотя этот пример работает просто отлично и сообщение регистрируется, если `bubbles: true` или `composed: true` удалить, все отключится. Ранее в этой главе я рассказывал о `bubbles: true`. Поскольку пользовательские события не всплывают по умолчанию, они никогда не выберутся из веб-компонента, используется теневая модель DOM или нет.

Свойство `composed` требуется в дополнение к `bubbles`. То, что событие всплывает, не означает, что оно пройдет через теневую границу. Это свойство позволяет прорвать границу.

Фактически регистрируется точно то же, что и раньше, за исключением составного пути при использовании открытого режима теневой

модели DOM. Конечно, кнопка больше не является частью пути, потому что пользовательское событие теперь генерируется из теневого корня компонента. Поэтому сейчас зарегистрированный метод `event.composedPath()` – это

```
[document-fragment, sample-component, body, html, document, Window]
```

Всплытие событий – отличная вещь, но у него могут иметься серьезные недостатки при попытке передачи сообщений объектам, которые не являются частью одной и той же родословной. Мы рассмотрим это чуть позже, поскольку в действительности это проблема архитектуры приложения. В качестве первого шага давайте усовершенствуем приложение Workout Creator из главы 10, чтобы сделать его пригодным для использования в качестве реального приложения.

## 14.4 Разделение данных

Давайте вспомним, что нам уже удалось сделать, работая над приложением Workout Creator из главы 10. Как подчеркивалось в этой главе, при работе с теневой моделью DOM и стилями функционал внутри был в основном визуальным. Мы создали список с упражнениями на выбор в левой половине приложения. Каждое упражнение из списка представляло собой веб-компонент упражнения, заполненный уникальными данными. При нажатии мышью на любом из них упражнение добавляется в план в правой части приложения.

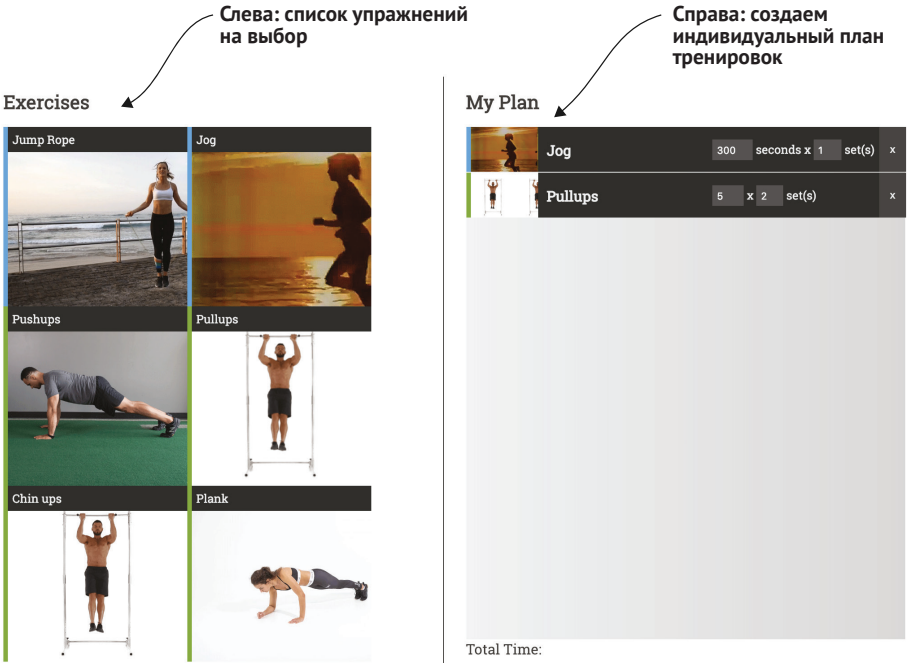



**Рис. 14.2 Вспомним, как выглядит приложение Workout Creator**

Попадая в план тренировок, каждое упражнение получало некий пользовательский интерфейс для настройки количества раз выполнения каждого повторения или задания либо настройки общего количества секунд, необходимого для выполнения упражнения. Также справа от упражнения была кнопка, чтобы удалить его из вашего плана. Хотя и не так много из этого интерфейса было функциональным!

Если вернуться к компоненту `exerciselibrary`, показанному в левой части рис. 14.2, он не делает ничего особенного. Он просто представил список доступных упражнений. Все щелчки мышью, которые делаются, чтобы добавить упражнение, здесь также не обрабатывались – событие передавалось компоненту `workoutcreatorapp`. Мы скоро это изменим и обработаем событие должным образом, но дело в том, что компонент `exerciselibrary` почти ничего не делает. Он показан в приведенном ниже листинге. Длинный список упражнений мы сократили.

**Листинг 14.8 Компонент `exerciselibrary`**

```
import Template from './template.js';

export default class ExerciseLibrary extends HTMLElement {
   constructor() {
       super();
       this.attachShadow({mode: 'open'});
       this.shadowRoot.innerHTML = Template.render([
           { label: 'Jump Rope', type: 'cardio', thumb: '', time: 300, sets: 1},
          ... more exercises ... ]);
   }
}

if (!customElements.get('wkout-exercise-lib')) {
   customElements.define('wkout-exercise-lib', ExerciseLibrary);
}
```

Сокращенный список упражнений

Хотя размещение этого огромного массива упражнений в качестве параметра функции, которая отображает HTML-код, работает, это плохая практика. Если вы хотите добавить другое упражнение или изменить то, что у вас есть, как человек, не знакомый с проектом, узнает, где искать? Почему этот список обязательно должен быть в этом конкретном компоненте, а не в `workoutcreatorapp` или существовать как статический метод чтения из компонента упражнения?

## 14.4.1 Модель–представление–контроллер

Ни один из этих компонентов особо не подходит для хранения подобных данных. Концепция такого шаблона проектирования, как MVC, заключается в том, что данные приложения, пользовательский интерфейс и управляющая логика разделяются на три основных компонента: модель, представление и контроллер. Хотя в этой книге мы не уделяли пристального внимания MVC, мы уже отделили представление от логики нашего контроллера.

Дадим определение этих терминов. *Представление* является презентационным. Это визуальный аспект вашего приложения, компонента

или т. д. Учитывая, что мы имеем дело с веб-приложением, представление, вероятно, будет HTML-кодом и стилями. Мы использовали отдельный импорт шаблонов в нашем недавнем веб-компоненте, который содержит только HTML-код и стили или представление.

*Контроллер* – это фрагмент посередине. Он будет обрабатывать всю логику, находящуюся между вашей моделью и представлением. Он реагирует на изменения в пользовательском интерфейсе, такие как нажатия кнопок или изменения полей ввода, и обновляет модель соответствующим образом. *Модель* в данном случае – просто наш список упражнений.

Модели обычно содержат набор данных, обеспечивают доступ к этим данным или делают и то, и другое. Объект JSON *может* быть моделью данных, но тогда разработчик должен знать, как взаимодействовать с базовыми данными. Например, перечисление всех имен пользователей из гипотетической модели может показаться простой задачей, пока вы не поймете, что объект JSON немного странный, и вам нужно будет перебрать множество объектов и дочерних объектов, найти объект name, а затем объединить свойства first name и last name. В этом случае модель данных может предоставить вам хорошую функцию для этого. Или, если данные должны поступать удаленно через REST API, модель данных может обрабатывать сетевые запросы, чтобы вы получили именно то, что вам нужно, посредством какой-нибудь асинхронной функции, которая выглядит как простая намеченная функция.

MVC и шаблоны проектирования в целом не всегда (или обычно, по моему опыту) так четко выражены. Мы используем их в качестве ссылок для извлечения, придаем стимул архитектуре нашего приложения и передаем эти идеи нашей команде, но никогда не придерживаемся их любой ценой или в ущерб здравому смыслу.

Например, в этом компоненте наш контроллер не реагирует ни на какие изменения пользовательского интерфейса (пока). Наши данные тоже не меняются. Так что на самом деле в этом конкретном примере контроллер не делает ничего особенного с видом или моделью, кроме как передает его. Однако это не означает, что мы не можем черпать вдохновение из шаблона проектирования MVC. Мы можем удалить данные из нашего контроллера или класса компонента для правильного разделения.

Давайте создадим новую папку в нашем источнике проекта с именем data. Здесь мы создадим новый JS-файл с именем exerciselibrary.js. На рис. 14.3 показана новая структура проекта, а в листинге 14.9 приведена новая модель данных exerciselibrary.js.

**Листинг 14.9 Модель данных для библиотеки упражнений**

```
export default {
   get all() {  <-------- Метод чтения для извлечения упражнений
       return [ <-------- Актуальный (сокращенный) список данных
           {
               label: 'Jump Rope',
               type: 'cardio',
```

```
                thumb: '',
                time: 300,
                sets: 1
            }
        ... другие упражнения ...
        ]
    }
}
```


**Рис. 14.3 Структура проекта с новой папкой data**

Когда мы выделяем этот файл в новую папку data, становится совершенно очевидно, куда отправится другой разработчик, чтобы редактировать упражнения. Также важно то, что мы могли бы расширять компонент, добавляя новые функции по мере необходимости, и он не превратился бы в спагетти-код из логики и данных. Мы также можем свободно размещать что-то в файле данных, используя больше строк. Поскольку файл теперь просто содержит данные, читабельность этих данных здесь важнее, в отличие от того, что было ранее, когда настройка компонентов и логика были наиболее важными вещами.

Вам, наверное, интересно, почему это должен быть JS. Почему это не может быть файл в формате JSON, который мы загружаем во время выполнения? Конечно же, это мог быть и он! Тем не менее мы уже немного коснулись API, применяя метод чтения `all()`. Если бы у нас было намного больше упражнений, мы могли бы использовать эту модель данных, что-

бы включить фильтры, поиск, разбиение на страницы и многое другое. Кроме того, учитывая, что это не класс и он доступен глобально, мы можем легко использовать эту модель данных в качестве единого источника библиотеки упражнений из любой точки приложения.

Мы также можем включить функцию загрузки в этот модуль, чтобы загрузить этот JSON-файл, или даже извлечь данные из API на базе REST. Хотя мы не будем вводить здесь эти дополнительные функции, это открывает нам путь для редактирования нашего плана тренировок.

### 14.4.2 Локальное хранилище

Хотя дополнительные функции могут быть полезны в библиотеке упражнений, это выходит за рамки простой демонстрации. Чтобы заставить наше приложение функционировать, это действительно может быть статический, неизменяемый список. С другой стороны, планировщик тренировок требует дополнительного внимания.

Напомню, что планировщик тренировок – это редактируемый и настраиваемый список, содержащий персональную программу тренировок. Учитывая, что эти данные представляют собой всего лишь одиночную процедуру тренировки для всего приложения, это также может быть единой централизованной моделью данных, доступной везде, как и библиотека упражнений. Требуется дополнительное внимание, чтобы обеспечить способы добавления, удаления и редактирования элементов, а также сохранения всего списка, что позволяет перезагрузить ранее сохраненный план после выхода из браузера.

Чтобы сделать возможным сохранение и загрузку, мы будем использовать уже давно существующую функцию под названием *локальное*, или *интернет-*, *хранилище*. Использовать его просто. Мы берем строку данных (да, только строки) и сохраняем ее, используя определенное имя или ключ:

```
localStorage.setItem("mykey", "a string");
```

Чтение так же просто:

```
localStorage.getItem("mykey");
```

Любое применяемое имя ключа является уникальным для *источника*. В качестве примера можно привести сайт http://mysite.com, который включает в себя протокол, имя домена и номер порта (порт по умолчанию, если тот не указан). Ключ с именем `mykey` будет возвращать другие данные, если используется на http://mysite.com, в сравнении с http://anothersite.com. Это означает, что мы также можем перечислить все ключи для текущего сайта и не получим обратно кучу ненужных нам вещей:

```
Object.keys(localStorage);
```

Учитывая это, давайте создадим модель данных, подобную библиотеке тренировок, которая позволит нам сохранять, редактировать, загружать, добавлять и удалять данные. В листинге 14.10 есть все эти функции, но самое главное, он отслеживает текущий список базовых упражнений,

который используется в любом месте приложения, в котором он нуждается. Скоро этот аспект станет очень важным, когда мы добавим возможность просмотра и воспроизведения тренировки.

**Листинг 14.10 Центральная модель данных для плана тренировок**

```
export default {
   get saved() {                          // Метод для получения списка всех сохраненных планов упражнений
       const savedplans = [];
       Object.keys(localStorage).forEach(function(key){
           savedplans.push(key);
       });
       return savedplans;
   },
   save(name) {                           // Сохраняем текущий план тренировок в локальное хранилище
       localStorage.setItem(name, JSON.stringify(this._currentWorkout));
   },
   load(key) {                            // Загружаем план тренировок из локального хранилища по имени
       this._currentWorkout = JSON.parse(localStorage.getItem(key));
   },
   edit(id, key, value) {                 // Редактируем конкретный план тренировок, используя уникальный идентификатор для ссылки
       let exercise;
       for (let c = 0; c < this._currentWorkout.length; c++) {
           if (id === this._currentWorkout[c].id) {
               exercise = this._currentWorkout[c];
               exercise[key] = value;
           }
       }
   },
   add(exercise) {                        // Добавляем упражнение в план тренировок и присваиваем ему уникальный идентификатор для последующего использования
       if (!this._currentWorkout) {
            this._currentWorkout = [];
       }
       exercise.id = this.createID();
       this._currentWorkout.push(exercise);
   },
   remove(id) {                           // Удаляем упражнение из плана, на которое указывает идентификатор
       if (!this._currentWorkout) { return; }
       for (let c = 0; c < this._currentWorkout.length; c++) {
           if (this._currentWorkout[c].id === id) {
               const deleted = this._currentWorkout.splice(c, 1);
               return;
           }
       }
   }
}
```

Давайте также добавим еще несколько удобных методов, которые помогут в других частях приложения.

**Листинг 14.10 Центральная модель данных для плана тренировок (*продолжение*)**

```
  clear() {                                    Убираем все упражнения из плана
      this._currentWorkout = [];
  },

  get exercises() {                            Метод чтения для списка плана текущих упражнений
       if (!this._currentWorkout) {
           this._currentWorkout = [];
       }
       return this._currentWorkout;
  },                                           Метод для создания уникального
                                               идентификатора для каждого упражнения
  createID() {
           return 'xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx'.replace(/[xy]/g,
           function(c) {
           var r = Math.random()*16|0, v = c == 'x' ? r : (r&0x3|0x8);
           return v.toString(16);
      });                                      Метод общей продолжительности списка
  },                                           упражнений для отображения времени
                                               в пользовательском интерфейсе
get totalDuration() {
  let ttlTime = 0;
  for (let c = 0; c < this._currentWorkout.length; c++) {
      ttlTime += this.getDurationOfExercise(this._currentWorkout[c]);
  }
  return ttlTime;                              Форматируем количество секунд в более
},                                             удобный для чтения формат, включая часы,
                                               минуты и секунды
formatTime(seconds) {
  return new Date(1000 * seconds).toISOString().substr(11, 8);
}
```

Предыдущие функции сохранения, редактирования, удаления и загрузки были довольно стандартным набором функций, которые помогают управлять списком. Следующий набор функций в продолжении листинга предназначен для добавления некоторых дополнительных справочных функций, используемых как внутри, так и вне этой модели данных.

Создание уникального идентификатора – важная вещь, потому что при добавлении нескольких упражнений, которые могут быть абсолютно одинакового типа, важно иметь возможность различать их при удалении или редактировании определенного упражнения – таким образом мы генерируем уникальный идентификатор каждый раз при добавлении упражнения. Чтобы создать уникальный идентификатор, я просто скопировал из сети код, который генерирует UUID (универсальные уникальные идентификаторы). Эти идентификаторы имеют стандартный формат и чрезвычайно высокую вероятность быть уникальными независимо от того, сколько ID вы генерируете. Наличие такого длинного кода и такого точного форматирования, вероятно, излишне для этого приложения, но его достаточно легко скопировать и включить сюда.

Такие удобные методы, как получение общей продолжительности набора упражнений и последовательное форматирование времени, также важны для централизации. Нет, их не так сложно сделать, но они будут сделаны совсем из разных мест. Важно не повторять подобный код, иначе разные реализации могут случайно стать непоследовательными, поскольку код со временем изменяется. Кроме того, если бы мы вдруг захотели использовать другой формат времени, можно было бы изменить его один раз в этом центральном месте.

Очистка данных кажется слишком простым процессом, для того чтобы иметь здесь отдельную функцию, притом что все, что она делает, – устанавливает данные в пустой массив. Но это дает возможность делать более сложные вещи в функции `clear`, по мере того как ваше приложение будет разрастаться, в то же время позволяя пользователям этой модели данных выполнять те же действия.

Теперь, когда базовая модель данных создана, самое время улучшить наш пользовательский интерфейс для взаимодействия с ней! Чтобы сэкономить время и место, я скажу, что это отличная возможность, для того чтобы самостоятельно добавить класс компонента, хотя если вы застрянете, окончательный проект будет доступен в репозитории для этой книги на сайте GitHub. Тем не менее о главных моментах я расскажу прямо сейчас.

### 14.4.3 Подключение пользовательского интерфейса к модели данных

Кнопки для сохранения, загрузки и очистки списка упражнений можно добавить в HTML-код этого компонента, который находится в файле components/plan/template.js. Кроме того, мы можем даже отобразить меню, чтобы позволить пользователю выбирать из доступных планов тренировок в этом же файле шаблона. Импортируя модуль WorkoutPlanData из `'../../data/workoutplan.js'`, функция может генерировать все названия сохраненных планов и помещать их в список для выбора пользователем, как показано в приведенном ниже листинге.

**Листинг 14.11 Функция для генерации сохраненных планов из модели данных**

```
renderSavedPlans() {
   const saved =
     WorkoutPlanData.saved;
   let options = '<option value="none">Load a saved plan</option>';
   for (let c = 0; c < saved.length; c++) {
       options += `<option value="${saved[c]}">${saved[c]}</option>`;
   }
   return `<select id="menu">
              ${options}
           </select>`;
},
```

Получаем список сохраненных названий планов тренировок

Перебираем каждое название и создаем опцию

Возвращаем окончательное меню с опциями

Как только кнопки и меню станут доступны в HTML-коде, мы можем добавить к ним слушателей событий в файле component/plan/plan.js. Импортируя модуль `WorkoutPlanData` из `'../../data/workoutplan.js'`, мы можем добавить события щелчка в приведенном ниже листинге.

**Листинг 14.12 Слушатели событий клика для сохранения, загрузки и очистки плана**

```
this.dom.saveButton.addEventListener(
     'click', e => {
   WorkoutPlanData.save(this.dom.planName.innerText);
});
```

Сохраняем текущий список упражнений с определенным названием, указанным пользователем

```
this.dom.clearButton.addEventListener(
     'click', e => {
   WorkoutPlanData.clear();
});
```

Очищаем текущий список всех упражнений

```
this.dom.menu.addEventListener('change', e => {
   WorkoutPlanData.load(this.dom.menu.value);
});
```

Когда пользователь выбирает пункт меню из списка сохраненных планов, мы загружаем этот план

На рис. 14.4 показано состояние плана тренировок, после того как мы добавили дополнительный пользовательский интерфейс, но что не изменилось, так это то, как упражнение добавляется в визуальный список. При нажатии на упражнение оно добавляется к компоненту, но поскольку модель новая, она не синхронизирована с визуальными элементами. Например, при попытке удалить добавленное упражнение на самом деле ничего не произойдет, потому что этого упражнения там нет. В этой модели все различные части нашего приложения связаны друг с другом, поэтому обязательно нужно сделать так, чтобы все проходило через это центральное место!

Это легко исправить. Ранее компонент `workoutcreatorapp` (component/workoutcreatorapp/workoutcreatorapp.js) содержал слушателя событий клика и метод `onClick`, который использовал функцию `add` в компоненте плана тренировки, чтобы добавить новое упражнение. Еще в главе 10 я сказал, что это временно. Займемся этим прямо сейчас.

Давайте добавим этого слушателя кликов к компоненту `exerciselibrary` и на этот раз воспользуемся для этого моделью данных. Этот новый компонент показан в приведенном ниже листинге (components/exerciselibrary/exerciselibrary.js):

**Листинг 14.13 Прослушивание событий кликов для упражнений и добавление их в план**

```
import Template from './template.js';
import Library from
'../../data/exerciselibrary.js';
import WorkoutPlanData from '../../data/workoutplan.js';

export default class ExerciseLibrary extends HTMLElement {
   constructor() {
```

Импортируем модель данных «библиотека упражнений»

```
        super();
        this.attachShadow({mode: 'open'});
        this.shadowRoot.innerHTML = Template.render(Library.all);
        this.shadowRoot.addEventListener(
          'click', e => {
            if (e.target.constructor.name === 'Exercise') {
                WorkoutPlanData.add(
                    e.target.serialize());
            }
        })
    }
}
if (!customElements.get('wkout-exercise-lib')) {
    customElements.define('wkout-exercise-lib', ExerciseLibrary);
}
```

Прослушиваем события кликов из библиотеки упражнений, которые всплывают к теневому корню этого компонента

Сериализуем данные из упражнения, по которому кликнули мышью, и добавляем в модель данных

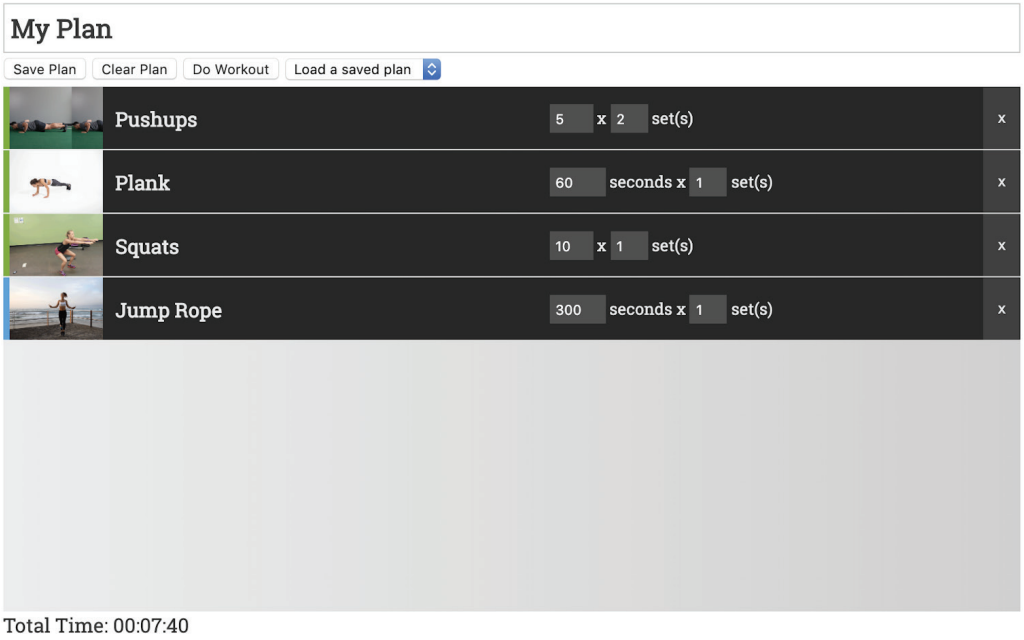



**Рис. 14.4 Дополнительный пользовательский интерфейс для управления списком планов**

Давайте на секунду остановимся и поразмышляем над тем, что мы сделали. Создание централизованной модели данных, к которой можно получить доступ откуда угодно, – это замечательно, но можно ли обойтись без нее? Разумеется! У каждого компонента могут быть собственные данные, как и раньше. Вы хотите запросить все доступные упражнения в библиотеке? Обратитесь к компоненту `exerciselibrary`. Равно как и в случае со списком плана тренировок и компонентом плана тренировок.

Пока что в нашем приложении, где есть несколько компонентов, прослушивать события и взаимодействовать с API каждого компонента

очень легко. На рис. 14.5 показано, с какой легкостью передаются данные между существующими компонентами благодаря своей природе в иерархии DOM.

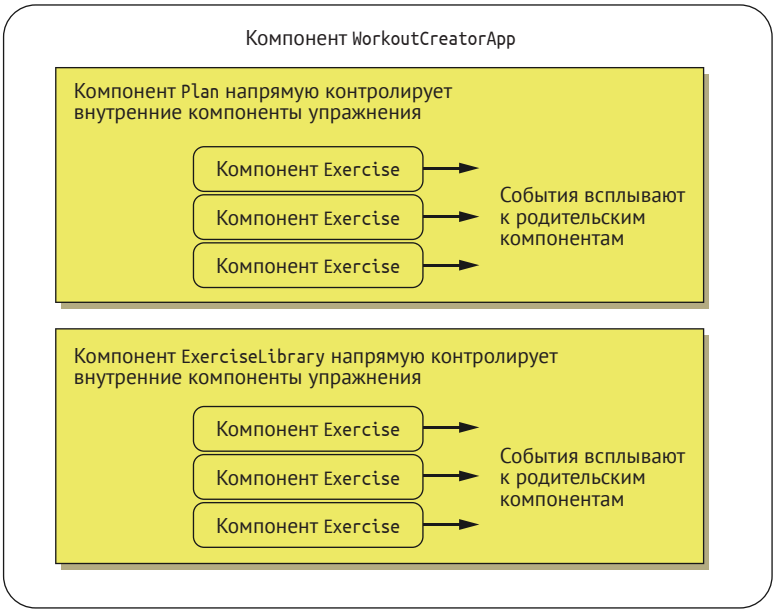



**Рис. 14.5 Простой поток данных с иерархией существующих компонентов**

Однако ситуация может и будет становиться сложной. Что, если бы у нас был еще один компонент, которому нужны те же данные, но он не является частью той же иерархии?

## 14.5 Воспроизведение упражнений

К чему создавать план тренировок, если его нельзя воспроизвести и выполнить какое-нибудь упражнение? Нам нужен один последний компонент, чтобы предложить режим воспроизведения упражнения. Как показано на рис. 14.6, мы превратим плеер в модальное окно, которое появляется при активации.

Поскольку этот компонент является элементом, который должен перекрывать все, что есть на странице, возможно, имеет смысл полностью исключить его из существующего приложения. Наш новый файл index.html всего проекта может выглядеть следующим образом.

**Листинг 14.14 Файл index.html нового проекта с плеером для воспроизведения тренировок**

```
<html>
<head>
   <title>Workout Creator</title>
```

```
    <script type="module"
        src="components/workoutcreatorapp/workoutcreatorapp.js"></script>
    <script type="module" src="components/playback/playback.js"></script>
    <link rel="stylesheet" type="text/css" href="main.css">
    <link href="https://fonts.googleapis.com/css?family=
        Roboto+Slab" rel="stylesheet">
</head>
<body>
    <wkout-creator-app></wkout-creator-app>
    <wkout-playback></wkout-playback>
</body>
</html>
```

Определение компонента воспроизведения тренировок

Воспроизведение тренировки на странице

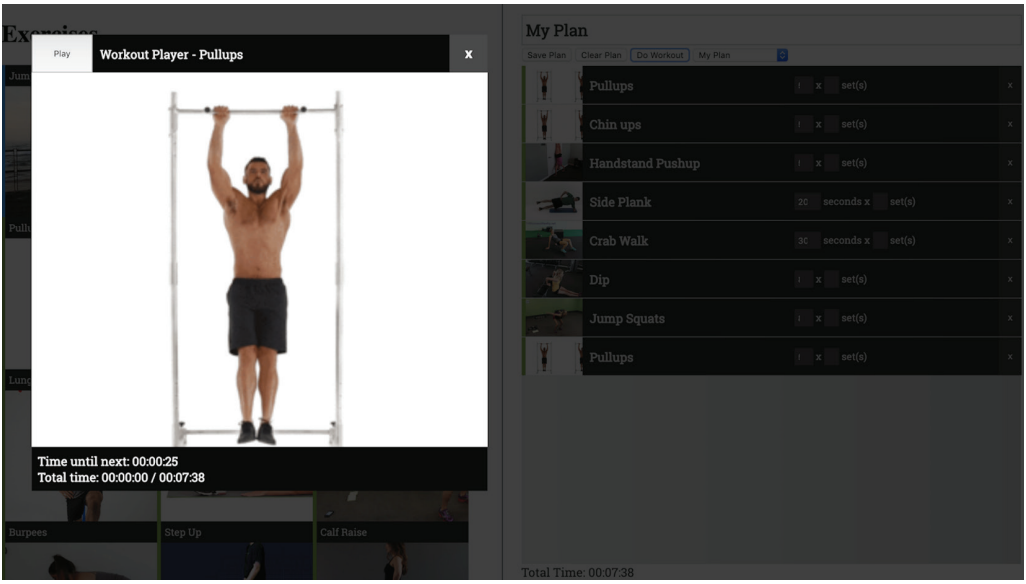



**Рис. 14.6 Модальное окно для воспроизведения тренировок**

Внутренняя работа компонента воспроизведения зависит от возможности воспроизводить и приостанавливать весь список упражнений последовательно, как это делается в списке музыкальных композиций. Чтобы это работало, нужно добавить элементы управления воспроизведением в модель данных плана тренировок (data/workoutplan.js). Эти дополнительные функции показаны в приведенном ниже листинге.

**Листинг 14.15 Дополнительные методы поддержки воспроизведения**

```
play() {
    if (!this._seconds) {
        this._seconds = 0;
    }
    this._timer = setInterval( () => {
        this._seconds ++;
        this.updateTime(this._seconds);
```

Начинаем воспроизведение с помощью таймера

```
    }, 1000);
},
stop() {
    this._seconds = 0;
    clearInterval(this._timer);
},
pause() {
    clearInterval(this._timer);
},
updateTime(seconds) {
    let exercise = this.getExerciseForTime(seconds);
    let exerciseChanged = false;
    if (this._currentExercise !== exercise) {
        this._currentExercise = exercise;
        exerciseChanged = true;
    }
},
```

Останавливаем воспроизведение, отключаем таймер и устанавливаем текущее время на 0

Приостанавливаем воспроизведение путем отключения таймера

Функция обратного вызова таймера для обновления текущего времени и упражнения

Приведенные выше функции являются основными для обеспечения элементов управления воспроизведением, чтобы запускать и воспроизводить упражнение. При проигрывании запускается таймер, заставляя более сложную функцию `updateTime` выяснить, какое упражнение выполняется в настоящее время. Во время паузы таймер останавливается. Функция `stop` останавливает таймер и сбрасывает время.

Далее нам нужно будет предоставить общие функциональные возможности, чтобы получить базовую информацию в любом месте, где она нужна. Даже функции `updateTime` нужно знать, какое упражнение в данный момент выполняется в течение определенного времени. И эта функция, в свою очередь, должна знать продолжительность конкретного упражнения. Итак, давайте приступим.

**Листинг 14.15 Дополнительные методы для поддержки воспроизведения плана тренировок (*продолжение*)**

```
getExerciseStartTime(exercise) {
    let time = 0;
    for (let c = 0; c < this._currentWorkout.length; c++) {
        if (this._currentWorkout[c].id === exercise.id) {
            return time;
        }
        time += this.getDurationOfExercise(this._currentWorkout[c]);
    }
},
getExerciseForTime(seconds) {
    let startTime = 0;
    for (let c = 0; c < this._currentWorkout.length; c++) {
        let duration = this.getDurationOfExercise(this._currentWorkout[c]);
        if (seconds <= startTime + duration && seconds >= startTime) {
            return this._currentWorkout[c];
```

Получаем общее время запуска упражнения, в секундах

Функция для поиска упражнения на определенное время

```
        }
        startTime += duration;
    }
},

get currentExercise() {
    if (!this._currentExercise) {
        this._currentExercise = this._currentWorkout[0];
    }
    return this._currentExercise;
},

getDurationOfExercise(exercise) {
    if (exercise.time) {
       return exercise.time * exercise.sets;
    } else {
        return exercise.estimatedTimePerCount * exercise.count *
      exercise.sets;
    }
}
```

Метод чтения для текущего упражнения

Получаем продолжительность упражнения, передаваемую в качестве параметра

Обновив модель данных, чтобы разрешить воспроизведение, мы можем легко использовать эти функции из компонента воспроизведения. Импортируя модель данных плана тренировок в этот компонент, можно вызывать функции `WorkoutPlan.play()`, `WorkoutPlan.pause()` и `Workout-Plan.stop()`. *См.* репозиторий для этой книги на сайте GitHub, чтобы найти исходный код компонента целиком.

Учитывая размещение этого нового компонента в DOM – за пределами основного приложения прямо на странице, – эта единая и глобальная модель данных имеет большой смысл.

Теперь все стало немного сложнее! На рис. 14.7 показано, как нам нужно прослушивать события и взаимодействовать с API-компонентами с помощью проигрывателя тренировок, изображенного на рисунке.

Работать таким образом вполне возможно, но это определенно раздражает. Без теневой модели DOM мы могли бы передавать события на главную страницу, а затем использовать метод `querySelector`, чтобы выбрать правильный компонент для выполнения действий. При наличии теневой модели DOM мы не можем использовать этот метод внутри дочерних компонентов. Чтобы обойти это, можно было бы создать API для каждого компонента, который должен дать доступ к своим дочерним компонентам, что довольно ненадежно, потому что каждый раз, когда мы выполняем реорганизацию в DOM для внесения визуальных изменений, мы следим за тем, чтобы API соответствовал новой структуре DOM.

Для этого варианта использования и аналогичных сценариев модель глобальных данных может стать неплохим решением! Модель данных определенно не обязательно должна быть глобальной. У каждого компонента может быть собственная модель данных, если это имеет смысл для вашего приложения. До сих пор в простых упражнениях, приведенных в этой книге, не было настоятельной необходимости отделять данные, как сейчас, но, безусловно, ваши компоненты могут быть намного более

сложнее, и именно тогда, в зависимости от вашего проекта, может иметь смысл действительно начать применять MVC или подобный шаблон на уровне компонентов.

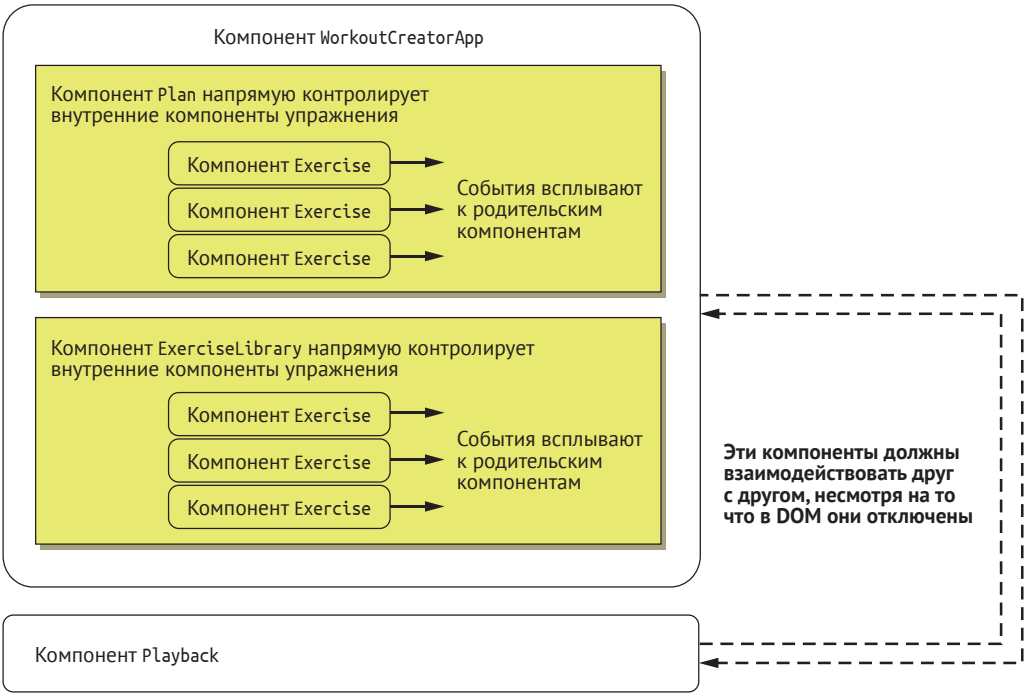



**Рис. 14.7 Взаимодействие компонентов без глобальной модели данных после добавления проигрывателя**

Как я уже говорил ранее в этой главе, использование более надежных решений, таких как Redux или MobX, также может быть отличным вариантом. Как и в случае с любым решением по типу «сделай сам», по мере того как у вас появляется потребность в большем количестве функций, становится все больше смысла использовать решения, проверенные на практике. Однако в нашем простом примере модели данных почти достаточно.

## 14.6 Передача событий с помощью шины

Обратите внимание: я сказал, что модели данных *почти* достаточно. Конечно, мы можем напрямую взаимодействовать с моделью данных, но возьмем, к примеру, воспроизведение. Мы можем вызвать функцию `WorkoutPlan.play()`, но как только включится таймер и начнется воспроизведение, истекшие секунды изменятся, и текущее упражнение будет меняться очень часто. Это не просто воспроизведение, а почти каждый аспект приложения. Изменение продолжительности одного упражнения

должно привести к обновлению общего времени вашей тренировки, так же как добавление и удаление упражнений из плана. Список событий, которые мы должны слушать, длинный.

Ранее в данной главе мы обсуждали передачу пользовательских событий через DOM. Опять же, до появления плеера, все в принципе было в порядке. Так получилось в нашем приложении, что события, которые нам нужно было слушать, всплывали там, где они были нам нужны, потому что иерархия DOM, которой мы следовали, соответствовала необходимому нам потоку данных.

С появлением плеера все усложнилось. Нам нужно было, чтобы все события всплывали вплоть до страницы index.html и каким-то образом передавались своему компоненту, поскольку события не передаются дочерним элементам. Возможно, создание API для плеера тренировок позволит достичь этой цели.

В качестве альтернативы можно сделать нечто похожее на нашу глобальную модель данных. Шина событий – это единый глобальный объект для передачи событий по всему приложению. Кроме того, мы можем использовать те же пользовательские события, которые уже применяли. Вместо вызова `dispatch-Event(mycustomevent)` из компонента мы можем вызвать `EventBus.dispatchEvent(Mycustomevent)`.

Шине событий, как минимум, нужен способ подписки на события и способ их отправки. В приведенном ниже листинге показаны оба этих способа в новом модуле, сохраненном в файле data/ventbus.js.

**Листинг 14.16 Простая шина событий**

```
export default {
   addEventListener(type, cb) {
       if (!this._listeners) {
           this._listeners = [];
       }

       let listener = { type: type, callback: cb };
       this._listeners.push(listener);
       return listener;
   },
   dispatchEvent(ce) {
       this._listeners.forEach( function(l) {
           if (ce.type === l.type) {
               l.callback.apply(this, [ce]);
           }
       });
   }
}
```

Добавляем слушателя событий; передаем тип события и функцию обратного вызова

Добавляем данные слушателя в массив для ссылки при отправке событий

Перебираем всех слушателей

Если тип пользовательского события совпадает с типом слушателя, вызываем функцию и передаем событие

Имена функций, которые я использовал, `addEventListener` и `dispatch-Event`, должны выглядеть знакомо. У них те же имена, что и у методов, которые вы применяли бы для выполнения тех же действий с DOM. Используются те же параметры и возвращаемые значения. Несмотря на то что это нестандартное решение, я думаю, что важно соблюдать после-

довательность при применении шины событий или событий, которые происходят в DOM; просто так намного легче запомнить использование. В приведенном ниже листинге показан фрагмент кода из плеера тренировок, в который добавляется слушатель событий после импорта модуля `EventBus`.

**Листинг 14.17 Добавляем слушателя событий в плеер для получения обновлений временных показателей**

```
EventBus.addEventListener(
    'onPlaylistTimeUpdate', e => {
  if (e.detail.exercise) {
    if (e.detail.exerciseChanged) {
      this.dom.currentExercise.innerHTML = e.detail.exercise.label;
      this.dom.window.style.backgroundImage =
        `url("${e.detail.exercise.thumb}")`;
    }
    this.dom.timer.innerHTML =
      Template.renderTime(e.detail.time, e.detail.exercise);
  }
});
```

Добавляем слушателя событий в шину событий

Просматриваем пользовательское свойство для event.detail, чтобы увидеть, изменилось ли упражнение, и обновляем показатели времени

Обновляем средство индикации времени в компоненте

Конечно, событие нужно отправить откуда-то. Учитывая, что вся логика воспроизведения находится в модели данных плана тренировок, приведенный ниже листинг показывает, что она добавлена туда.

**Листинг 14.18 Отправка событий в шину событий из модели данных плана тренировок**

```
play() {
  if (!this._seconds) {
    this._seconds = 0;
  }
  this._timer = setInterval( () => {
    this._seconds ++;
    this.updateTime(this._seconds);
  }, 1000);
},

updateTime(seconds) {
  let exercise =
    this.getExerciseForTime(seconds);
  let exerciseChanged = false;
  if (this._currentExercise !== exercise) {
    this._currentExercise = exercise;
    exerciseChanged = true;
  }

  let ce = new CustomEvent(
    'onPlaylistTimeUpdate', {
      detail: {
        exercise: this._currentExercise,
        exerciseChanged: exerciseChanged,
```

Вызываем функцию таймера для внутреннего обновления прошедших секунд

Вычисляем текущее упражнение и изменилось ли оно

Создаем событие и отправляем его из шины событий

```
            exerciseIndex: this._currentWorkout.indexOf(this._currentExercise),
            time: seconds,
        }});
    EventBus.dispatchEvent(ce);
},
```

Мы не будем переписывать здесь все приложение, но в репозитории для этой книги на сайте GitHub будут показаны все события, которые были добавлены. Помимо событий, связанных с пользовательским интерфейсом, таких как щелчки мышью, приложение подверглось рефакторингу, чтобы использовать шину событий для передачи всех событий. Однако у меня есть одно последнее улучшение, связанное с типами событий.

### 14.6.1 Статические методы чтения и типы событий

В предыдущем примере, отправляя временные обновления, тип события использовал строку. Точная строка выглядела так: `'onPlaylistTimeUpdate'`. Имя наподобие `onUpdate` тоже подойдет, но поскольку сложность вашего приложения растет, а через шину или даже просто DOM проходит множество событий, поддерживать уникальность своих типов пользовательских событий становится все труднее. Если ссылаться на крайний пример, то давать название вроде `change` немного опасно, потому что это нативное событие DOM, которое регулярно происходит по окончании изменения значения элемента формы; поэтому ваша функция обратного вызова может прийти в замешательство относительно того, какое событие она на самом деле получает, если вы назовете свой тип пользовательского события `change`.

Точно так же у вас может быть несколько пользовательских событий, которые происходят, когда в приложении что-то обновляется. Можно просто назвать их все `onUpdate`, и это звучит заманчиво, но это также приведет к путанице. Вот почему полезно иметь типы событий, которые немного длиннее и более конкретные.

С другой стороны, возможно, не имеет значения, что ваш тип события дифференцирован. Прослушивание `change` можно осуществлять независимо от того, пользовательское это событие или нативное, и, просто зная элемент, вызвавший событие, используя `event.current-Target` или `event.target`, можно получить всю необходимую информацию. В любом случае, типы событий могут быть неспецифичными, как этот, или настолько специфичными, насколько это помогает вашему варианту использования. Помните, что это просто строки, которые представляют тип события, а не сами функции, поэтому они могут быть настолько гибкими, насколько вам нужно, и при этом не нужно беспокоиться по поводу API.

Наличие уникальных событий, которые не приводят к путанице, – именно и есть та причина, по которой мы использовали имя `onPlaylistTimeUpdate`. Проблема с такими более длинными именами событий заключается в том, что при работе с компонентами может быть трудно запомнить, как называется каждое событие. Хуже того, легко можно

сделать ошибку в написании! Проблема с неправильным написанием заключается в том, что события не выдают ошибку. Предполагаемый слушатель просто не будет вызван, потому что вы отправляете или прослушиваете не то событие.

Мы познакомились со статическими методами чтения в главе 3, когда обсуждали функцию веб-компонентов `observedAttributes`. Пользовательские события – еще одно идеальное место для их использования. Поскольку модель данных – это не класс и уже является статической, мы можем просто использовать здесь простой метод чтения:

```
get PLAYLIST_UPDATE_EVENT() { return 'onPlaylistTimeUpdate'; },
```

Если бы это был класс, как класс компонента, из которого мы отправляем пользовательские события, можно было бы пометить его как статический:

```
static get PLAYLIST_UPDATE_EVENT() {return onPlaylistTimeUpdate '; },
```

Теперь, отправляя события или прослушивая их, мы можем избежать строки с опечатками. Слушатель может измениться на

```
let ce = new CustomEvent(WorkoutPlanData.PLAYLIST_UPDATE_EVENT, {
```

Добавить слушателей можно аналогичным способом:

```
EventBus.addEventListener(WorkoutPlanData.PLAYLIST_UPDATE_EVENT
```

Поскольку эти методы чтения являются статическими, экземпляру класса или компонента не нужно находиться нигде в поле зрения, чтобы иметь возможность использовать геттер. Эти типы событий доступны глобально. Теперь, если вы сделаете опечатку, появится ошибка с предупреждением. Более того, если вы используете интегрированную среду разработки, такую как VS Code или WebStorm, редактор кода автоматически предложит вам имя статического метода чтения, чтобы вы с самого начала не сделали ошибку.

### 14.6.2 Шаблоны проектирования как рекомендация

Не могу не подчеркнуть, что шаблоны проектирования, такие как шина событий, – всего лишь рекомендация. Если они помогут вашему приложению, отлично! Если нет, не используйте их. Всегда существует спор относительно существующих и старых шаблонов приложений и интерес к ним. И эти дебаты могут разгораться. Новые и популярные фреймворки имеют тенденцию усиливать определенные шаблоны. Некоторые разработчики могут работать с новыми для них способами и рассматривать их как единственный способ решения проблемы.

Подобно тому, как эти шаблоны не являются окончательным ответом на вопрос, шаблоны, представленные в этой главе, также не являются окончательным решением. Например, чрезмерное использование шины событий также может повредить, что может вызвать неразбериху в вашем приложении. Передача событий пользовательского интерфейса посредством глобальной шины, которые имеют отношение только к ва-

шему компоненту, может серьезно повлиять на его способность к совместному использованию и на то, насколько он может функционировать отдельно.

Существуют и ресурсы получше, в которых обсуждаются проектирование приложений и шаблоны. Наша цель здесь – просто показать, что веб-компоненты не ограничены по сравнению с другими современными фреймворками. Не все функции встроены в браузер, но существует бесчисленное множество JS-библиотек, которые могут помочь – если простого пользовательского решения, подобного тому, что изложено в этой главе, недостаточно.

## Резюме

Из этой главы вы узнали:

- как создавать пользовательские события и чем они отличаются от нативных событий при прохождении через DOM, в особенности через теневую модель DOM;
- как работать с синхронизацией веб-компонентов, прослушивая, когда они готовы или определены;
- как использовать статические методы чтения, чтобы избежать опечаток при работе с пользовательскими событиями;
- о некоторых шаблонах проектирования, с акцентом на работу с парадигмой MVC.

# 15

# Сокрытие сложностей

***Эта глава охватывает следующие темы:***

- использование фреймворка A-Frame для создания сцены с эффектом присутствия с поддержкой виртуальной реальности;
- использование компонента `model-viewer` от компании Google для предварительного просмотра 3D-моделей на экране и в дополненной реальности;
- управление трансляцией с камеры с помощью WebGL;
- использование фреймворка Babylon.js для создания компонента сцены в 3D-формате;
- отслеживание движения рук с помощью библиотек Tensorflow.js и handtrack.js.

Хотя эта книга подходит к концу, наше путешествие по веб-компонентам в действительности только начинается. Потребовалось некоторое время, чтобы создать основу для них, и еще больше времени, чтобы получить поддержку со стороны браузеров, но тем не менее последний из современных браузеров (Microsoft Edge) скоро будет обеспечивать их полную поддержку.

Проделанный путь был порой чреват препятствиями и тупиками. Мы были свидетелями того, как некоторые функции появляются и исчезают.

Среди устаревших – HTML-импорт, а также Polymer, первая библиотека веб-компонентов. Я уверен, что это не было совпадением, поскольку эта библиотека находилась под сильным влиянием HTML-импорта как отправной точки для каждого нового веб-компонента.

Было грустно видеть, что HTML-импорт уходит, но именно так и происходит при работе над веб-стандартом при наличии заинтересованных сторон от разных производителей браузеров. Как бы я ни любил JS-модули и шаблонные литералы для хранения HTML-кода и стилей, это не идеальное решение, которое подойдет всем. Для меня как разработчика это здорово, но не всем нравится HTML-код и стили внутри JS-кода.

## 15.1 Взгляд в будущее веб-компонентов

На протяжении долгих лет мы могли делать удивительные вещи в интернете, применяя лишь HTML-код и стили. Требование использовать JS как способ создания визуальных аспектов в компоненте – немного больное место. Вот почему я рад, что команда Chrome объявила о своем намерении выпустить модули HTML и CSS!

Я лично чувствую, что эти новые типы модулей станут одним из немногих больших шагов для веб-компонентов. Возможность создания небольшого фрагмента HTML- и CSS-кода за пределами вашего более крупного приложения и сложности вашего веб-компонента позволит значительно расширить возможности разработки веб-компонентов для тех, кто может быть не в курсе последних технологий JS. Это даст возможность лучше сосредоточиться на структуре и стиле компонентов, оставляя логику и интерактивность отдельной задачей.

Если вы похожи на меня и вам одинаково нравится HTML, CSS и JS, возможно, для вас это не имеет большого значения. Но позволить людям сосредоточиться на том, что у них хорошо получается, и использовать свои индивидуальные таланты для создания того же компонента в качестве команды будет удивительно!

Модули CSS могут быть еще важнее. Разрешение импорта небольших таблиц стилей в компоненты начинает учитывать то, что я считаю самым острым вопросом для компонентов с поддержкой Shadow DOM. Без стилей, способных пробить теневую модель DOM, годы рабочих процессов прошли впустую. Самым большим рабочим процессом CSS, которого нам не хватает, является концепция общей дизайн-системы, способной стилизовать ваш компонент или набор компонентов.

Дизайн-системы уже являются достаточно модульными в своем исходном коде. Только после сборки они становятся монолитным или полумонолитным CSS-файлом, предназначенным для стилизации всего приложения сверху донизу. Поскольку модули CSS входят в наши рабочие процессы, возможно, мы будем напрямую полагаться на крошечные модульные файлы с исходным кодом. Соедините это с (уже выпущенными в Chrome) *теневыми фрагментами* и *теневыми темами*, которые скоро появятся, и в ближайшем будущем у нас может быть чрезвычайно

надежное решение, которое создает еще лучший рабочий процесс для дизайн-систем в целом.

Я прогнозирую, что дизайн-системы и темизация приложений с использованием этих новых функций станут еще одним очередным переломным моментом для веб-компонентов. Хотя на самом деле это принесет пользу не только им. Учитывая, что большинство современных фреймворков в той или иной форме работают с компонентами (некоторые даже используют веб-компоненты ), эти функции могут быть одинаково актуальны для всех из них.

Это и правду фантастическая новость для всех. Мы уже наблюдаем тенденцию к использованию решений, не зависящих от фреймворка. Redux, MobX, lit-html и многие другие библиотеки решают небольшую целевую проблему. Хотя Redux популярна среди пользователей React, а lit-html – среди пользователей LitElement, эти решения можно использовать, где угодно. Забегая вперед, я вижу, что это будет продолжаться. Мы все можем использовать одни и те же решения для схожих проблем, независимо от того, на каком фреймворке (или его отсутствии) построен наш проект. Более того, сами веб-компоненты являются независимыми и могут использоваться в других фреймворках, как и любой иной элемент.

Если говорить о LitElement, то эта библиотека, похоже, завоевывает популярность. Совсем недавно появилась версия 1.0, готовая для промышленной эксплуатации… или, скорее, версия 2.0. И уже было заявлено, что пакет lit-element работает с NPM (www.npmjs.com/package/lit-element). Хотя разработчики из команды Polymer сумели «умыкнуть» название, оставшееся от предыдущего автора, они также хотели избежать путаницы, чтобы их версию 1.0 не путали с предыдущим проектом. Итак, LitElement была наконец выпущена как версия 2.0, пропуская версию 1.0. Не говоря о том, что Ionic уже некоторое время использует компилятор StencilJS. Stencil имеет собственную экосистему, но генерирует нативный веб-компонент без зависимостей.

Смотреть в будущее может быть захватывающим, но также важно понимать, что мы можем сделать прямо сейчас и как будущие изменения в веб-компонентах будут влиять на нас по мере продвижения вперед. Вот что является захватывающим. Похоже, что фундаментальные основы веб-компонентов в ближайшее время не изменятся. Все, что изложено в этой книге, должно оставаться актуальным на протяжении долгих лет.

Вещи, которые изменятся, – это не фундаментальные строительные блоки; вместо этого изменения будут приходить с рабочим процессом разработчика, когда речь идет о работе с макетом, стилями, шаблонами проектирования приложения и т. д. Тем не менее все эти детали не будут видны разработчикам, которые просто хотят использовать ваш компонент. Даже если вы будете использовать устаревшие методы 2019 года в своем компоненте в 2025 году, он все равно должен работать, потому что сложности внутри вашего компонента не очень важны для использования вне вашего компонента.

В феврале 2019 года я услышал статистику от команды Polymer, касающуюся использования веб-компонентов. Согласно этой статистике при

просмотре страниц в Chrome в 10 % из них так или иначе используются веб-компоненты. Эта цифра подчеркивает для меня самый большой посыл и то, почему я такой большой поклонник веб-компонентов.

Чтобы было понятно: 10 % – это слегка удивительный показатель, если честно. Это означает, что мы все, вероятно, используем веб-компоненты и даже не знаем об этом. Веб-компонент – это просто еще один элемент на странице. Они удивительно просты в использовании, но внутри компоненты могут делать невероятно сложные вещи!

Скрытая сложность веб-компонентов – это именно то, что так будоражит меня. Мы можем обернуть нечто потенциально безумно сложное и представить его пользователям в виде элемента с хорошо документированным API или несколькими атрибутами. Инкапсуляция, обеспечиваемая теневой моделью DOM, позволяет нам спать по ночам, зная, что, несмотря на все сложности, лежащие внутри, внешняя страница не принесет никаких сюрпризов.

Справедливости ради стоит сказать, что компоненты в любом современном фреймворке могут предлагать это. Когда появилась первая версия Angular, я написал несколько изящных «директив», по сути, компонентов того времени. Проблема в том, что когда Angular версии 1 вышел из моды, компоненты и директивы, которые у меня были, просто перестали быть актуальными, и мне пришлось переписывать их, если я хотел продолжать использовать тот же функционал.

Итак, понятие скрытых сложностей – тема, которой я бы хотел закончить эту книгу. Созданные нами проекты и компоненты были интересными, но я приложил все усилия, чтобы они были небольшими, чтобы мы могли обсудить большую часть, если не весь код, на страницах этой книги. Однако теперь я бы хотел раздвинуть эти искусственные границы и исследовать еще несколько тем!

## 15.2 3D и смешанная реальность

Я не уверен, что сегодня в интернете есть что-то более сложное, чем 3D и смешанная реальность! Держу пари, мы все знакомы с 3D. Используя 3D в реальном времени, мы можем смотреть на объект или сцену под любым углом. До недавнего времени большинству из нас удавалось взаимодействовать с 3D-сценой только с помощью традиционных модальностей пользовательского интерфейса, возможно, используя клавиши со стрелками для перемещения по игре или мышь для перетаскивания и вращения объекта, чтобы смотреть на него под разным углом.

Ситуация стала меняться в 2013 году с появлением виртуальной реальности. Я был одним из тех, кто участвовал в кампании, целью которой являлось финансирование проекта по созданию видеоочков Oculus Rift DK1 на сайте Kickstarter, и именно в 2013 году появились первые комплекты. Кроме того, примерно в это же время можно было приобрести небольшое недорогое картонное приспособление, которое удерживало ваш телефон прямо перед глазами, занимая всю область просмотра.

Кроме того, поскольку вы уже пристегнули телефон к лицу, можно было отслеживать относительное движение вашей головы. Когда вы смотрите вокруг в реальном мире, эту информацию можно отправить в 3D-сцену. Поэтому теперь, вместо того чтобы оглядываться, проводя мышью по плоскому интерфейсу, как мы это делали во времена шутеров от первого лица, в видеоиграх 90-х, ваш собственный взгляд и голова – это то, как вы смотрите вокруг.

Как такое возможно? Как телефон узнает о движениях головы? Сейчас акселерометр, гироскоп и магнитометр являются стандартными датчиками для любого телефона. Акселерометр определяет, насколько быстро ваш телефон движется в определенном направлении, а гироскоп может определить, насколько быстро ваш телефон вращается в определенном направлении. Магнитометр – это датчик, чаще всего известный как компас. Он может определять направление вашего телефона в пространстве с точки зрения сторон света: север, восток, запад или юг. *Алгоритмы сбора и обобщения данных от сенсоров* объединяют все их воедино и могут точно определить движения вашей головы! В сочетании с 3D-сценой с разделенным экраном, в которой левая точка обзора слегка смещается вправо, а затем наводится на глаза, вы получаете эффект трехмерного стереоизображения, словно на самом деле находитесь в этом виртуальном мире.

Вам, наверное, интересно, какое отношение это имеет к интернету и веб-компонентам. Все эти возможности какое-то время были частью сети в виде отдельных частей, даже без виртуальной реальности, и веб-компоненты определенно могут инкапсулировать и скрывать эти сложности.

Оставляя в стороне огромное количество математических вычислений и кода для слежения за движениями головы с использованием сбора и обобщения данных от датчиков, 3D – в целом очень сложная тема! Хотя трехмерная графика с использованием обычного кода на вашем процессоре возможна, она довольно медленная. Вот почему при любой современной и серьезной работе, связанной с 3D в реальном времени, используется графический процессор. В интернете единственный способ использовать графический процессор – это WebGL.

WebGL – это невероятно низкоуровневый код, и это не JavaScript. В приведенном ниже листинге показан пример шейдера WebGL, с помощью которого можно взять изображение и сделать радиальное затухание до черного цвета по центру.

**Листинг 15.1 Шейдер WebGL**

```
attribute vec2 a_position;
attribute vec2 a_texCoord;
uniform vec2 u_resolution;
varying vec2 v_texCoord;
uniform vec2 offset;

void main() {
```

Входящие и общие переменные, помогающие вычислить точки вершины

Основная функция для расчета и установки вершин на основе размера холста

```
    vec2 zeroToOne = a_position / u_resolution;
    vec2 zeroToTwo = zeroToOne * 2.0;
    vec2 clipSpace = zeroToTwo - 1.0 + offset;
    gl_Position = vec4(clipSpace.x * 1.0, clipSpace.y * -1.0, 0.0, 1.0);
    v_texCoord = a_texCoord;
}
```

Обратите внимание, что вышеупомянутый шейдер является так называемым «вершинным» шейдером. Вершины активно используются в трехмерной модели в качестве взаимосвязанных точек в пространстве. Для нас, поскольку у нас все очень просто и мы используем это только для манипулирования пикселями, наши точки в пространстве – просто квадрат с точным размером нашего холста.

Приведенный ниже фрагментный шейдер дополняет вершинный шейдер. Если раньше мы рисовали вершины и создавали нечто вроде плоского холста в контексте WebGL, то теперь мы можем настроить каждый цвет пикселя, который рисуем на этом холсте.

**Листинг 15.1 Шейдер WebGL (*продолжение*)**

```
precision mediump float;
varying vec2 v_texCoord;

uniform sampler2D u_image0;

void main(void) {
    vec4 sourcePixel =
    ➥texture2D(u_image0, v_texCoord);
    float multiply = 1.0;
    vec2 center = vec2(0.5, 0.5);

    float dist = distance(v_texCoord, center);
    gl_FragColor = (0.6-dist) * sourcePixel;
}
```

Координата пикселя, над которым мы работаем

Поступающая текстура (как на фотографии, которую мы предоставили шейдеру)

Получаем пиксель из текстуры в текущем местоположении пикселя

Устанавливаем темноту пикселя в зависимости от того, как далеко он находится от центра

Конечно, такой простой эффект очень далек от сцены, наполненной трехмерными моделями. Тем не менее предыдущий код является неплохим примером того, с чем вы должны работать для визуализации своей графики. Здесь задействовано много 3D-математики, и управляет всем этим HTML-элемент `<canvas>` и JavaScript. Учитывая всю эту сложность, обычно используется трехмерная библиотека более высокого уровня, например Three.js (https://threejs.org) или Babylon.js (www.babylonjs.com). По крайней мере, при наличии этих библиотек вам не придется обрабатывать рендеринг графики, писать шейдеры, которые намного сложнее, чем те, что приводятся в листинге 15.1. Вместо этого вы можете работать с такими виртуальными объектами, как сферы, кубы, и любыми трехмерными моделями, которые вы загружаете.

Но даже тогда перемещать вещи в трехмерном пространстве по-прежнему сложно! Мы имеем дело с матрицами преобразования, кватернионами и многим другим. Веб-компоненты могут помочь вам при работе с конкретными вещами, чтобы скрыть всю эту безумную сложность, в зависимости от вашего варианта использования.

### 15.2.1 A-Frame

Говоря технически, A-Frame не использует надлежащие веб-компоненты. Если посмотреть на его исходный код, можно понять, насколько действительно это утверждение с технической точки зрения. Однако я хочу рассказать о нем как о необычном, но чрезвычайно актуальном случае использования веб-компонента.

На сайте A-Frame (https://aframe.io) он описывается как «веб-фреймворк для создания виртуальной реальности». Здорово, но для меня это нечто большее. Я думаю, что мощь и привлекательность A-Frame заключаются в том, что он позволяет разработчикам и не разработчикам создавать 3D-сцены в интернете, которые также работают в виртуальной реальности.

Причина, по которой это так легко делать с помощью A-Frame, заключается в том, что на самом деле вы не программируете, когда приступаете к работе с ним. Эта библиотека позволяет создавать сцены с тегами на HTML-странице. Взять, к примеру, приведенный ниже листинг. Это простая 3D-сцена в стиле «Привет, мир», которая есть на сайте AFrame в качестве первого примера.

**Листинг 15.2 Сцена Hello WebVR**

```
<html>
<head>
   <title>Hello, WebVR! • A-Frame</title>
   <script src="aframe.min.js"></script>
</head>
<body>
<a-scene background="color: #ECECEC">
        <a-box position="-1 0.5 -3" rotation="0 45 0" color="#4CC3D9" shadow>
        </a-box>
        <a-sphere position="0 1.25 -5" radius="1.25" color="#EF2D5E" shadow>
        </a-sphere>
        <a-cylinder position="1 0.75 -3" radius="0.5" height="1.5" color=
        "#FFC65D" shadow></a-cylinder>
        <a-plane position="0 0 -4" rotation="-90 0 0" width="4" height="4"
        color="#7BC8A4" shadow></a-plane>
</a-scene>
</body>
</html>
```

Подключаем библиотеку A-Frame

Элемент, который содержит всю трехмерную сцену целиком

Пример 3D-объекта; a-box/куб

Этот небольшой фрагмент HTML-кода дает нам целую 3D-сцену! На рис. 15.1 показано все, что отображается в браузере. Кроме того, можно увидеть маленькую иконку в виде очков виртуальной реальности в правом нижнем углу.

При нажатии на этот значок вы переходите в режим погружения. На настольном компьютере это не так интересно – вы просто переходите в полноэкранный режим. На телефоне становится интереснее. Обычно при входе в режим погружения A-Frame разделяет экран, чтобы показывать немного разное содержимое слева и справа для стереоскопического

изображения. Также здесь вы столкнетесь с искажением – может показаться, что это чем-то напоминает очки Google Cardboard, где объективы увеличивают обзор, чтобы охватить все ваше поле зрения, когда телефон находится в нескольких миллиметрах от ваших глаз.

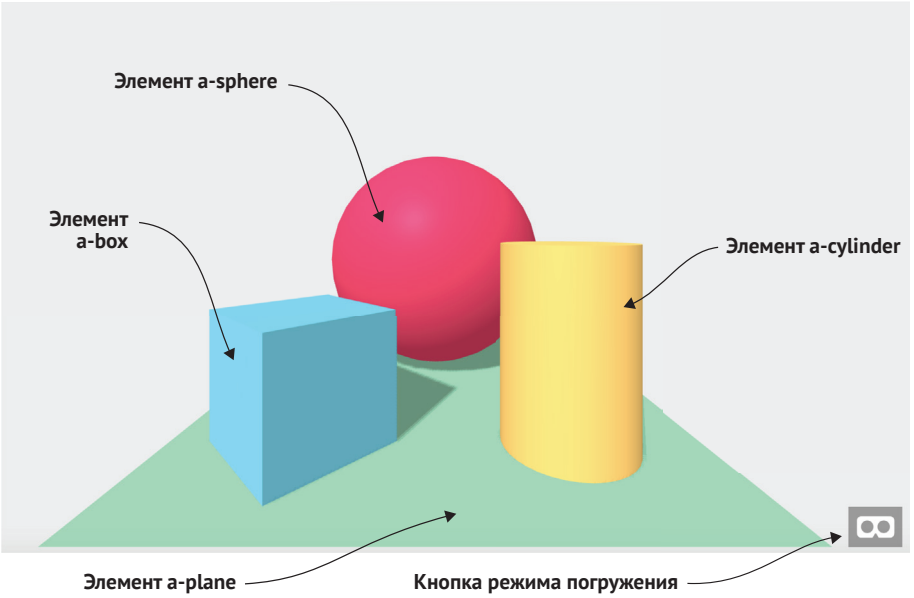



Рис. 15.1 Пример сцены в A-Frame

Если на вашем телефоне установлена платформа Daydream от компании Google, режим погружения становится еще интереснее, ведь теперь поддерживается Bluetooth-контроллер Daydream. Daydream – это платформа виртуальной реальности, разработанная Google. Она работает на смартфонах с сопутствующей гарнитурой и контроллером. Здесь по умолчанию при входе в режим погружения Daydream запускается автоматически.

Интернет-браузеры для настольных компьютеров также поддерживаются, равно как и новые браузеры для настольных компьютеров, ориентированные на виртуальную реальность, такие как Supermedium и Firefox Reality. Это немного сложнее, чем телефон, потому что теперь ваш браузер должен поддерживать ряд VR-гарнитур и контроллеров. Но все же режим погружения здесь работает аналогичным образом, используя реальную гарнитуру и контроллеры, такие как Oculus Rift, Oculus Go, HTC Vive и Vive Focus.

WebVR – стандарт, который уже можно считать устаревшим. По сути, WebVR определяет API, написанный на JavaScript, который реализуют браузеры, чтобы отображать виртуальную реальность и принимать данные о расположении и повороте, сообщая, где находятся ваши контроллеры и гарнитура. Учитывая новый ажиотаж вокруг дополненной реальности, следующая версия WebVR теперь называется WebXR. Она должна

включать в себя как можно больше разных способов погружения. Это напоминает ситуацию с веб-компонентами, которая была несколько лет назад. Некоторые компании-разработчики браузеров пошли дальше и попытались реализовать то, что, по их мнению, будет отличным стандартом. Эксперименты с WebVR, проведенные сообществом веб-разработчиков, доказали, что что-то работает, а что-то нет, и теперь мы вступаем в новый раунд стандартов с WebXR.

Какое отношение A-Frame имеет к веб-компонентам? Что же, давайте вернемся к разметке. Объявление 3D-сцены выполняется довольно просто с помощью того, что похоже на веб-компонент `a-scene`:

```
<a-scene background="color: #ECECEC"></a-scene>
```

При открытии инспектора DOM в инструментах разработки, как показано на рис. 15.2, видно, что этот компонент включает в себя элементы сцены, а также тег `<canvas>` для рендеринга трехмерной сцены. Интересно, что такие элементы, как `<a-box>`, которые обозначают куб или прямоугольник в трехмерной сцене, имеют нулевую высоту и ширину и нигде конкретно не расположены.

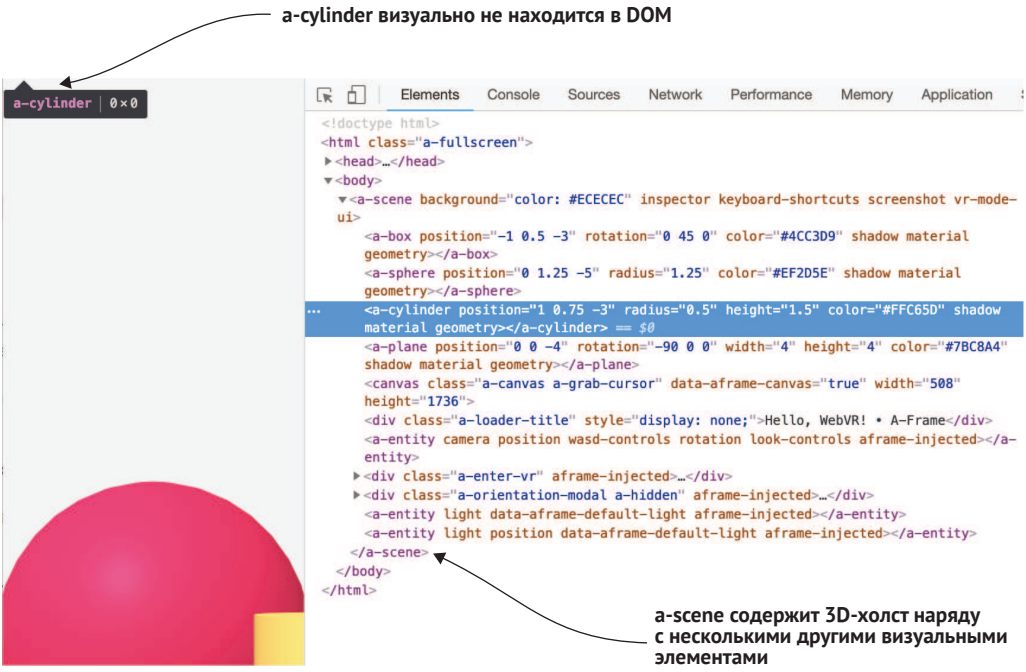



**Рис. 15.2 Изучение сцены из A-Frame**

Это объясняется тем, что эти элементы, обозначающие объекты в элементе `<a-scene>`, фактически не используются визуально. A-Frame использует HTML-элементы в качестве невизуальных моделей данных, которые будут созданы в 3D. Интересно, не правда ли? С одной сторо-

ны, у нас есть компонент <a-scene>, который обрабатывает невероятную сложность полноценной 3D-сцены, позволяя ей работать при различных настройках виртуальной реальности и оборудования.

С другой стороны, в <a-scene> у нас есть множество невизуальных компонентов, которые ничего не делают, кроме как помогают создать 3D-сцену. Я думаю, что понятие невизуальных компонентов суперинтересно. У меня есть сомнения относительно того, полезны они или нет. Будучи хорошим JavaScript-разработчиком, я думаю, что все, что не является визуальным, должно делаться с использованием JS, оставляя HTML-код снаружи. Тем не менее существует хорошая доступность к невизуальным веб-компонентам. Любой человек, не обладающий знаниями JavaScript, может просто разместить на странице что-нибудь невизуальное, например фоновый аудиоплеер, не заботясь об инстанцировании, JS-библиотеках или каких-либо других проблемах.

Здесь, поскольку эти невизуальные компоненты и визуальный тег <a-scene> помогают друг другу создавать всю сцену, чтобы все это выглядело согласованно в вашей HTML-разметке, я определенно приветствую это! Просто интересно начать редактировать сцену вживую с помощью инструментов разработчика и наблюдать, как 3D-сцена мгновенно меняется, как показано на рис. 15.3, где я меняю цвет куба и вращение.

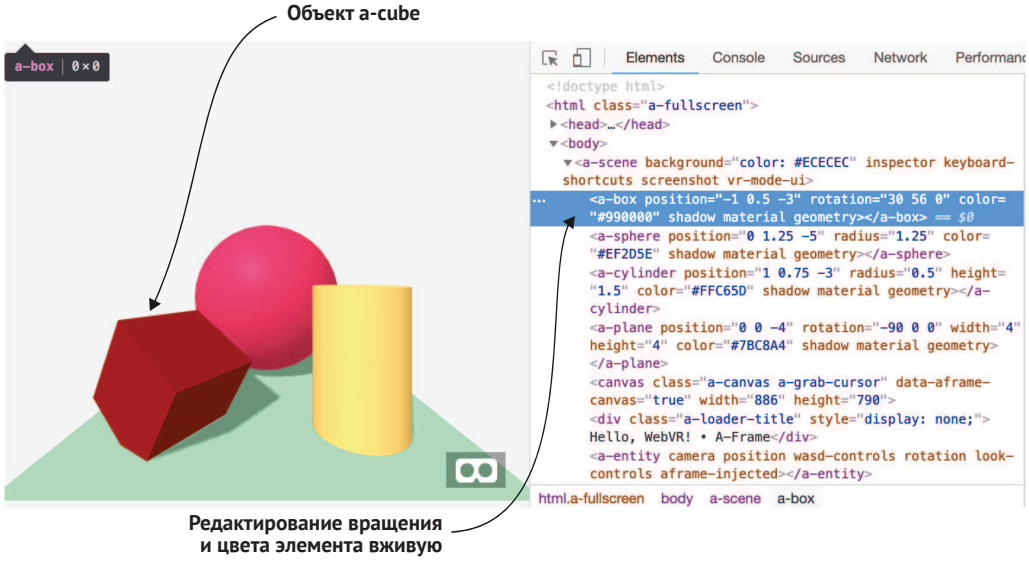



**Рис. 15.3 Изменение объектов A-Frame вживую в инструментах разработчика браузера**

Таким образом, у нас есть создание пользовательских элементов, функции attributeChangedCallback, connectedCallback и многое другое. В самом начале я утверждал, что это не веб-компонент с технической точки зрения. Причина проста. На этом этапе A-Frame не использует API пользовательских элементов. Он использует старый API document.register- Element с полифилом, чтобы он работал везде. Один из ведущих ав-

торов A-Frame заявил, что скоро они начнут использовать API пользовательских элементов (https://github.com/aframevr/aframe/issues/3923), но не всегда и везде. Я все же считаю, что A-Frame – это отличный вариант применения для веб-компонентов. Кроме того, тут нет теневой модели DOM. Здесь действительно нет причин для этого, поскольку у элементов нет стилей, и предпочтительнее разрешить неограниченный доступ к внутренней модели DOM внутри тега `<a-scene>` для манипулирования сценой, какой хочет разработчик. В результате нам не нужно управлять невизуальными внутренними дочерними узлами никак иначе, кроме как обычными элементами. При использовании теневой модели DOM эти дочерние узлы должны управляться как слоты.

## 15.2.2 Компонент model-viewer

Одним из специфических, но популярных вариантов использования 3D в интернете является возможность предварительного просмотра 3D-модели, позволяющая пользователю перетаскивать ее, вращать и увеличивать. Google создала веб-компонент специально для этого под названием `model-viewer`. Документацию и примеры можно посмотреть по адресу https://googlewebcomponents.github.io/model-viewer, но я думаю, что стоит немного поэкспериментировать с ним.

В репозитории для данной книги на сайте GitHub я сделал именно это. В файле simpledemo.html можно увидеть этот компонент в действии. Мы будем наращивать интерактивность и добавим цвет фона, как в демонстрации, по мере продвижения. На самом деле здесь не нужно делать ничего особенного. Мы просто разместим ссылку на JS-файл компонента, добавим немного стилей для установки размера компонента и, наконец, разместим компонент на странице, как показано в приведенном ниже листинге.

**Листинг 15.3 Демонстрация работы компонента `model-viewer`**

```
<html>
<head>
   <script src="model-viewer.js"></script>
   <title>Simple Demo for Model Viewer</title>
   <style>
       body {
           margin: 0;
       }
       model-viewer {
           width: 100vw;
           height: 100vh;
       }
   </style>
</head>
<body>
     <model-viewer src="Astronaut.glb"></model-viewer>
</body>
</html>
```

JS-файл компонента model-viewer

Устанавливаем размеры компонента, чтобы он занимал всю страницу

Добавляем компонент на страницу

Самостоятельно вы здесь мало что сделаете, но для вашего удобства я загрузил 3D-модель астронавта и JS-код компонента в репозиторий для этой книги на сайте GitHub, чтобы вы могли продолжить работу, не теряя времени. 3D-модель – совершенно новый 3D-формат под названием glTF. Сжатый как бинарный пакет, ее формат файла имеет расширение .glb. Это еще одна сложность, поскольку 3D-форматы нужно распаковать и проанализировать для создания 3D-модели в 3D-движке.

После запуска и начала работы он не выглядит так впечатляюще без взаимодействия! Это также может быть и изображение. Компонент `model-viewer` дает нам целый набор атрибутов для работы. Вероятно, наименее впечатляет возможность добавить цвет фона, показанный на рис. 15.4. Давайте начнем с лавандового фона:

```
<model-viewer src="Astronaut.glb" background-color="#9999bb"></model-viewer>
```

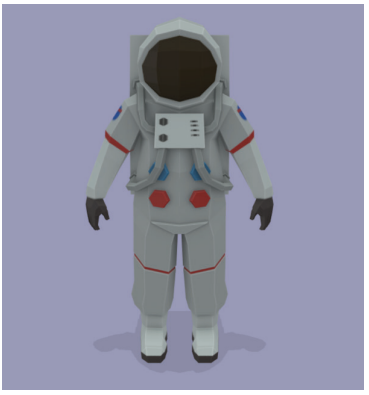


**Рис. 15.4 Модель космонавта на цветном фоне**

Далее сделаем этот 3D-контекст полезным. Компонент `model-viewer` позволяет выполнять автоповорот, как если бы астронавт стоял на медленно вращающемся диске:

```
<model-viewer src="Astronaut.glb"
              auto-rotate
              background-color="#9999bb"></model-viewer>
```

Или, возможно, вы хотите, чтобы он стал еще немного более интерактивным, позволяя ему вращаться, перетаскивая:

```
<model-viewer src="Astronaut.glb"
              controls
              background-color="#9999bb"></model-viewer>
```

Обратите внимание на детализацию, когда вы перетаскиваете его, чтобы он вращался. При перетаскивании происходит небольшое ускорение, и оно довольно быстро пропадает после отпускания, но поскольку в конце оно ослабевает, то не кажется слишком резким.

Примите во внимание эту крошечную деталь юзабилити, когда будете думать о других мелочах, которую делает этот компонент, от рендеринга геометрии до использования холста WebGL и загрузки трехмерной моде-

ли с использованием геометрии, материалов и текстур. Создание такого компонента требует времени, поэтому здорово, что Google уже создала его и поделилась с нами его открытым исходным кодом.

В конечном итоге `model-viewer` – это просто еще один компонент, который мы можем включить в нашу страницу, не вникая во все сложности, что дает нам возможность обратить внимание на другие аспекты нашего приложения.

### 15.2.3 model-viewer и поиск с помощью Poly

Помните, в главе 3 мы создавали поиск 3D-моделей с помощью площадки 3D-объектов Poly от компании Google? Тогда было лишним детально освещать такие темы, как демонстрация 3D-модели и возможность взаимодействия с ней, да и сейчас тоже. Но нам не нужно вдаваться в эти подробности; мы можем просто сделать нашу функцию поиска и найти реальный результат, предварительно просмотренный целиком в 3D с помощью компонента `model-viewer`.

В приведенном ниже листинге показан пример поиска, дополненный URL-адресом glTF в качестве атрибута каждой миниатюры изображения. Мы можем прослушивать события клика на каждой миниатюре, брать этот URL-адрес и обновлять компонент `model-viewer`.

**Листинг 15.4 Компонент для поиска на сайте Poly и компонент `model-viewer` для предварительного просмотра**

```
<html>
<head>
   <title>Poly Search with Preview</title>
   <script src="model-viewer.js"></script>
   <script src="poly-search.js" type="module"></script>

   <style>
       model-viewer {
           width: 50vw;
           height: 50vh;
       }
   </style>
</head>
<body>
<model-viewer src="../Astronaut.glb" controls></model-viewer>

<label>Enter search term: </label>
<input type="text" onchange="updatePolySearch(event)" />
<br /><br />

<poly-search apikey="<enter your API key here>"
             format="GLTF2"
             thumbheight="50"
             backgroundcolor="#99ffff"
             baseuri="https://poly.googleapis.com/v1/assets"
             searchterm="parrot">
</poly-search>
```

Импортируем компонент poly-search и подключаем файл компонента model-viewer

Даем указание poly-search, чтобы он включал только результаты в формате glTF (для совместимости с компонентом model-viewer)

Не забудьте ввести собственный API-ключ из главы 3, чтобы этот пример работал

```
<script>
   function updatePolySearch(event) {
       document.querySelector('poly-search').searchTerm = event.target.value;
   }

   document.querySelector('poly-search').addEventListener('click', e => {
       const model =
               e.target.getAttribute('gltf');
       document.querySelector('model-viewer').setAttribute('src', model);
   });
</script>
```

При щелчке мышью получаем URL-адрес glTF и обновляем компонент model-viewer

```
</body>
</html>
```

Для краткости я обернул компонент poly-search как его собственный модуль и использовал шаблонный литерал внутри для управления его стилями. Раньше мы этого не делали, потому что глава 3 идет до введения этих концепций. Не стесняйтесь и попробуйте сделать это самостоятельно или посетите репозиторий для данной книги на сайте GitHub.

Единственное изменение в логике – фильтрование этих результатов для включения только файлов в формате glTF и получение URL-адреса результата. Это изменение показано в приведенном ниже листинге.

**Листинг 15.5 Фильтрация результатов по glTF, включая URL-адрес элемента изображения результата**

```
for (let c = 0; c < assets.length; c++) {
   for (let d = 0; d < assets[c].formats.length; d++) {
       if (assets[c].formats[d].formatType ===
                     this.getAttribute('format')) {
           html += '<img gltf="' +
             assets[c].formats[d].root.url +
             '" src="' +
             assets[c].thumbnail.url +
             '" width="' +
             this._thumbwidth + '" height="' + this._thumbheight + '"/>';
       }
   }
}
```

Выполняем фильтрацию по формату (glTF, как указано в атрибуте компонента)

Добавляем специальный атрибут к результату изображения для URL-адреса glTF

В получившейся демоверсии мы изначально загружаем модели попугаев, но, скорее всего, мы найдем 3D-модель для всего, что вводим в это окно поиска. Конечно, при нажатии на каждый результат вы получаете полный интерактивный предварительный просмотр. На рис. 15.5 я ввел слово «spaceship», а в компоненте poly-search из главы 3 показывает соответствующие результаты. При нажатии на любой результат передает событие в окружающую HTML-страницу, которая устанавливает для атрибута `src` значение в виде URL-адреса в качестве результата.

Подумайте о том, что мы только что сделали! Мы взяли простой компонент, прежде чем достаточно узнали о чем-то, связанном с веб-компонентами, объединили его со сложным веб-компонентом Google – на

самом деле мы и понятия не имеем, как он работает, – и сделали нечто суперполезное очень простым способом. Сокрытие сложности внутри нескольких простых тегов и создание нечто большего, чем сумма его фрагментов, – моя любимая часть веб-компонентов.

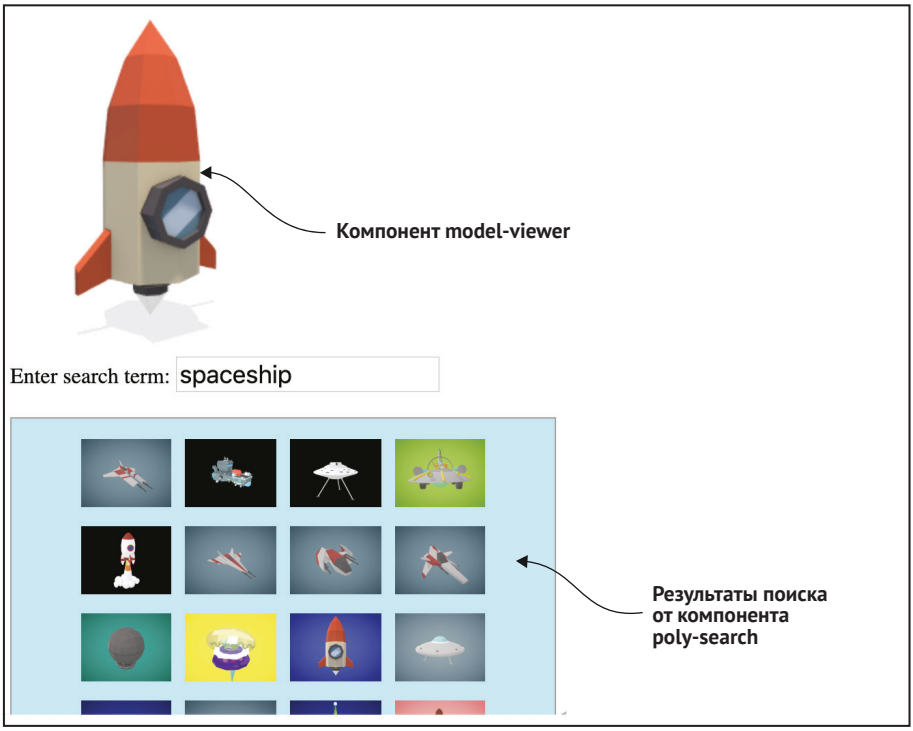



Рис. 15.5 Компонент poly-search с компонентом model-viewer

## 15.2.4 Дополненная реальность и model-viewer

Несмотря на сложность 3D, можно пойти еще дальше. Дополненная реальность – это следующий захватывающий этап после реальности виртуальной. В то время как виртуальная реальность позволяет вам видеть полностью искусственный и виртуальный мир, дополненная реальность позволяет размещать виртуальные объекты в реальном мире. Это чрезвычайно сложно с точки зрения аппаратного обеспечения.

Экраны, сквозь которые вы не видите, повсюду вокруг нас. С другой стороны, создание экрана, через который пользователь может видеть, пока он монтируется у него на голове, – большая проблема, с которой сейчас пытаются справиться некоторые крупные, хорошо финансируемые компании.

Примечательно, что самые большие усилия по созданию подобного оборудования предпринимаются Magic Leap и Microsoft HoloLens. Их устройства стоят тысячи долларов и, честно говоря, не соответствуют тому, что вы можете себе представить, ввиду ограниченного поля зре-

ния. Под этим я подразумеваю, что когда вы испытываете одно из этих устройств, виртуальные объекты в вашей сцене ограничены областью вашего обзора, аналогичной тому, как если бы вы держали лист бумаги размером 8,5×11 на расстоянии вытянутой руки, как показано на рис. 15.6. Если смотреть на эти виртуальные объекты на расстоянии, просто удивительно, что можно увидеть и как эти объекты существуют в реальном мире. Тем не менее когда вы приближаетесь и объекты больше этого ограниченного поля зрения, они обрезаются!

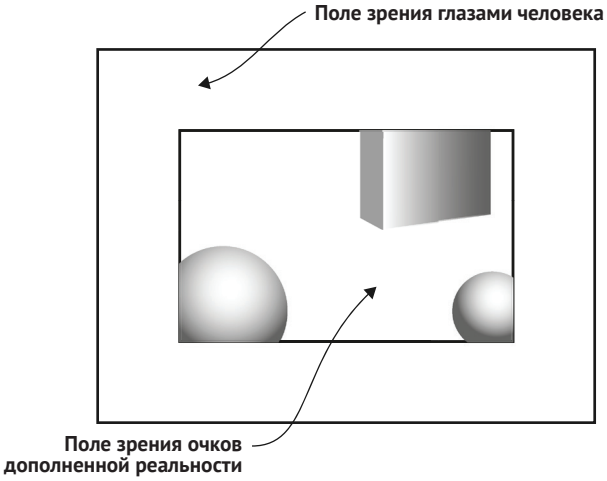



**Рис. 15.6 Пример поля зрения в очках дополненной реальности**

Эта и аналогичные проблемы наряду с ценой являются причиной, почему люди отказываются от таких устройств и активно используют дополненную реальность на своих телефонах, пока мы ждем гарнитуру. Отдельно от этого (или, возможно, в сочетании с футуристическими очками в своих секретных аппаратных лабораториях) компании Apple и Google работают над библиотеками для своих смартфонов – ARKit и ARCore, – чтобы дать разработчикам возможность создавать интерфейсы с дополненной реальностью в смартфонах.

Сложность, которую решают эти библиотеки, заключается в попытке «увидеть» мир.

Используя компьютерное зрение, библиотеки ARKit и ARCore находят интересные «функции» в реальном мире с помощью камеры вашего телефона. Эти функции проявляются как трехмерные точки, которые они находят. Когда эти трехмерные точки найдены, их можно связать вместе, и они проявляются как найденная поверхность, например пол, стол или стена. Когда поверхность найдена, на нее можно поместить 3D-объект или сцену, как показано на рис. 15.7, благодаря ARCore Quickstart от компании Google (https://developers.google.com/ar/develop/java/quickstart).

Эти скрытые сложности продолжают накапливаться! Компонент `model-viewer` также пытается взяться за это. К сожалению, на момент написания

данной главы чистая дополненная реальность на базе веб-интерфейса не поддерживается в вашем телефоне. Это связано с тем, что для этого требовалась экспериментальная версия Chrome (Chrome Canary) с WebVR. Сейчас, когда WebXR находится в стадии разработки, Chrome Canary больше не поддерживает функции, которые использовал `model-viewer`, чтобы предоставить нам дополненную реальность.

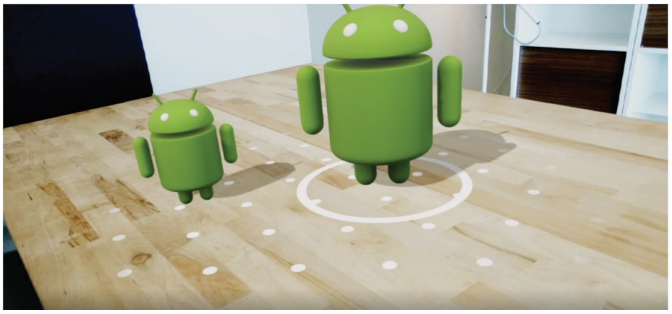


**Рис. 15.7 Размещение виртуального объекта в реальном мире с использованием ARCore**

Вы все равно можете попробовать дополненную реальность с компонентом `model-viewer`, если у вас есть устройство Magic Leap или более новый iPhone с iOS версии 12 и выше. Если у вас есть подобная возможность, то попробовать очень легко (хотя, признаюсь, я не пробовал Magic Leap).

В случае с Magic Leap это так же просто, как настроить соответствующий атрибут `<modelviewer src = "Astronaut.glb" magic-leap>` и включить библиотеку @magicleap/prismatic в свою страницу. Поскольку у меня нет устройства Magic Leap, и у вас, вероятно, тоже, вариант с iPhone является наиболее доступным для нас.

Напомню, что я сказал, что чистая дополненная реальность на базе веб-интерфейса в настоящее время не поддерживается `model-viewer`, потому что в своем текущем виде этот компонент немного жульничает, когда речь идет о iOS. Для активации дополненной реальности он использует функцию быстрого предварительного просмотра Quick Look, разработанную Apple, которая теперь поддерживает 3D и дополненную реальность.

При входе в режим погружения с веб-страницы открывается приложение Quick Look с вашей 3D-моделью. Несмотря на разные ухищрения, через которые прошел компонент, его очень легко опробовать, если у вас более новый iPhone. В приведенном ниже листинге показано простое изменение нашего последнего примера.

**Листинг 15.6 Дополненная реальность с компонентом `model-viewer`**

```
<model-viewer src="Astronaut.glb"
              background-color="#45aa22"
              ios-src="Astronaut.usdz">
</model-viewer>
```

Новый атрибут ios-src с файлом с расширением USDZ для активации дополненной реальности на телефоне с iOS версии 12 и выше

Для поддержки Quick Look модель должна быть представлена в новом 3D-формате Apple, USDZ. Я скачал и предоставил эту модель в репозитории для этой книги на сайте GitHub, чтобы вы могли легко посмотреть ее.

### 15.2.5 Ваш собственный 3D-компонент

Конечно, бывают случаи, когда вы отлично знаете, как работает сложный компонент, потому что сами его писали. Скрыть эту сложность от остальной части приложения, чтобы вы могли беспокоиться о разработке одного компонента за раз, может быть полезно. Разработка для 3D – совсем иной контекст, и она определенно может доставить вам неприятностей, когда остальная часть вашего приложения представляет собой 2D-интерфейс.

Оборачивание того, что вы делаете, в компонент, содержащий тег `<canvas>` для отображения и весь код JavaScript, необходимый для запуска полной 3D-сцены с циклом рендеринга, может быть отличным способом. Смешение 3D и 2D напоминает прекрасное использование палитры цветов, которую мы создали в прошлых главах!

Эта демонстрация, в частности, будет иметь трехмерную сцену, которая содержит простой примитив, такой как сфера, куб или низкополигональная сфера. Эта сцена будет включать в себя настраиваемую камеру и источники света, а также функцию перетаскивания, позволяющую пользователю вращать камеру, чтобы смотреть на сцену под любым углом.

На стороне двумерного пользовательского интерфейса палитра цветов позволит изменять цвет и прозрачность 3D-примитива. Кроме того, у нас есть несколько простых кнопок, позволяющих пользователю выбрать любой 3D-примитив, который он хочет просмотреть. На рис. 15.8 показано, что мы получим в итоге.

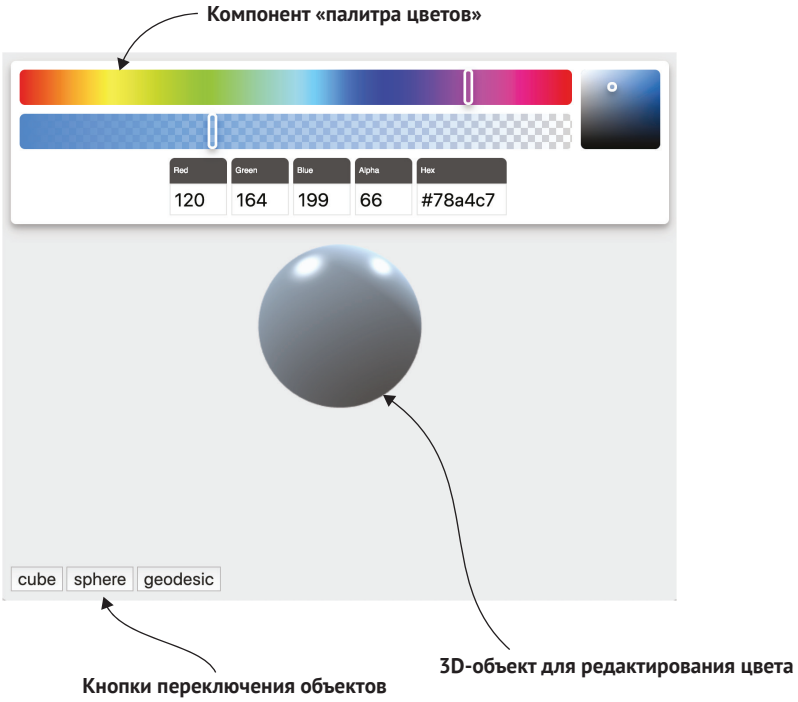



Рис. 15.8 Палитра цветов с использованием 3D

Всего будет три компонента, включая палитру цветов, которую мы уже сделали. Два других – это компонент 3D-сцены и компонент приложения, который содержит и 3D-сцену, и палитру цветов. На рис. 15.9 показана структура папок всего приложения.

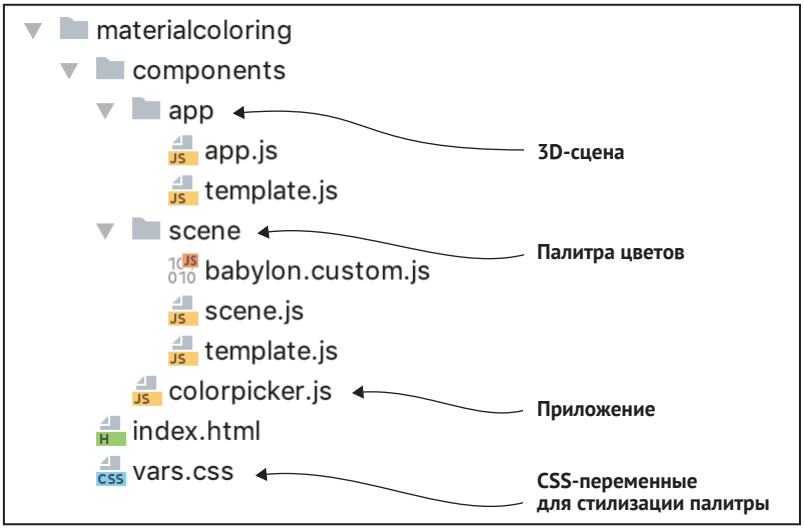



Рис. 15.9 Структура палитры

Есть несколько вещей, которые могут показаться необычными. В этой книге мы использовали два файла для каждого компонента – один для класса компонента и один для хранения HTML-кода и стилей. Это по-прежнему так. У компонента `scene` есть дополнительный файл, содержащий библиотеку Babylon.js.

Как я упоминал ранее в этой главе, 3D сложно сделать, а WebGL слишком низкоуровневый, чтобы мы могли с ним работать. Именно поэтому 3D-библиотека – довольно стандартная плата при работе над подобными вещами. Three.js, вероятно, является популярной 3D-библиотекой в настоящее время и фактически используется как в компоненте `model-viewer`, так и в A-Frame.

Это отличная библиотека, но в последнее время я предпочитаю Babylon.js. Впрочем, это дело вкуса. Мне нравится, что представляет собой полный пакет Babylon.js, тогда как Three.js больше основана на плагинах, если вам нужно что-то, помимо простого функционала.

В этом нет абсолютно ничего плохого – всему свое время и место, и я бы сказал, что обе эти библиотеки одинаково потрясающие.

Помимо дополнительной библиотеки, папки компонента палитры цветов не существует! Я просто скопировал файл сборки компонента, который мы создали в главе 12 с помощью Rollup.

Единственное, что вы, возможно, забыли, – это CSS-файл в корне проекта, содержащий CSS-переменные, которые помогли стилизовать палитру цветов.

Начнем с простого. Вначале в приведенном ниже листинге показана базовая HTML-страница с приложением.

**Листинг 15.7 Файл index.html трехмерной палитры цветов**

```
<head>
  <title>Material Coloring</title>
  <script                                    Подключаем основное приложение
    type="module"
    src="components/app/app.js">
  </script>
  <script                                    Подключаем 3D-библиотеку Babylon.js
    src="components/scene/babylon.custom.js">
  </script>
  <script                                    Подключаем палитру цветов
    src="components/colorpicker.js">
  </script>
  <link                                      Подключаем CSS-переменные для темизации палитры цветов
    rel="stylesheet"
    type="text/css"
    href="vars.css"/>

  <style>
      body {
          margin: 0;
          padding: 0;
          overflow: hidden;
```

```
        }

        mc-app {  ← Делаем так, чтобы приложение занимало всю страницу
            width: 100vw;
            height: 100vh;
        }
    </style>
</head>
<body>
    <mc-app></mc-app>  ← Размещаем приложение на странице
</body>
</html>
```

Здесь есть две ссылки на скрипты, которые немного выделяются. Во-первых, мы подключили библиотеку Babylon.js здесь, а не в компоненте, где она используется. Я бы предпочел импортировать ее в качестве модуля, но брать весь исходный код Babylon и работать с ним – для нашего простого примера – это чересчур. С другой стороны, было бы неплохо подключить теги `<script>` при настройке свойства компонента 3D-сцены `innerHTML`. К сожалению, из-за проблем, связанных с безопасностью, скрипты нельзя загружать подобным образом. Вместо этого нам пришлось бы создать новый элемент `<script>`, установить исходник и добавить его вручную. Вариант в листинге 15.7 выглядит проще.

Во-вторых, вы заметите, что палитра цветов не импортируется как модуль. Повторюсь, это сделано для удобства. Вместо того чтобы копировать весь исходный код компонента или импортировать исходник, где нам пришлось бы иметь дело с несколькими уровнями, и снова проходить целую главу в этом репозитории, проще скопировать файл сборки, который мы упаковали.

Переходя к компоненту приложения в components/app, мы начнем с файла template.js, чтобы просмотреть HTML-код и стили. Эти подробности показаны в приведенном ниже листинге.

**Листинг 15.8 Файл template.js**

```
import Scene from '../scene/scene.js';

export default {
    render() {
        return `${this.css()}
                ${this.html()}`;
    },

    mapDOM(scope) {
        return {
            scene:  ← Кешируем ссылки на компонент scene и палитру цветов
                scope.querySelector('mc-scene'),
            colorpicker: scope.querySelector('wcia-color-picker')
        }
    },

    html() {
        return `<mc-scene  ← Компонент 3D-сцены с объектом-примитивом по умолчанию – кубом
```

```
                    object="cube">
                </mc-scene>
                <div id="model-buttons">
                    <button class="object-button">cube</button>
                    <button class="object-button">sphere</button>
                    <button class="object-button">geodesic</button>
                </div>
                <wcia-color-picker class="modal" hex="#99224A">
                </wcia-color-picker>`;
    },
    css() {
        return `<style>
                . . . CSS here
                </style>`;
    }
}
```

Кнопки для клика и изменения примитива, который показывает 3D-сцена

Палитра цветов с уже установленным цветом по умолчанию

Стили довольно просты, однако это немного отличается от нашего обычного макета, поскольку мы накладываем все элементы поверх сцены посредством абсолютного позиционирования, как показано ниже.

**Листинг 15.9 Абсолютное позиционирование элементов над 3D-сценой**

```
:host {
    display: inline-block;
}

#model-buttons {
    position: absolute;
    width: 100%;
    bottom: 10px;
    left: 10px;
}

#model-buttons button {
    font-size: 20px;
}

mc-scene {
    position: absolute;
    width: 100%;
}

wcia-color-picker {
    position: absolute;
    width: calc(100% - 20px);
    margin: 10px;
}
```

Кнопки модели находятся внизу и над 3D-сценой

3D-сцена занимает весь компонент, но находится под всем остальным

Палитра цветов находится в верхней части страницы, над 3D-сценой и с отступами со всех сторон

Далее, поскольку мы связываем только два этих компонента (палитру цветов и 3D-сцену), код JavaScript в файле component/app/app.js также очень прост. Он ничем не отличается от любого другого компонента, который мы сделали, и в приведенном ниже листинге показаны фрагменты, которые не являются стандартной установкой веб-компонента.

**Листинг 15.10 Код JavaScript компонента приложения**

```
import Template from './template.js';

export default class App extends HTMLElement {
   constructor() {
       super();
       this.attachShadow({mode: 'open'});
       this.shadowRoot.innerHTML = Template.render();
       this.dom = Template.mapDOM(this.shadowRoot);
       const observer = new MutationObserver( e => this.onMutationChange(e));
       observer.observe(this.dom.colorpicker, { attributes: true });
       this.shadowRoot.addEventListener('click', e => this.onClick(e));
       this.dom.scene.color =
               this.dom.colorpicker.hex;
       this.dom.scene.alpha = this.dom.colorpicker.alpha;
   }

   onClick(e) {
       if (e.target.classList.contains(
               'object-button')) {
           this.dom.scene.object = e.target.innerText;
       }
   }

   onMutationChange(changes) {
       for (let c = 0; c < changes.length; c++) {
           switch (changes[c].attributeName) {
               case 'hex':
                   this.dom.scene.color = this.dom.colorpicker.hex;
                   break;

               case 'alpha':
                   this.dom.scene.alpha = this.dom.colorpicker.alpha;
                   break;
           }
       }
   }
}

if (!customElements.get('mc-app')) {
   customElements.define('mc-app', App);
}
```

Прослушиваем изменения атрибутов

Прослушиваем события клика, чтобы перехватить их при нажатии на кнопки 3D-объекта

Изначально устанавливаем цвет и альфа-канал 3D-сцены на основе значений по умолчанию для палитры цветов

После того как все события клика перехвачены из компонента, выполняем фильтрацию по классу, слушая только события кнопок 3D-объекта

Устанавливаем цвет или альфа-канал в зависимости от изменения

Необходимость снова использовать `MutationObserver` для такой простой задачи причиняет неудобства. Это слишком сложно, поэтому я хотел бы, чтобы в палитре цветов было встроенное пользовательское событие, как мы рассматривали в предыдущей главе. Это может быть отличным домашним заданием для вас. Вы можете вернуться назад и сделать все самостоятельно, а также выполнить рефакторинг предыдущего кода, чтобы использовать его.

Двигаемся дальше. Теперь мы переходим к компоненту 3D-сцены. Учитывая, насколько мало здесь HTML-кода и стилей, потому что мы

просто используем элемент <canvas> для отображения 3D, сначала мы покажем файл components/scene/template.js.

**Листинг 15.11 HTML-код и стили 3D-сцены**

```
export default {
   render() {
       return `${this.css()}
               ${this.html()}`;
   },

   mapDOM(scope) {
       return {
           scene: scope.querySelector('canvas')
       }
   },

   html() {
       return `<canvas touch-action="none">
                 </canvas>`;
   },

   css() {
       return `<style>
                 :host {
                   display: inline-block;
                   width: 100%;
                   height: 100%;
                 }
                 canvas {
                   width: 100%;
                   height: 100%;
                 }
               </style>`;
   }
}
```

Кешируем элемент canvas для использования в классе компонента

Единственным элементом здесь является canvas, а touch-action = «none» активирует взаимодействие с мышью в Babylon

CSS здесь служит только для установки размера компонента, а canvas – для заполнения страницы

Как я уже говорил, это действительно просто. Здесь нам только нужно показать тег <canvas>. С точки зрения написания кода для 3D в целом, в классе компонентов из файла components/scene/scene.js мы не делаем ничего слишком сложного. Тем не менее нужно еще разобраться с настройками для сцены, освещения и камеры. Разделив JS-модуль, давайте сначала рассмотрим фрагменты стандартного веб-компонента в приведенном ниже листинге.

**Листинг 15.12 Настройка веб-компонента для 3D-сцены**

```
import Template from './template.js';

export default class Scene extends HTMLElement {
   static get observedAttributes() {
       return ['object', 'color', 'alpha'];
   }
```

Атрибуты, за которыми мы наблюдаем (3D-объект, цвет и альфа-канал)

```
    set color(val) {                          JS-методы для поддержки рефлексии компонентов
        this.setAttribute('color', val);
    }
    get color() { return this.getAttribute('color'); }
    set alpha(val) { this.setAttribute('alpha', val); }
    get alpha() { return parseFloat(this.getAttribute('alpha')); }
    set object(val) { this.setAttribute('object', val); }
    get object() { return this.getAttribute('object'); }

    attributeChangedCallback(                 Функция attributeChangedCallback обрабатывает
      name, oldVal, newValue) {               изменения цвета и альфа-канала и трехмерного
        switch (name) {                       объекта с помощью методов, которые еще не показаны
            case 'alpha':
                this.updateColor();
                break;

            case 'color':
                this.updateColor();
                break;

            case 'object':
                this.switchMesh(newValue);
                break;
        }
    }

    constructor() {
        super();
        this.attachShadow({mode: 'open'});
        this.shadowRoot.innerHTML = Template.render();
        this.dom = Template.mapDOM(this.shadowRoot);
        this.initScene();                     Инициализируем еще
    }                                         не показанную 3D-сцену
}

if (!customElements.get('mc-scene')) {
    customElements.define('mc-scene', Scene);
}
```

И снова видно, что у нас есть много места, съеденного рефлексией. Я сжал код немного больше, чем обычно, потому что он занимает много места. Как я уже говорил в предыдущей главе, этот скучный, повторяющийся код – именно то, что решит хорошая утилита или библиотека!

Теперь в приведенном ниже листинге мы рассмотрим JS-код для настройки 3D-сцены. Это просто дополнительные функции в том же классе.

**Листинг 15.13 Код настройки 3D-сцены**

```
initScene() {
    this.engine = new BABYLON.Engine(         Настройка движка Babylon.js и сцены
      this.dom.scene, true);
    this.scene = new BABYLON.Scene(this.engine);
    this.scene.clearColor = new BABYLON.Color3(0.894, 0.894, 0.894);

    const camera = new BABYLON.ArcRotateCamera(   Настройка камеры и освещения
```

```
      "Camera",
      Math.PI / 2,
      Math.PI / 2, 4,
      BABYLON.Vector3.Zero(), this.scene);
    const light1 = new BABYLON.HemisphericLight("light1",
      new BABYLON.Vector3(1, 1, 0), this.scene);
    const light2 = new BABYLON.PointLight("light2",
      new BABYLON.Vector3(0, 1, -1), this.scene);
    camera.attachControl(this.dom.scene, true);

    this.engine.runRenderLoop(
      () => this.render() );
    window.addEventListener(
      'resize', () => this.onResize());
}

render() {
    this.scene.render();
}

onResize() {
    this.engine.resize();
}
```

Прикрепляем элементы управления взаимодействием к камере для перетаскивания и поворота

Babylon нужен цикл рендеринга для постоянного повторного рендеринга и обновления сцены, когда происходят изменения

Изменяем размер 3D-сцены при изменении размера всей страницы

Вероятно, что вы никак не ожидали увидеть здесь, если привыкли к традиционной веб-разработке, – это функция `render`. Эта распространенная функция в игровых движках для 3D и 2D. Рендеринг сцены должен происходить каждые несколько миллисекунд. Функция `scene.render()` в основном собирает все в сцене, преобразовывает это, материалы и все остальное и повторно отображает все в `<canvas>` на основе данного момента времени. Если это не делать несколько раз, все просто застрянет на месте, не двигаясь. Эта функция также является неплохим местом, для того чтобы добавить пользовательский код, который постоянно обновляет каждый кадр. Например, если бы вы перемещали объект из точки A в точку B, то могли бы постоянно увеличивать расположение. Поэтому создается впечатление, что он плавно перемещается.

Последнюю часть кода можно увидеть ниже, где мы обновляем объект-примитив (или сетку) на новый тип при изменении, а также все изменения цвета или альфа-канала.

**Листинг 15.14 Функции для обновления трехмерного объекта, цвета и альфа-канала**

```
updateColor() {
    if (!this.currentMesh) {
        return;
    }
    const material = new BABYLON.StandardMaterial('material', this.scene);
    if (this.color) {
        material.diffuseColor = new
      BABYLON.Color3.FromHexString(this.color);
    }
    if (this.alpha) {
```

Создаем совершенно новый материал (или что-то вроде 3D-стиля) для объекта

Устанавливаем цвет материала, используя текущее свойство цвета компонента

```
        material.alpha = this.alpha/100;
    }
    this.currentMesh.material = material;
}

switchMesh(mesh) {
    if (this.currentMesh) {
        this.currentMesh.dispose();
    }
    switch (mesh) {
        case 'sphere':
            this.currentMesh = BABYLON.MeshBuilder.CreateSphere
            ("sphere", {}, this.scene);
            break;

        case 'cube':
            this.currentMesh = BABYLON.MeshBuilder.CreateBox
            ("cube", {}, this.scene);
            break;

        case 'geodesic':
            this.currentMesh = BABYLON.MeshBuilder.CreateSphere
            ("sphere", { segments: 2 }, this.scene);
            break;
    }
    this.updateColor();
}
```

Устанавливаем значение для альфа-канала и прозрачности материала

Устанавливаем материал сетки (3D-объекта), используя новый материал, который мы создали

При создании новой сетки избавляемся от текущей, которая есть у нас в сцене

Создаем новую сетку, добавляя ее в сцену

После появления новой сетки материал также нужно обновить

С этими последними изменениями мы только что создали симпатичное маленькое 3D-приложение! Возможно, вы хотите пойти дальше и добавить дополнительные функции, такие как текстуры, отражения, рельефные карты, импорт сцен и выбор объектов и т. д. Это больше, чем может здесь поместиться, но начало неплохое.

Мы, безусловно, хорошо разбираемся в некоторых совершенно разных 3D-сценариях. Недавно я сам довольно много занимался разработкой 3D-приложений, и веб-компоненты сыграли важную роль в организации проекта и разделении ответственностей. Будучи разработчиком прототипа, мне нужно было постоянно менять эти приложения изо дня в день по мере редизайна. Во многих случаях это так же просто, как переместить компонент из одного места в моей HTML-разметке в другое. Даже если этот компонент представляет собой всю трехмерную сцену целиком, это не имеет большого значения. И конечно же, когда мне нужно поработать над некой основной 3D-функцией, я могу мысленно переключить контексты со своего двумерного пользовательского интерфейса, открыть структуру проекта 3D-компонента и работать там.

## 15.3 Видеоэффекты

В смешанной реальности мне нравится то, что у меня есть окно, чтобы видеть мир по-другому. Хотя дополненная реальность добавляет виртуальные объекты в реальный мир, мне всегда нравилось, что можно полностью

изменять то, как мы видим мир. В течение долгого времени у нас были видеоэффекты в кино и на телевидении, и возможность делать это в цифровом виде не является чем-то новым. Настраивать трансляцию в прямом эфире причудливыми способами может быть действительно забавно.

Обработка пикселей сама по себе может быть сложной, но если для вас это удовольствие, настраивать трансляцию видеоизображения снова и снова может быть непросто. Таким образом, мы сталкиваемся с двумя различными типами сложности. Управление пикселями из видео – сложность, которую мы не скрываем. Вместо этого мы скрываем скучную сложность, заключающуюся в том, чтобы заставить поток работать и показать данные кадра.

### 15.3.1 Обработка пикселей с помощью JavaScript

Некоторое время назад я заинтересовался экспериментами с видео и создал собственный видеокомпонент. Я не буду здесь подробно рассматривать базовый код, а просто использую его в качестве финального компонента книги, чтобы поэкспериментировать и немного развлечься! Я поместил этот компонент в репозиторий для данной книги на сайте GitHub.

Для этого есть два компонента. Первый – это простой видеокомпонент, который использует обычный однопоточный JS для манипулирования видеопикселями. Приведенный ниже листинг представляет собой копию файла videofx/demos/video-simple.html.

**Листинг 15.15 Простая демонстрация воспроизведения видео**

```
<html>
<head>
   <title>Demo: Simple Video Playback</title>
   <script                                    Импортируем модуль компонента
     type="module"
     src="../video.js">
   </script>

   <style>
       wcia-video {                           Задаем размеры компонента
           width: 500px;
           height: 500px;
       }
   </style>
</head>

<body>
<h2>Demo: Simple Video Playback</h2>
<p>
   Simple video playback
</p>
<wcia-video useCamera></wcia-video>           Вставляем компонент на страницу
</body>
</html>
```

Несмотря на то что компонент может принимать атрибут `src=path/to/video`, гораздо проще (и веселее) вести трансляцию в реальном времени со своего компьютера, вместо того чтобы загружать куда-то большое видео, а это именно то, что и делает атрибут `useCamera`. Прямая трансляция с камеры – как раз то, что вы увидите при загрузке страницы.

Хотя технически компонент работает, он пока не меняет видеокадры. Для этого давайте переключим рендеринг на внутренний холст компонента и дадим ему указание, как часто нужно рендерить каждый кадр. Установим еще пару атрибутов:

```
<wcia-video useCamera useCanvasForDisplay canvasRefreshInterval="50">
</wcia-video>
```

Здесь мы решили обновлять холст каждые 50 миллисекунд. Если установить значение вроде 500, то мы увидим очень изменчивое видео. Если значение будет слишком низким, нашему браузеру будет сложно угнаться за нами. В любом случае, теперь, когда мы используем внутренний холст, мы можем перемещаться по нему!

Я включил сюда набор фильтров, которые довольно просты в использовании. Всего лишь установите фильтр для компонента с помощью еще одного блока `<script>` на странице, как показано ниже.

**Листинг 15.16 Настройка видеофильтра**

```
<wcia-video useCamera useCanvasForDisplay canvasRefreshInterval="50">
</wcia-video>

<script type="module">
   import Filters from
     '../filters/canvas/filters.js';
   document.querySelector('wcia-video').canvasFilter =
      Filters.toBlackAndWhite;
</script>
```

Добавляем видеокомпонент на страницу с помощью камеры

Импортируем библиотеку фильтров, предоставленную в компоненте

Настраиваем фильтр компонента

Черно-белый фильтр рендерит каждый пиксель либо в черный, либо в белый цвет и создает потоковое видео в режиме реального времени, как на рис. 15.10, хотя вживую оно намного круче!

**Рис. 15.10 Прямая трансляция с камеры с черно-белым фильтром**

Файл videofx/demos/video-filters.html содержит черно-белый фильтр плюс еще несколько, но я думаю, что лучше сделать данные кадра доступными прямо на странице. В листинге 15.17 показаны изменения в демоверсии, которую можно найти в файле videofx/demos/videocustomfilter.html.

**Листинг 15.17 Эффект снега**

```
<wcia-video
       frameDataMode="imagedata"
       canvasRefreshInterval="50"
       useCamera
       useCanvasForDisplay>
</wcia-video>

<p>Amount of snow</p>
<input type="range" min="0" step=".01" max="1" value="0.7"
   oninput="snow = event.target.value">

<script>
   var snow = .7;
   const customfilter = function(pxs) {
       for (var c = 0; c < pxs.data.length; c+=4) {
           if (Math.random() < snow) {
               pxs.data[c] = Math.random() * 255;
               pxs.data[c+1] = Math.random() * 255;
               pxs.data[c+2] = Math.random() * 255;
           }
       }
       return pxs;
   };

   document.querySelector('wcia-video').addEventListener
       ('frameupdate', function(event) {
       var data = event.detail;
       data.canvascontext.putImageData(
       customfilter(data.framedata), 0, 0, 0, 0, data.width, data.height );
   });
</script>
```

Устанавливаем атрибут, чтобы разрешить событие frameData из видеокомпонента

Ползунок, чтобы изменять количество «снега»

Пользовательская функция для обработки и изменения пикселей в каждом видеокадре

Прослушиваем события обновления кадра и обрабатываем пиксели

Таким образом, эти видеокадры довольно легко обрабатывать – мы просто перебираем данные. Каждый пиксель использует четыре значения, по одному для красного, зеленого, синего цветов и альфа-канала. Хотя альфа-канал (`pxs.data [c + 3]`) здесь не используется, было бы здорово поэкспериментировать с ним, если бы понадобилось изменить компонент таким образом, чтобы у него не было черного фона.

В любом случае, этот пользовательский фильтр используется просто для того, чтобы добавлять случайно окрашенные пиксели в случайных местах в каждом кадре. Сколько этих случайно окрашенных пикселей – определяется значением ползунка. Результат показан на рис. 15.11. Это похоже на то время, когда телевизионный сигнал был еще аналоговым и на экране появлялся «снег» или шум, когда сигнал был слабым.

**Рис. 15.11 Прямая трансляция с камеры с эффектом «снега»**

Получение реальных данных видеокадров может быть мощной штукой! Мы вернемся к этому в самом конце, используя реальное приложение, которое не просто меняет пиксели. Однако, прежде чем мы это сделаем, стоит отметить, что хотя обработка пикселей с помощью JS проходит достаточно аккуратно, она также и довольно медленная. Хотя мой браузер легко справлялся с обновлением холста каждые 50 мс, наша обработка изображений была удивительно простой. В целом использование процессора – не лучший способ для этого. Еще хуже – делать это в JS в своем браузере. Поскольку JavaScript является однопоточным языком (пока вы не познакомитесь с Web Workers – https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers), выполнение этих интенсивных операций может привести к блокировке вашего пользовательского интерфейса, и будет казаться, что все работает вяло.

## 15.3.2 Шейдеры WebGL

Выгрузка пикселей в графический процессор – именно тот способ, с помощью которого можно избежать этой медлительности при необходимости запуска обработки изображений, подобной этой. Теперь, когда графические процессоры являются стандартными для каждого устройства, все большее распространение здесь получает разгрузка отрисовки графики. На самом деле еще несколько лет назад многие CSS-эффекты получили более мощный графический процессор, чтобы все работало намного плавнее.

Чтобы получить доступ к графическому процессору в своем браузере, используйте WebGL. Возможно, вы помните, что в начале этой главы я упоминал, что 3D в сети, включая виртуальную и дополненную реальность, работает на WebGL, но он слишком низкоуровневый, чтобы быть продуктивным для обычного человека.

Я по-прежнему придерживаюсь этой точки зрения! Но я расширил видеокомпонент для использования WebGL. Опять же, что хорошо при работе с классами веб-компонентов, так это то, что в то время, как ваш класс расширяет `HTMLElement`, вы можете расширить свой класс веб-компонента. В случае с WebGL-версией видеокомпонента я добавил несколько дополнительных функций для обработки низкоуровневого кода

шейдера, а также изменил внутренний холст на контекст WebGL вместо типичного двумерного холста. Ниже показана демоверсия, которая находится в файле videofx/demos/videogl-filters.html и содержит несколько разных фильтров WebGL.

**Листинг 15.18 Демоверсия видеокомпонента с WebGL**

```
<html>
   <head>
       <title>Demo: Copy to Canvas</title>
       <script                                  Ссылка, намекающая на использование WebGL
               type="module"
               src="../glvideo.js">
       </script>

       <style>
           wcia-glvideo {                       Обязательно используйте новый
               width: 250px;                    CSS-селектор для разных тегов компонента
               height: 250px;
           }
       </style>
   </head>

   <body>
       <h2>Demo: Apply WebGL Filter</h2>
       <p>
           Apply WebGL Filter - possible glfilters are "sepia",
           "greyscale", "sobel_edge_detection", "freichen_edge_detection",
           "freichen_inverted", and "sobel_inverted"
       </p>
       <wcia-glvideo                            Добавляем видеокомпонент на страницу
               useCamera
               useCanvasForDisplay
               canvasRefreshInterval="10"
               useWebGL='{"filter": "freichen_inverted"}'>
       </wcia-glvideo>                                  Определяем параметры WebGL,
   </body>                                              включая фильтр
</html>
```

С технической точки зрения, шейдер Freichen и Sobel используется для обнаружения краев объектов в видео. Стирая все, кроме краев, вы начинаете приближаться к основам компьютерной визуализации, когда, если пойти дальше, объекты могут начать распознаваться в ваших видеокадрах. Это намного сложнее, чем можно здесь описать. Но в своей непосредственной форме, если не сказать ничего другого, он прекрасно передает эффект штриховки, таким образом, у вас может получиться собственная версия музыкального клипа «Take On Me» группы A-ha, как показано на рис. 15.12!

Написание шейдеров WebGL (крошечных программ, которые манипулируют пикселями) – в действительности сложный процесс, и это иногда приводит в бешенство. Шейдеры написаны как простые строки на JavaScript, без подходящего способа отладки. Также может быть доволь-

но сложно настроить среду, в которой можно писать свои собственные шейдеры. Однако повторюсь: такие веб-компоненты могут упростить эту задачу. Этот же компонент также позволяет писать собственные шейдеры, как показано ниже.

**Рис. 15.12 Эффект обнаружения краев с помощью видеокомпонента на базе WebGL**

**Листинг 15.19 Пользовательские шейдеры, используемые в видеокомпоненте на базе WebGL**

```
<html>
<head>
   <script type="module" src="../glvideo.js"></script>

   <script  <-------------- Тег script, где хранится пользовательский вершинный шейдер
     id="2d-vertex-shader"
     type="x-shader/x-vertex">
// Используйте здесь любой шейдер или вершинный шейдер, включенный в листинг 15.1.
   </script>

   <script  <-------------- Тег script, где хранится фрагментный шейдер
     id="2d-fragment-shader"
     type="x-shader/x-fragment">
// Используйте здесь любой шейдер или фрагментный шейдер, включенный в листинг 15.1.
   </script>

   <style>
       wcia-glvideo { <-- Устанавливаем размеры видеокомпонента
           width: 640px;
           height: 480px;
       }
   </style>

</head>

<body>
<wcia-glvideo  <-------------- Размещаем компонент на странице
       useCamera
       useWebGL
       useCanvasForDisplay
       canvasRefreshInterval="10">
</wcia-glvideo>
```

```
<script type="module">
   import Shaders from '../filters/webgl/shaders.js';
   import Constants from '../filters/webgl/constants.js';

   var video = document.querySelector('wcia-glvideo');
   video.webglProperties.vertexShader =
      document.getElementById('2d-vertex-shader').text;
   video.webglProperties.fragmentShader =
      document.getElementById('2d-fragment-shader').text;
   </script>
</body>
</html>
```

**Устанавливаем в качестве значения для шейдеров WebGL содержимое тега script**

Результат еле заметен, но у нас получился эффект, показанный на рис. 15.13; объект в центре фотографии (моя кошка) имеет идеальную четкость, но по мере того, как фотография все дальше удаляется от внешних краев, она постепенно становится черной.

**Рис. 15.13 Радиальное затухание по краям**

## 15.4 Отслеживание движений рук и машинное обучение

Когда я писал эту последнюю главу, я знал, что хочу сделать еще кое-что с веб-арфой, о которой шла речь в главе 5. У меня было несколько идей касательно компьютерного зрения и отслеживания движений, но затем была опубликована довольно захватывающая статья об отслеживании движений рук с помощью JavaScript: https://hackernoon.com/handtrackjs-677c29c1d585.

Эта относительно новая область машинного обучения включает в себя обучение набора данных или модели, чтобы определять, что правильно, а что нет. В случае отслеживания движения рук эту модель обучали распознавать изображения рук. Хотя все обучение в этом эксперименте проводилось с помощью TensorFlow, программной библиотеки для машинного обучения, разработанной компанией Google, которая написана на Python, обученную модель можно использовать в нашем браузере с помощью Tensorflow.js.

Детали не имеют большого значения. Нужно только объяснить, как все это работает «под капотом». Кроме того, из-за той тяжелой работы, которую потребовалось проделать, автору действительно нужно отдать должное. Его зовут Виктор Дибия, а библиотеку Handtrack.js вместе с отличной демонстрацией можно найти здесь: https://github.com/victordibia/handtrack.js.

Отдавая должное автору и знакомясь с этой технологией, мы можем скрыть его удивительную библиотеку и все сложности этого проекта, создав веб-компонент для отслеживания движений рук! На самом деле, учитывая, что эта библиотека использует исходное видео и отображает детали отслеживания в другой холст, Handtrack.js во многом похожа на внутреннюю работу видеокомпонента, над которым мы работали. Так же, как и в случае с WebGL, класс видеокомпонента можно расширить, чтобы создать специализированный компонент `handtracker`. Это относительно простое расширение показано в приведенном ниже листинге.

**Листинг 15.20 Веб-компонент для отслеживания движения рук**

```
import Video from './video.js';                          Расширяем базовый
                                                         класс видеокомпонента
export default class HandTracker extends Video {
    static get HAND_LOCATION() { return 'onHandLocation'; }
        constructor() {
            super();
            const modelParams = {
                flipHorizontal: true, // поворот, например, видео
                maxNumBoxes: 20,   // Максимальное количество обнаруживаемых ящиков
                iouThreshold: 0.5,    // Порог IOU для подавления не-максимумов
                scoreThreshold: 0.6,  // Порог достоверности для предсказаний
            };
            handTrack.load(modelParams)  <-- Загружаем модель Handtrack.js Tensorflow
               .then(lmodel => {
                    this._model = lmodel;
            });
    }

    runDetection() {
        if (!this._model) { return; }
        this._model.detect(this.dom.video).then(predictions => {
            const pts = [];
            for (let c = 0; c < predictions.length; c++) {
                const centerpoint = {};
                centerpoint.x = (predictions[c].bbox[0] +
                  (predictions[c].bbox[2] / 2));
                centerpoint.y = (predictions[c].bbox[1] +
                  (predictions[c].bbox[3] / 2));
                pts.push(centerpoint);
            }
            this._model.renderPredictions(predictions, this.dom.canvas,
                this.canvasContext, this.dom.video);

            const ce = new CustomEvent( HandTracker.HAND_LOCATION,
                { detail: { points: pts }, bubbles: true, composed: true });
```

```
            this.dispatchEvent(ce);
        });
    }

    init() {
        super.init();
        handTrack.startVideo(this.dom.video).then((status) => {
            this.onResize();
            console.log(this.visibleVideoRect)
            if (status) { this.runDetection(); }
        });
    }

    getCurrentFrameData(mode, noredraw) {
        this.runDetection();
    }
}

if (!customElements.get('wcia-handtracker')) {
    customElements.define('wcia-handtracker', HandTracker);
}
```

Отправляем пользовательские события для найденной центральной точки расположения рук

Запускаем обычную инициализацию видеокомпонента и отслеживание движений рук

Продолжаем обнаружение каждого кадра, не забывая про перерисовку холста исходного компонента

Вы можете спросить: «И это все?» Ну, определенно здесь не так уж и много всего, но есть недостающий фрагмент. Библиотека, которую распространяет автор, не является модулем (исходный код – да, но у нее есть зависимости Tensor-Flow, а я хочу, чтобы этот пример оставался простым); поэтому, вместо того чтобы изощряться, для включения ее в этот компонент мы просто подключим ее на демонстрационной HTML-странице, которую я покажу ниже.

**Листинг 15.21 Демонстрационный HTML-файл Handtracker**

```
<html>
<head>
    <script
        type="module"
        src="../handtracker.js">
    </script>
    <script src="../handtrack.min.js"></script>

    <style>
        wcia-handtracker {
            width: 500px;
            height: 500px;
        }
    </style>
</head>

<body>
    <h2>Demo: Hand Tracker <span id="loc"></span></h2>
    <wcia-handtracker useCamera useCanvasForDisplay
      canvasRefreshInterval="50"></wcia-handtracker>

    <script>
        document.addEventListener('onHandLocation', function(e) {
```

Импортируем веб-компонент handtracker

Подключаем библиотеку Handtrack.js

Размещаем компонент handtracker на странице

```
            if (e.detail.points.length > 0) {
                document.getElementById('loc')
            ➥.innerText = e.detail.points[0].x + ',' + e.detail.points[0].y;
            }
        })
    </script>
</body>
</html>
```

Обновляем текст в заголовке, чтобы показать, где находится первая рука

Не так уж и много для такой сложной и полезной вещи, но она работает как по волшебству! На рис. 15.14 показана демостраница в действии.

Как прекрасно все это ни было, у Виктора эта демоверсия уже запущена и работает. Сама по себе моя демонстрация компонента ничего не добавляет к этому разговору. Тем не менее в качестве веб-компонента с пользовательским событием, отправляемым для уведомления слушателей о расположении рук, теперь мы можем использовать этот компонент в приложении «Веб-арфа» из главы 5.

**Demo: Hand Tracker 256,173**

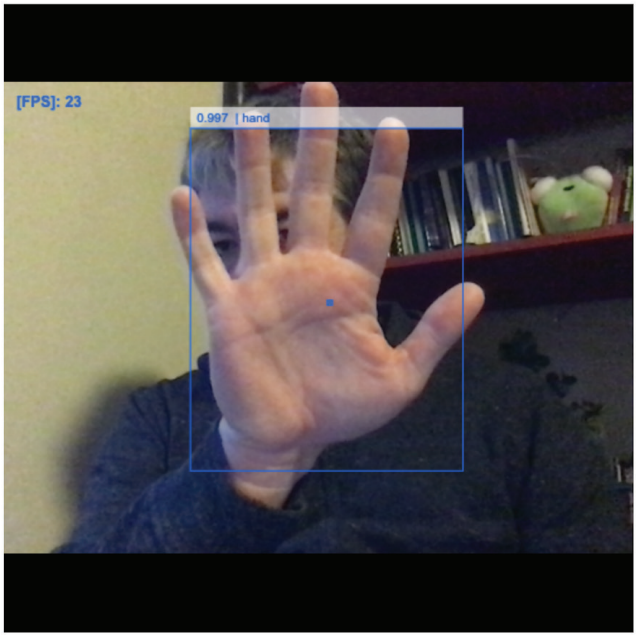



**Рис. 15.14 Демонстрация работы веб-компонента для отслеживания движений рук**

Мало того, что мы можем использовать его, для его интеграции требуется не так уж много усилий! Для начала перейдите в файл index.html приложения «Веб-арфа» и добавьте библиотеку:

```
<script src="../videofx/handtrack.min.js"></script>
```

Поскольку я скопировал код веб-арфы прямо в папку главы 15 из репозитория для этой книги, мы можем создать ссылку внутри папки видеокомпонента, которую только что использовали. Оттуда нам просто нужно изменить webharp/components/app/app.js.

Помните, глава 5 была до того, как мы начали разделять стили и HTML-код в файл template.js, поэтому мы добавим его в строку, с помощью которой устанавливаем значение для свойства `innerHTML`. Еще одно небольшое изменение заключается в том, чтобы теперь мы больше не слушаем событие перемещения мыши, а вместо этого напрямую подключаемся к пользовательскому событию отслеживания движения рук компонента. В приведенном ниже листинге показано, как изменился этот класс компонента.

**Листинг 15.22 Веб-арфа, интегрированная с компонентом handtracker**

```
import Strings from '../strings/strings.js';
import HandTracker from '../../../videofx/handtracker.js';

export default class WebHarpApp extends HTMLElement {
    connectedCallback() {
        this.innerHTML = `
            <style>
                wcia-handtracker {
                    position: absolute;
                    background: none;
                    width: 100%;
                    height: 100%;
                }
                webharp-strings {
                    position: absolute;
                    width: 100%;
                    height: 100%;
                }
            </style>
            <wcia-handtracker useCamera useCanvasForDisplay
               canvasRefreshInterval="50"></wcia-handtracker>
               <webharp-strings
                  strings="${this.getAttribute('strings')}">
            </webharp-strings>`;

        this.stringsElement = this.querySelector('webharp-strings');
        this.addEventListener(HandTracker.HAND_LOCATION, e =>
        this.onMouseMove(e));
    }

    onMouseMove(event) {
            if (event.detail.points.length > 0) {
                this.stringsElement.points = { last: this.lastPoint,
                current: { x: event.detail.points[0].x, y:
                event.detail.points[0].y } };
                this.lastPoint = { x: event.detail.points[0].x, y:
```

Новая HTML-строка, включающая в себя handtracker и CSS для позиционирования позади компонента strings

Изменяем слушателя событий: вместо перемещения мыши теперь идет местоположение рук

```
                    event.detail.points[0].y };
            }
    }
}
```

Устанавливаем точки для первой найденной руки вместо положения мыши

```
if (!customElements.get('webharp-app')) {
    customElements.define('webharp-app', WebHarpApp);
}
```

Вероятно, вы можете начать представлять все возможные способы улучшить это приложение – например, убрав с экрана окно, окаймляющее ладонь. Более того, у вас может возникнуть желание избавиться от ограничения, из-за которого можно отслеживать движения только одной руки, и использовать весь список точек, чтобы играть на арфе обеими руками. Это может быть отличным домашним заданием. Однако сейчас на рис. 15.15 показана веб-арфа в ее нынешнем виде с отслеживанием движения рук.

Должен признать, что экспериментировать с этой веб-арфой было весело. Фактически эта глава посвящена экспериментам с забавными примерами веб-компонентов, которые я либо создал сам, либо нашел в течение последней пары лет.

Хотя я надеюсь, что вы тоже повеселились, я также надеюсь на то, что сведения, которые вы почерпнули из этой последней главы, – это то волнение, которое я испытываю, говоря о веб-компонентах. В главе 2 мы начали с создания самого простого из всех компонентов: ползунка. Мы все еще используем этот ползунок в этой главе в палитре цветов в 3D-приложении для редактирования материалов. Мы перешли от карусели изображений в компоненте к созданию сцен смешанной реальности, которые можно просматривать с помощью гарнитуры Oculus Rift или Magic Leap, и закончили компонентом обработки видеоэффектов, который использует машинное обучение для отслеживания движения ваших рук.

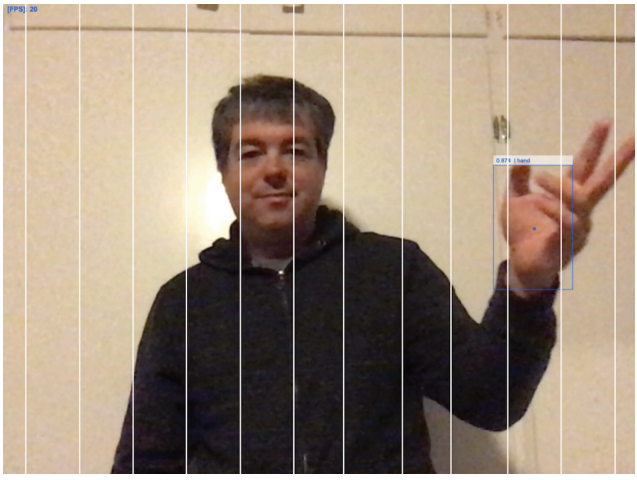


**Рис. 15.15 Веб-арфа с возможностью отслеживания движения рук**

Мы проделали все это, и не имеет значения, насколько вы опытны. Любой из этих компонентов можно добавить на любую обычную HTML-страницу. Не важно, если вы слишком робкий, чтобы заглянуть внутрь какого-либо конкретного компонента, – вы можете просто использовать его. Кроме того, у веб-компонентов действительно простой API, поэтому любой начинающий JS-разработчик может начать работать с ними прямо сейчас.

У нас наверняка будут более сложные рабочие процессы, которые станут популярными по мере того, как разработчики компонентов будут выходить за рамки первоначальных стандартов. Но эти стандарты в ближайшее время не изменятся. В случае с веб-компонентами у нас останется то, что мне всегда нравилось в интернете. Это место, где каждый может внести свой вклад, опираясь на основные строительные блоки внутри или на плечи экспертов и креативщиков. Как бы то ни было, я искренне желаю вам всего наилучшего и надеюсь, что веб-компоненты и эта книга – первый шаг на пути к удивительным вещам, которые вы создаете. Прежде всего, пожалуйста, делитесь с другими! Это одно из преимуществ таких людей, как вы, которые творят сегодня для интернета. Спасибо за то, что читали эту книгу!

## Резюме

Из этой главе вы узнали:

- что может произойти в будущем веб-компонентов;
- как веб-компоненты могут сделать такие пугающие непосвященного технологии, как смешанная реальность и машинное обучение, доступными;
- как скрывать собственные сложные системы, такие как полноценная 3D-сцена в веб-компоненте;
- как использовать компоненты, созданные в этой книге, для новых технологий.

# Приложение ES2015 для веб-компонентов

## A.1 Что такое ES2015?

Раньше изменений в языке JavaScript было немного, и они находились далеко друг от друга. Возможно, вы не знаете, что «JavaScript» в действительности не является официальным названием этого языка – это то, как мы называем его с 1996 года, когда язык Java был королем, а компания Netscape хотела укрепить позиции своего языка LiveScript. Также это был тот год, когда по инициативе Netscape была проведена стандартизация JavaScript ассоциацией ECMA International (www.ecma-international.org).

После принятия его в качестве нового языкового стандарта мы должны были называть его ECMAScript, но такое название не очень удобно произносить. Таким образом, на протяжении более 20 лет этот язык известен как JavaScript (или JS), а стандарт называется ECMAScript. Когда в 1999 году вышел ECMAScript 3, что касается изменений, в течение какого-то периода времени ничего не происходило.

Только в 2009 году была завершена пятая редакция ECMAScript. К сожалению, выпуск четвертой версии был отменен из-за того, что она была основана на языке ActionScript от компании Macromedia и оказалась слишком амбициозной в отношении изменения языка в сознании многих людей. ECMAScript 5 возрастом 10 лет – это стандарт, с которым знакомо большинство из нас. Эту версию также называют ES5.

Поэтому, конечно же, в 2015 году, когда была завершена работа над шестой версией языка, люди называли его ES6, что, к сожалению, было немного неточным! Шестая версия ECMAScript была первой редакцией, которую комитет по стандартам решил назвать в соответствии с годом выпуска, то есть ES2015.

С 2015 года мы видим, что каждый год появляется новая версия. При сжатых сроках изменения были довольно устойчивыми и небольшими. В наши дни более полезно посмотреть, была ли выбранная вами языковая функция принята браузерами, на которые вы ориентируетесь.

Несмотря на некоторые замечательные возможности языка JS начиная с 2015 года, мне бы хотелось сосредоточиться на нескольких основных функциях ES2015 (ES6), которые действительно значительно улучшают разработку веб-компонентов.

# A.2 Переосмысление способа объявления переменных с помощью ES2015

Строго говоря, вы можете продолжать использовать ключевое слово var, когда захотите. Объявление переменных таким образом работает до тех пор, пока существует JS, и в ближайшее время оно не изменится.

## A.2.1 Объявление переменных с помощью ключевого слова let

ES2015 дает нам еще два способа объявления переменных: let и const. С точки зрения использования мало что изменилось – просто стало немного строже и разумнее. С помощью ключевого слова let вы можете объявлять свои переменные, как делали это всегда, используя var:

```
var x = 5; // Старый способ
let x = 5; // Новый способ
```

Разница между let и var зависит от области видимости. Объявления переменных с помощью ключевого слова let более знакомы тем, кто работает с другими языками программирования. Переменные будут существовать только в блоке, с которым они были созданы, а также во всех вложенных блоках внутри. Блоки – это строки кода, заключенные в фигурные скобки, например if/then, цикл for или объявление функции.

Рассмотрим приведенный ниже листинг.

**Листинг A.1 Объявление переменной внутри цикла for**

```
for (var c = 0; c < 5; c++) {
     var message = 'hi' + c;
}
console.log(message);
```

Переменная message объявляется внутри цикла for

Используя ключевое слово var, мы постоянно устанавливаем для переменной message значение «hi» наряду с текущей итерацией цикла. Поскольку в этом цикле мы ограничиваем значение c до 5, в нашем журнале консоли выводится «hi4». Тот факт, что наша переменная вообще содержит что-либо после этого цикла, немного уникален для JS.

Как правило, в других языках наша переменная message будет видна в блоке, в котором она находится, а именно в цикле for. На практике переменная просто не существовала бы вне этой области видимости! Использование ключевого слова let для объявления переменной делает такое поведение поведением по умолчанию.

Давайте изменим var на let, как показано ниже.

**Листинг A.2 Объявление переменной с помощью ключевого слова let внутри цикла for**

```
for (let c = 0; c < 5; c++) {
    let message = 'hi' + c;
}
console.log(message);
```

Объявление переменной выглядит так же, как и раньше, за исключением того, что теперь вместо ключевого слова «var» используется «let»

Теперь переменная `message` не только не определена, но и JavaScript выдает ошибку:

```
Uncaught ReferenceError: message is not defined
```

Еще одна интересная особенность `let` состоит в том, что переменную нельзя использовать до того, как она будет объявлена, в отличие от ключевого слова `var`. Да, здесь требуется разъяснение, потому что если вы не привыкли к JS, то можете подумать: «Как я могу использовать переменную до того, как она объявлена?!» Это возможно благодаря *поднятию*. Поднятие имеет реализации вне простого объявления переменной, но когда переменная объявляется, она «поднимается» или перемещается в верхнюю часть блока.

Рассмотрим этот код:

```
x = 5;
var x;
console.log(x);
```

При поднятии переменной объявление фактически перемещается в верхнюю часть этого блока до выполнения. Таким образом, в действительности переменная `x` объявляется до того, как для нее будет установлено значение 5, несмотря на то что говорит код. С другой стороны, при использовании ключевого слова `let` для объявления переменной `x` появится следующая ошибка:

```
Uncaught ReferenceError: x is not defined
```

Значит ли это, что `let` не поднимает переменную? Нет, на самом деле `x` все равно будет поднята, но между началом блока и моментом, когда код определяет переменную, создается так называемая «временная мертвая зона». Внутри этой мертвой зоны к переменным нельзя получить доступ или установить для них значения. На рис. А.1 показана временная мертвая зона, которая появляется при использовании ключевого слова `let`, и простое поднятие переменной с помощью `var`.


**Рис. А.1 Разница между доступом к переменной до ее объявления с помощью ключевого слова var в сравнении с использованием let. При использовании ключевого слова let появляется «временная мертвая зона», где переменная не доступна**

Теперь вы могли бы спросить себя: «Чем это может быть полезно?» Все, что я написал о `let`, ограничивает нас по сравнению с `var`! Зачем стремиться к появлению ошибок? Здесь действительно важны читабельность кода и намерение, которое вы объявляете с помощью своего кода всем, кто приходит и читает его позже.

Когда кто-то будет читать ваш код, видя ключевое слово `let`, он автоматически будет знать, что вы не используете свою переменную внутри любого другого блока, кроме того места, где он ее видит. Тот, кто читает ваш код, видит, что ваша переменная объявлена внутри цикла `for`? Он будет на 100 % уверен, что ваш код не использует ту же ссылку на переменную где-либо еще. Даже если у нее есть такое же имя в другом блоке, он полностью уверен, что переменные не ссылаются на одно и то же. Использование ключевого слова `let` также убеждает читающего ваш код, в том, что вы не обращаетесь к переменной и не устанавливаете для нее значение, перед тем как объявить ее.

Такое очень строгое поведение с ошибками помогает вашему коду соответствовать сделанным вами обещаниям. Если вы нарушите это обещание, ваш код просто не будет работать! С другой стороны, использование ключевого слова `var` для объявления переменных не дает таких обещаний, и ваш код будет выглядеть довольно неоднозначно относительно ваших намерений.

### A.2.2 Объявление переменной с помощью ключевого слова const

Объявление переменной с помощью ключевого слова `const` практически идентично объявлению с помощью `let`. Вы даете те же обещания людям, которые читают ваш код, что вы не будете использовать переменную до того, как она будет объявлена, и что переменная не определена вне блока, в котором она находится.

Единственное отличие в случае использования `const` состоит в том, что когда вы объявляете переменную с помощью этого ключевого слова, для нее нельзя установить другое значение. Опробуем этот пример:

```
const x = 5;
console.log(x);
x = 6;
```

В данном примере мы устанавливаем для переменной `x` значение, равное 5. В журнале консоли выводится цифра 5. Все прекрасно. Но когда мы устанавливаем значение, равное 6, то получаем ошибку:

```
Uncaught TypeError: Assignment to constant variable.
```

Итак, действительно ли нельзя изменить переменную с помощью ключевого слова `const`? Похоже, что нет, но все зависит от того, что вы подразумеваете под словом «изменить». Мы не можем просто установить для переменной `x` какое-то полностью другое значение, и это правда, но мы, безусловно, можем, так сказать, отредактировать ее. Если наша пере-

менная не является примитивным типом, таким как строка или число, а является объектом, обладающим собственными свойствами, мы можем отредактировать эти свойства:

```
const x = { a: 5 };
x.a = 6;
```

При использовании этого примера ошибки нет. Однако если мы установим для переменной x в качестве значения другой объект, это определенно вызовет ошибку.

### A.2.3 Важно по соглашению, но не по функциональности

Как вы убедились, ключевые слова let и const не дают дополнительной функциональности. Учитывая это, использовать данные новые функции JS совсем не обязательно. Если после прочтения всего этого вы по-прежнему предпочитаете использовать ключевое слово var, у вас действительно не возникнет никаких проблем (помимо упомянутых аспектов читабельности). Тем не менее, учитывая, что вы создаете веб-компоненты, вам необходимо будет использовать хотя бы одну функцию ES2015. Поэтому особого повода отказываться от использования ключевых слов let и const нет. Если другие люди читают ваш код, им, вероятно, будет интересно, почему вы до сих пор используете var.

## A.3 Классы

В других языках классы можно рассматривать как копии чертежей или шаблоны. Когда вы создаете такую копию, то создаете четко определенную конструкцию, из которой можно создавать экземпляры объектов. Класс на самом деле не служит ничему, кроме как быть этим шаблоном. Несмотря на то что классы – это просто копия чертежа, они служат для планирования того, как будут действовать объекты, которые мы создаем из них. Любой экземпляр или объект, созданный из определенного класса, всегда будет действовать определенным образом, потому что программист определил все методы, свойства и логику в классе.

Обычно класс выглядит примерно так:

```
Class MyClass {
      . . .
}
```

А вот как можно создать экземпляр класса (рис. A.2):

```
const myInstance = new MyClass();
```

Однако в JavaScript ситуация с классами обстоит немного иначе. Поскольку JavaScript является языком, основанным на прототипах, классов в истинном понимании этого слова в действительности не существует, как в объектно-ориентированных языках. Вместо этого новая функция класса ES2015 предлагает приятный синтаксис, который делает JS более объектно-ориентированным. «Под капотом» и правда нет «копии» – вы

просто создаете объект времени выполнения, используя эту функцию, который вы клонируете для создания экземпляра.

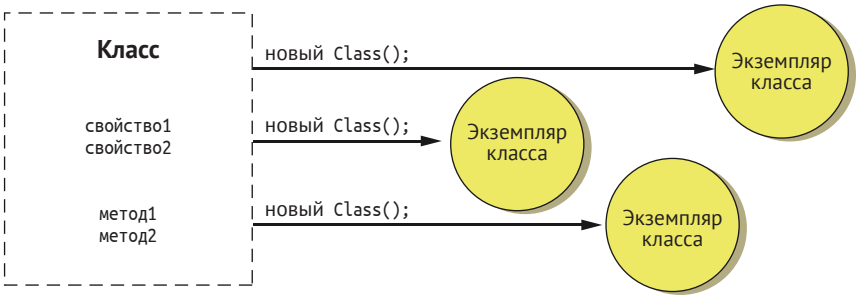



**Рис. А.2 В объектно-ориентированных языках программирования экземпляры класса обычно создаются из класса, своего рода как копия чертежа**

Учитывая это, даже несмотря на то что они выглядят одинаково и предлагают отличную функциональность, классы в JS не предлагают того же, что делают классы в других языках. Существуют хорошие ресурсы для изучения всего, что касается объектно-ориентированного программирования и классов в JS. В этом разделе будут изложены некоторые основы, которые помогут вам освоить концепции, касающиеся веб-компонентов, не вдаваясь в подробности.

## A.3.1 Конструктор

Конструктор является довольно простой концепцией, но используется и упоминается везде, где идет обсуждение классов. С точки зрения использования, между конструктором в JavaScript и конструктором в большинстве других языков на самом деле нет никакой разницы.

Конструктор – это функция, которая описывает любую пользовательскую логику, которая возникает при создании экземпляра класса. Например, можно определить класс в JavaScript с помощью конструктора, который просто выводит информацию в консоль при создании экземпляра класса, как показано в приведенном ниже листинге.

**Листинг А.3 Использование конструктора класса**

```
<script>
    class MyClass {
        constructor() {  <-- Метод constructor()
            console.log('hi from my class');
        }
    }

    let instance = new MyClass();  <-- Создаем экземпляр класса
</script>
```

Просто создавая экземпляр этого класса, мы выводим информацию в консоль. Как правило, любая инициализация логики помещается в конструктор.

Существует одно дополнительное правило при использовании наследования в классе, которое заключается в том, что вы должны вызывать метод `super();` в самой первой строке, даже если у родительского класса нет конструктора (см. ниже).

**Листинг А.4 Необходимо вызвать метод `super()` в конструкторе подкласса**

```
<script>
    class MyParentClass {                    <-- Объявляем базовый класс
    }

    class MyClass extends MyParentClass {    <-- Наследуем базовый класс
        constructor() {
            super();                         <-- При наследовании требуется вызов
            console.log('hi from my class');     метода super() в конструкторе
        }
    }

    let instance = new MyClass();
</script>
```

Вызов метода `super();` – это способ вызова конструктора унаследованного класса, в данном случае `MyParentClass`.

## A.3.2 Свойства

В большинстве языков класс обычно служит «копией» как для определения методов, так *и* для определения свойств. Посмотрите на пример, приведенный ниже:

**Листинг А.5 Общий пример класса для любого языка**

```
class MyClass {
    property1: . . .;    <-- Свойство, объявленное в классе (в чистом JS это не работает)
    property2: . . .;

    method1() {          <-- Метод, объявленный в классе (работает в JS)
       . . .
    }

    method2() {
       . . .
    }
}
```

В JS в классе определяются только методы. Свойства – это отдельная история. Если вы хотите создать переменную, которая существует в области видимости вашего класса, вам нужно создать ее в одном из методов в виде свойства `this`, которое представляет область видимости экземпляра вашего класса. К сожалению, данное различие означает, что ваш код JS, возможно, станет труднее читать. В других языках, где переменные объявляются в самом классе, не составит труда точно выяснить, какие свойства доступны в вашем классе, потому что обычно они объявляются вверху.

В случае с JS мне нравится объявлять свои переменные внутри конструктора, чтобы попытаться восполнить этот недостаток. Если бы я хотел использовать конструктор, чтобы свойства моего класса было легче читать, я мог бы опробовать подход, описанный в приведенном ниже листинге.

**Листинг А.6 Объявление свойств в конструкторе, а не в самом классе**

```
class MyClass {
    constructor() {
        this.property1; <-- Объявляем свойство в классе, у которого нет значения
        this.property2 = <-- Объявляем свойство в классе с начальным значением
            'a starting value';
    }

    method1() {
       . . .
    }

    method2() {
       . . .
    }
}
```

Еще одна вещь, которую вы можете упустить, если работали с другими языками, – это понятие частных, защищенных и общедоступных свойств в вашем классе.

## A.3.3 Закрытые свойства в JavaScript

В традиционных объектно-ориентированных языках программирования, помимо возможности объявления свойств класса, разработчики могут также указать способ доступа к этим переменным. Обычно в этих языках есть три типа свойств класса:

- Private (Закрытый);
- Protected (Защищенный);
- Public (Открытый).

На рис. А.3 показаны все типы свойств при попытке получить к ним доступ извне класса.

Закрытая переменная – это переменная, доступная только из вашего класса. Это означает, что если вы создадите экземпляр класса с помощью ключевого слова `new`, а затем попытаетесь получить доступ к свойству, оно будет неопределенным или сгенерирует исключение, как показано в приведенном ниже листинге.

**Листинг А.7 Псевдокод, где показаны свойства закрытого класса в других языках**

```
Class MyClass {
       private x; <-- Объявляем закрытую переменную в классе

       constructor() {
```

```
        x = 5;
    }
}

instance = new MyClass();
instance.x = 6;
```

Ничего не получилось, потому что свойство является закрытым


**Рис. А.3 Пример различий между открытыми, закрытыми и защищенными переменными (не JavaScript)**

Закрытые переменные предлагают вашему классу защиту от пользователей этого класса, которые приходят и меняют его внутреннее устройство. Будучи создателем класса, вы должны определить, как этот класс используется.

Например, внутри класса есть переменная, которую вы используете для отслеживания чего-либо. Допустим, вы можете отслеживать, сколько раз пользователь нажимал кнопку. Внутри класса мы пишем это: `counter = 0`. Каждый раз, когда пользователь кликает по кнопке, мы увеличиваем значение счетчика: `counter ++`. Как правило, во многих языках этот счетчик может быть закрытой или защищенной переменной. Это не позволит разработчику настроить `myinstance .counter` извне как ему заблагорассудится, тем самым полностью уничтожив фактический счет!

В других языках производный класс также не получает доступ к закрытым переменным. Посмотрите на два этих класса:

**Листинг А.8 Некорректный доступ к закрытому свойству в подклассе (псевдокод)**

```
Class MyParentClass {
    private x;
    constructor() {
        x = 5;
    }
}

Class MyClass extends MyParentClass {
```

Закрытая переменная объявлена в базовом классе

Значение для переменной устанавливается в конструкторе базового класса

```
    constructor() {
      super();
      x = 6;
    }
}
```

Выбрасывается исключение, потому что x – закрытая переменная, которая не объявлена в этом классе

В этом примере, хотя класс `MyClass` и наследует от `MyParentClass`, он не может получить доступ к переменной `x`. Если бы она была защищенной, а не закрытой, то все было бы иначе. Однако в случае с JavaScript эти различия исчезают. Поскольку свойства не объявлены и все они доступны в области видимости класса с помощью слова `this`, различий нет – все свойства являются открытыми. Это означает, что после создания экземпляра вашего объекта из класса на любое свойство или метод, которые можно использовать внутри класса, можно ссылаться вне класса.

Разработчики JavaScript занимались этим некоторое время, еще до того, как классы стали чем-то особенным. Некоторые из них создали для этого довольно гениальные и также неприглядного вида обходные пути. Я сторонник того, чтобы все было просто, и использую нижнее подчеркивание для добавления имен переменных. Примерно так:

```
this._property2 = 'a starting value';
```

Нижнее подчеркивание в имени вашей переменной на самом деле *не делает* ничего. Это соглашение, которое многие из нас используют, чтобы притворяться, что эта переменная не является открытой и к ней нельзя получить доступ извне.

Да, разработчики могут настроить `myobject._counter` вне класса, но, используя это подчеркивание, очевидно, что любой, кто читает код, знает, что он делает «неправильные» вещи.

Хотя я предпочитаю нижнее подчеркивание для простоты и использую его в этой книге, более современный подход заключается в применении еще одной функции ES2015 под названием `WeakMap`. `WeakMap` и `Map` – две схожие концепции в JS. Они используются в качестве хранилища типа «ключ/значение». Объекты `Map` и `WeakMap` не только имеют, возможно, более приятный API по сравнению с простым объектом, они также принимают непримитивные типы данных в качестве ключей (на самом деле `WeakMap` требует наличия непримитивных ключей).

Это означает, что фактически мы можем использовать весь экземпляр класса в качестве ключа. Поскольку нам нужна автоматическая сборка мусора, мы будем использовать объект `WeakMap` для реализации нашей закрытой переменной вместо `Map` в приведенном ниже листинге.

**Листинг A.9 Использование объекта WeakMap для моделирования закрытых свойств в JavaScript**

```
const vars = new WeakMap();

const _private = obj => {
    if (!vars.has(obj)) {
        vars.set(obj, {});
    }
```

Инициализируем объект WeakMap

Объект для доступа к закрытым переменным, организованный по экземпляру класса

```
    return vars.get(obj);
};

class MyClass {
    constructor() {
        _private(this).test =
                'hi from my class';
    }
}
```

Устанавливаем значение для закрытой переменной внутри класса

Поначалу это может сбивать с толку, но потерпите. Сначала мы создаем `WeakMap` для хранения коллекций наших закрытых переменных. Помните, что у одного класса может быть много экземпляров, поэтому мы используем каждый экземпляр в качестве ключа для `WeakMap`, управляемого самим классом.

Итак, если ключ – это экземпляр, что же со значением? Каждое значение представляет собой объект JS, который содержит еще больше пар типа «ключ/значение». Каждая из этих пар является именем закрытой переменной и значением самой переменной, как показано на рис. А.4.

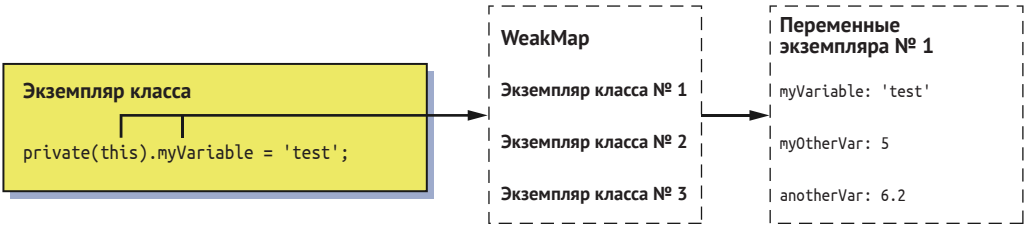



**Рис. А.4 Реализация, в которой используются закрытые переменные, проходящие через `WeakMap` с ключами на базе экземпляров класса. Каждый ключ ссылается на объект, содержащий закрытые переменные**

Наконец, мы также объявляем функцию `_private`, чтобы она помогла нам управлять использованием этого объекта `WeakMap` и упростить получение и настройку закрытых переменных. Помимо получения правильной закрытой переменной для экземпляра, эта функция создает объекты для хранения закрытых переменных отдельного экземпляра.

Похоже, что этот метод представляет собой популярный и современный способ сделать ваши переменные закрытыми. В отличие от простого использования нижнего подчеркивания для обозначения закрытых переменных, этот метод фактически делает переменные недоступными в экземпляре класса. Существует немало обходных путей, подобных этому. Возможно, скоро у нас действительно появится поддержка закрытых переменных (поля закрытых классов уже есть в Chrome Canary), но пока мы можем выбрать только то решение, которое удовлетворяет нашим потребностям, взвесив простоту использования и реальную невозможность доступа за пределами экземпляра класса.

### A.3.4 Геттеры и сеттеры

Еще один набор функций, которые будут иметь отношение к нашей работе с веб-компонентами, – это методы чтения (геттеры) и устанавливающие методы (сеттеры). Геттеры и сеттеры – это методы, которые извне выглядят как свойства. Давайте притворимся, что в приведенном ниже листинге у нас есть класс со счетчиком кликов, который мы не хотели бы изменить извне, но который хотим сделать доступным для чтения.

**Листинг A.10 Счетчик кликов мышью, отслеживаемый внутри**

```
class MyClass {
        constructor() {
                this._counter = 0;
        }

        mouseClickHandler() {
                this._counter ++;
        }
}
```

Инициализируем счетчик в конструкторе класса

Увеличиваем значение счетчика в гипотетическом обработчике mouseClickHandler()

Как было подробно описано в предыдущем разделе, посвященном закрытым переменным, использование нижнего подчеркивания для `this._counter` является примитивной реализацией для обозначения этой переменной как закрытой. Опять же, это делается по соглашению и означает, что мы не должны иметь доступ к `_counter`:

```
let myInstance = new MyClass();
let myCounter = myInstance._counter;
```

Чтобы сделать нашу переменную доступной для чтения, но не для записи, нам нужно определить доступ к ней с помощью метода чтения. Повторюсь, геттеры и сеттеры – это не свойства, к которым можно получить доступ. Вместо этого в приведенном ниже листинге мы создаем метод для нашего класса, который работает как свойство.

**Листинг A.11 Создаем метод, разрешающий чтение, но не запись в свойство**

```
class MyClass {
        constructor() {
                this._counter = 0;
        }

        get counter() {
                return this._counter;
        }

        mouseClickHandler() {
                this._counter ++;
        }
}
```

Геттер для переменной counter

Теперь у нас есть способ запросить переменную counter, но поскольку мы не добавили сеттер, ее настройка на самом деле ничего не даст. Внутреннее значение по-прежнему будет равно 0:

```
let myInstance = new MyClass();
let myCounter = myInstance.counter; // Работает
myInstance.counter = 5;
console.log(myInstance.counter) // Выводит 0
```

Это именно то, что нам нужно! Мы хотим предоставить способ доступа к количеству раз, которое была нажата кнопка мыши, но, как и на рис. А.5, мы не хотим, чтобы кто-то приходил и назначал переменной counter значение в виде любого числа. Конечно, в приведенном ниже листинге мы также можем добавить сеттер, используя set вместо get в определении метода.

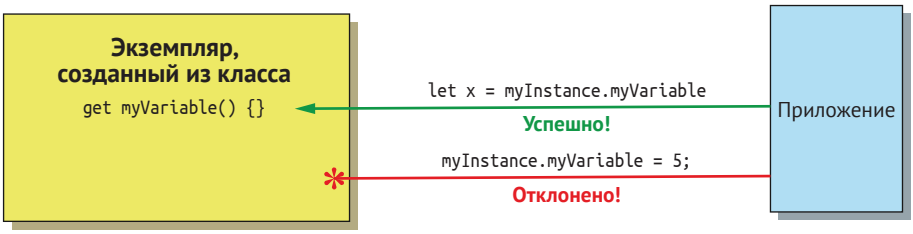



**Рис. А.5 Пример объявления метода чтения в классе без сеттера. Внешнее приложение может получить переменную, но не может установить для нее значение**

**Листинг А.12 Определение методов чтения и устанавливающих методов для класса**

```
class MyClass {
        constructor() {
                this._counter = 0;
        }

        set counter(val) {  <-- Сеттер, дополняющий метод чтения
                this._counter = val;
        }

        get counter() {
                return this._counter;
        }
        mouseClickHandler() {
                this._counter ++;
        }
}
```

Конечно, в контексте этого примера сеттер не имеет особого смысла. Более того, использование и геттеров, и сеттеров для чтения и записи из простого свойства – это немного чересчур. Почему бы просто не сделать counter открытым свойством?

Определение геттеров и сеттеров полезно, когда вы хотите выполнить какой-то код в дополнение или вместо чтения или записи переменной. Например, возможно, что при настройке свойства `counter` также обновляется и график. Здесь это может быть столбец из гистограммы, который растет до значения счетчика, и все это делается просто с помощью метода чтения:

```
set counter(val) {
        this.counterElement.style.height = val + "px";
        this._counter = val;
}
```

## A.3.5 Статические методы

Давайте перейдем к *статическим* методам. Статические методы также называют *методами класса*, потому что они запускаются в самом классе, а не в экземпляре класса.

Начнем с простого, но бесполезного примера: сложим два числа и вернем результат. Конечно, обычно вы просто используете оператор + и складываете их, но давайте сделаем так, чтобы это был метод.

**Листинг A.13 Метод add**

```
class MyClass {
        constructor() {
        }

        add(a, b) {  <-------- Метод сложения, определенный внутри класса
                return a + b;
        }
}
```

Чтобы использовать наш новый метод `add`, нам нужно сначала создать экземпляр класса:

```
let myInstance = new MyClass();
let total = myInstance.add(5, 6); // В сумме получается 11
```

Хотя если подумать, мы не будем использовать какие-либо свойства экземпляра класса. Ранее в нашем примере со счетчиком мы записывали количество как свойство класса. Без использования экземпляра, созданного из нашего класса, для увеличения значения нашего счетчика, у нас не было бы способа узнать, что это было до того, как мы попытались увеличить значение.

Однако в этом случае мы ничего не отслеживаем – нам на самом деле не нужен экземпляр класса; мы просто хотим вызвать функцию и получить результат.

**Листинг A.14 Пример метода класса, использующего ключевое слово static**

```
class MyClass {
        constructor() {
        }
```

```
        static add(a, b) {
               return a + b;
        }
}
```

Функцию в листинге А.14 теперь можно запускать из самого класса (отсюда и название: метод класса):

```
let total = MyClass.add(5, 6);
```

Статические или классовые методы довольно полезны, но они также непосредственно применимы к тому, как мы слушаем атрибуты в веб-компонентах. Это можно увидеть в главе 4, где используется метод `observedAttributes`, чтобы сообщать вашему веб-компоненту, какие атрибуты следует отслеживать на предмет изменений.

Статические методы можно дополнительно сочетать с геттерами и сеттерами. Использование статического геттера может быть отличным способом определения константных значений, которые должны использоваться совместно в вашем приложении.

**Листинг А.15 Статический метод чтения для совместного использования константных значений в вашем приложении**

```
class MyClass {
        static get URL() {
               return "http://myserviceurl/api/v2";
        }
}
```

С помощью данного статического метода чтения этот URL-адрес может использоваться где угодно, даже без получения экземпляра класса:

```
let url = MyClass.URL;
```

## A.4 Модули

Чтобы изучить, что такое модули, нужно взглянуть на одну распространенную функцию, которая есть в других языках: импорт. Рассмотрим приведенный ниже класс, написанный на Java, из учебного курса на сайте www.javatpoint.com/java-swing.

**Листинг А.16 Пример импорта в Java**

```
import javax.swing.*;

public class SwingHelloWorld {
     public static void main(String[] args) {
               JFrame f=new JFrame();
               JButton b=new JButton("click");
               b.setBounds(130,100,100, 40);
               f.add(b);
               f.setSize(400,500);
               f.setLayout(null);
```

Добавляем кнопку в пользовательский интерфейс

Создаем экземпляр кнопки

Устанавливаем значения для осей x и y, ширину и высоту

Устанавливаем размер кнопки

```
                f.setVisible(true);
    }
}
```

В этом примере на базе Java мы программными средствами создаем кнопку и помещаем ее в окно. Если не знать язык Java, то это выглядит довольно просто, не так ли? Нечто очень похожее и лаконичное можно было бы сделать и с помощью HTML и JavaScript. Разница в случае с JavaScript заключается в том, что мы будем использовать пространство имен `document` для создания нашей кнопки:

```
document.createElement('button');
```

## A.4.1 Объекты верхнего уровня в JS

Вы когда-нибудь задумывались обо всех этих методах, которые мы используем каждый день, из объектов `document` или `window`? Их много, и хотя поначалу вас это может ошеломить, с этим можно справиться, как только вы привыкнете. Эти объекты верхнего уровня или глобальные объекты предназначены для управления DOM и вашими визуальными элементами, находящимися внутри модели. Между тем существуют и другие глобальные объекты, которые используются для иных целей. Мы выводим журналы в нашу консоль с помощью `console.log` и можем выполнять парсинг строк в формате JSON с помощью `JSON.parse`. У нас также есть объект верхнего уровня `Math`, который можно использовать для тригонометрических вычислений, создания случайных чисел и многого другого.

Когда вы думаете обо всех этих объектах верхнего уровня, о которых мы как разработчики JavaScript должны знать, все это может показаться немного хаотичным. С другой стороны, если посмотреть на пример, приведенный в листинге A.16, можно заметить такие объекты, как JFrame и JButton, которые используются для создания окна и кнопки соответственно, – но откуда они появились?

Чтобы ответить на этот вопрос, нужно учитывать, что графические интерфейсы не обязательно являются тем, что делают разработчики Java. Многие этим занимаются, но многие также будут счастливы выполнять работу, связанную с backend-программированием. Учитывая широкий спектр всего, что Java нужно делать и с чем приходится работать, когда дело доходит до сторонних библиотек, в Java, как и в большинстве других языков, существует функциональная возможность – `import`.

Обратите внимание на импорт `javax.swing.*;` в верхней части класса. На самом деле это сокращение. Расширенный вариант выглядел бы так:

```
import javax.swing.JFrame;
import javax.swing.JButton;
```

Используя синтаксис .*, мы импортируем все классы или вложенные классы в `javax.swing` и делаем их доступными по имени в классе, в который вы их импортировали, поэтому у `JButton` хватает смекалки создать визуальную кнопку.

## A.4.2 Синтаксис модуля для импорта и экспорта

До сих пор в JavaScript, если говорить о браузерах, никогда не было встроенного способа управления внешними зависимостями, кроме использования тега <script>. Сторонние библиотеки, такие как require.js, пытались восполнить этот пробел, но это так и не было принято в качестве спецификации. Теперь у нас официально есть нативная функция модулей. Для того чтобы использовать модули, которые допускают применение импорта, как и другие языки, нужно сделать кое-какие настройки.

Сначала давайте подготовим небольшой код, который можно использовать в качестве импортируемого модуля. В отдельном файле мы можем написать всего несколько строк:

```
export default function demo() {
   console.log('demo');
}
```

Если разобрать, что мы написали, очевидно, что здесь мы определяем функцию с именем `demo`, которая выводит «demo». Ключевое слово `export` – то, что делает эту функцию способной импортироваться. Ключевое слово `default` просто сообщает любому коду, с помощью которого импортируется этот сценарий, что эта функция является переменной, объектом или функцией по умолчанию, которая используется при импорте сценария.

Чтобы было понятнее, давайте посмотрим на импорт, который показан в листинге ниже. Для этого нам нужно объявить, что используемый нами тег <script> имеет тип `module`.

**Листинг A.17 Установка типа тега <script> для активации JS-модулей**

```
<script type="module">   <-- Используем тип module
   import DemoModule from "./moduledemo.js";   <-- Импортируем скрипт
   DemoModule();
</script>
```

Мы можем просто импортировать несколько строк кода, который мы только что написали. Название `DemoModule` в данном случае является выдуманным. С помощью этого импорта мы можем вызывать то, что импортируем, как показано на рис. A.6. Поскольку мы объявили нашу функцию как `default` в импортируемом коде, не нужно дополнительно ничего уточнять.

## A.4.3 Работа с несколькими функциями в одном модуле

Все-таки нам нужно немного больше конкретики, если из JS-файла необходимо импортировать не одну функцию, как показано ниже.

**Листинг A.18 Экспорт нескольких функций в одном модуле**

```
export function hi() {   <-- Функция, экспортированная из модуля
   console.log('hi');
```

```
}

export function bye() {
   console.log('bye');
}
```

Дополнительная функция, экспортированная из того же модуля

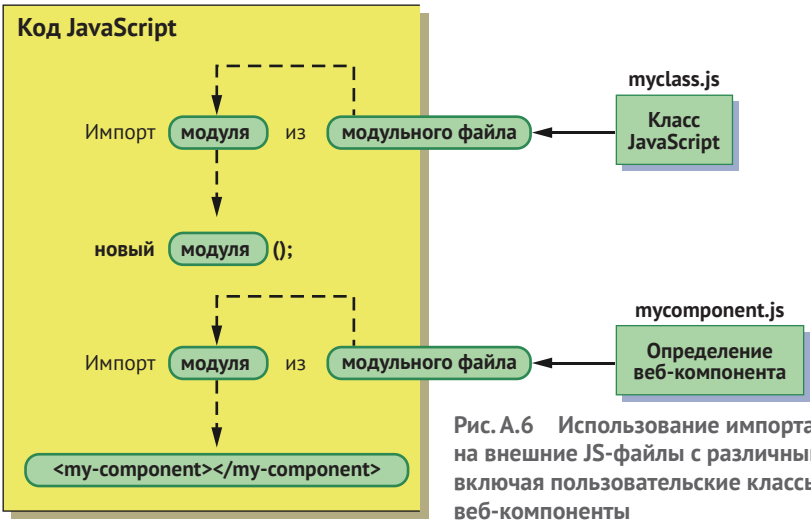



Рис. А.6 Использование импорта, чтобы ссылаться на внешние JS-файлы с различными целями, включая пользовательские классы или даже другие веб-компоненты

Раньше мы могли использовать сокращенный вариант и придумывать любое имя, какое захотим. В приведенном ниже листинге нам нужно использовать реальные имена функций, которые мы определили в модулях, когда их импортируем.

**Листинг А.19 Импорт нескольких конкретных функций из одного модуля**

```
<script type="module">
   import { hi, bye } from "./multiplemoduledemo.js";
   hi();
   bye();
</script>
```

Используем первый из двух экспортов

Импортируем два экспорта из одного модуля

Это не значит, что мы не можем придумать собственные имена, если бы действительно этого хотели. Для этого можно использовать модификатор as.

**Листинг А.20 Альтернативные имена для функций из модуля**

```
<script type="module">
   import { hi as SomeName, bye as SomeOtherName } from
     "./multiplemoduledemo.js";
   SomeName();
   SomeOtherName();
</script>
```

Используем ключевое слово «as» для ссылки на импорт по пользовательскому имени

Наконец, мы можем просто использовать методы `hi` и `bye` для объекта с модификатором `as`.

**Листинг A.21 Альтернативные имена для функций в виде группы из модуля**

```
<script type="module">
   import * as Greeting from
     "./multiplemoduledemo.js";
   Greeting.hi();
   Greeting.bye();
</script>
```

Используем * для импорта всего, что есть под объектом Greeting

Модули отлично подходят для использования в веб-компонентах. В главе 5 подробно описано, как их можно использовать для обеспечения полной самостоятельности ваших веб-компонентов, управляя всеми их зависимостями.

## A.5 Шаблонные литералы

Использование шаблонов в JavaScript в качестве базовой функции языка уже давно назрело. Хотя и верно, что было много библиотек и фреймворков, предлагавших нечто подобное, приятно, что теперь мы можем сделать что-то без внешней библиотеки.

До появления шаблонных литералов разработчики JavaScript использовали одинарные или двойные кавычки для определения строк. Вставка переменных в строки, а также создание многострочных строк – все это выглядело довольно уродливо. Это описано в главе 6, где я знакомлю читателей с шаблонными литералами как наиболее подходящим способом для вставки разметки в наши веб-компоненты.

Что в целом делает для нас шаблонизация? Рассмотрим приведенную ниже строку:

```
`Hi, my name is Ben Farrell and I live in Oakland, CA`
```

### A.5.1 Вставка переменных в шаблонный литерал

Хорошо, если вы оказались мной (и если я не перееду в новый город), но как персонализировать эту строку? Для начала можно использовать несколько переменных:

```
const firstName = 'Ben';
const lastName = 'Farrell';
const city = 'Oakland';
const state = 'CA';
```

Здесь мы извлекли информацию, которая персонализирует эту строку в переменные, – идея, конечно, заключается в том, что вы можете вписать любое имя, город и штат в эту строку. Раньше мы могли сделать это с помощью методов конкатенации строк:

```
const greeting = 'Hi, my name is ' + firstName + ' ' + lastName + ' and I
    live in ' + city + ', ' + state;
```

Данный путь всегда был немного рутинным. Это не страшно, но если вам все время нужно следить за тем, чтобы не забыть добавить пробелы во всех нужных местах вокруг вашей переменной, в сочетании с тем фактом, что это JS-код для представления вашего шаблона, это означает, что на самом деле это не то выражение, которое вы можете использовать откуда угодно.

Вместо этого можно использовать шаблонные литералы, чтобы сделать то же самое:

```
const greeting = `Hi, my name is ${firstName} ${lastName}
    and I live in ${city}, ${state}`;
```

Обратите внимание, что это все одна строка, включая переменные. На рис. А.7 можно увидеть различные варианты использования литералов.

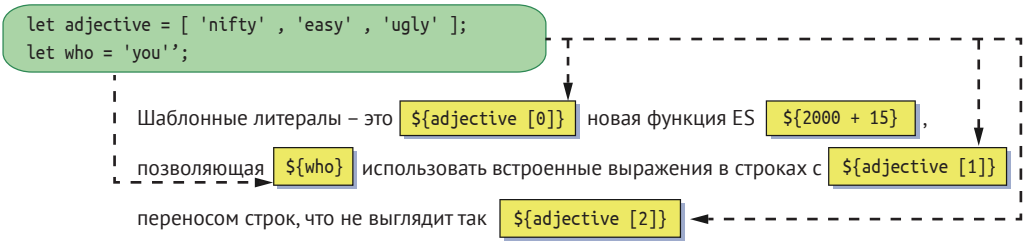



**Рис. А.7 При объединении наших переменных и нашей строки шаблонного литерала мы получаем: «Шаблонные литералы – это отличная новая функция ES2015, позволяющая вам использовать встроенные выражения в строках с простым переносом строк, что не выглядит так безобразно»**

## А.6 Толстая стрелка

Толстая стрелка – это новая функция JS, которая решает давнюю проблему в этом языке. Метод `bind`, помимо `apply`, решил эту проблему до появления ES2015. Толстая стрелка, или функция-стрелка, теперь решает проблему области видимости более понятным способом. В этом отношении данная проблема не является уникальной для веб-компонентов или классов. Толстая стрелка позволяет нам сохранять область видимости везде, где нам это нужно. Специально для классов веб-компонентов она делает наших слушателей событий и функции обратного вызова намного более читабельными и простыми в использовании.

### А.6.1 Проблема, связанная с областью видимости функции обратного вызова

Вероятно, вы знакомы со слушателями событий, так как мы используем их постоянно при проверке событий мыши, ввода с клавиатуры и много-

го другого. Как правило, вы используете их с двумя параметрами, первый из которых – это строка, описывающая событие, которое нужно прослушать, а второй – функция, вызываемая, когда объект, который вы прослушиваете, отправляет событие.

На данный момент мы проигнорируем синтаксис толстой стрелки и рассмотрим обычный вариант:

```
target.addEventListener('mousemove', function(event) {
       ...do something
});
```

Или если у вас уже настроена функция для обработки события:

```
target.addEventListener('mousemove', myFunction);
```

В каждом из этих случаев происходит довольно неприятная вещь: мы потеряли первоначальную область видимости, а в вызываемой функции у нас совершенно новая область. Чтобы было понятнее, взгляните на приведенный ниже листинг.

**Листинг A.22 Пример с таймером, показывающий потерю области видимости в классе**

```
class ScopeTest {
   constructor() {
       this.message = 'hi';
       setInterval(this.onTimer, 1000);
   }

   onTimer() {
       console.log(this.message);
   }
}
let test = new ScopeTest();
```

Запускаем таймер, используя функцию onTimer, которая вызывается при каждом тайм-ауте

Не определено, потому что «this» больше не находится в области видимости класса

В этом примере мы создаем экземпляр класса. Сразу же в конструкторе мы установили для строки с именем `message` значение «hi». Также мы запускаем таймер, который срабатывает каждую секунду. Он вызывает функцию `onTimer`, выводящую в консоль содержание нашей переменной `message`.

## A.6.2 Потеря области видимости в классах

Проблема состоит в том, что при запуске этого кода в консоли выводится слово `undefined`. Почему не выводится «hi»? Что произойдет, если мы изменим наш конструктор, чтобы вызвать функцию напрямую?

**Листинг A.23 Вызов функции напрямую, чтобы избежать потери области видимости**

```
class ScopeTest {
   constructor() {
       this.message = 'hi';
```

```
        this.onTimer();
    }

    onTimer() {
        console.log(this.message);
    }
}
let test = new ScopeTest();
```

Вызываем функцию onTimer напрямую, а не через setInterval

Будет выведено «hi», потому что «this» все еще находится в области видимости класса

В этом случае и вправду выводится сообщение в виде слова «hi», но разница между этими двумя методами зависит от области видимости.

*Область видимости* – это контекст, в котором мы можем получить доступ к переменным, функциям и объектам. На самом деле этот контекст – ссылка `this`. Попробуйте добавить этот код в свой конструктор:

```
constructor() {
    console.log(this);
    this.message = 'hi';
    this.onTimer();
}
```

Вот что будет выведено:

```
ScopeTest {message: "hi"}
```

И конечно, если вы откроете это в инструментах разработчика, то также сможете увидеть там функцию `onTimer`. Использование этой строки кода, `console.log(this)`, дает вам ссылку на ваш класс! Это именно то, что нам нужно для управления кодом в нашем классе, и именно поэтому мы можем обращаться к переменным с помощью ключевого слова `this` или вызывать функции.

С другой стороны, если мы установим надлежащий таймер в конструкторе, используя `set-Timeout (this.onTimer, 1000)`, и напишем уже знакомую строку `console.log(this)`:

```
onTimer() {
    console.log(this);
}
```

то попадем в нераспознаваемую область видимости:

```
Window {postMessage: ƒ, blur: ƒ, focus: ƒ, close: ƒ, frames: Window, . . .}
```

Фактически, если бы вы запускали функцию `onTimer` из конструктора без таймера, ваша область видимости все равно была бы областью видимости вашего класса.

Итак, как вы, вероятно, убедились, когда вы передаете функцию, например, слушателю событий, таймеру или чему-то похожему, что будет вызвано позже, ваша функция обратного вызова внезапно оказывается в новой области видимости! Конечно, это проблематично, потому что тогда возникает вопрос, как снова ссылаться на область видимости вашего класса из функции обратного вызова.

В нашем примере, в рамках нашего обратного вызова таймера, мы не можем получить доступ к нашей переменной `message`. Что, если вместо

этого мы захотим принять меры в отношении DOM нашего веб-компонента? Щелчок мышью был бы крайне полезен для прослушивания, если бы мы не могли предпринять какие-либо контекстные действия с ним.

## A.6.3 Управление областью видимости с помощью толстой стрелки

В JS было много способов решить эту проблему с течением времени, но у нас наконец-то появилась новая функция специально для этого – толстая стрелка. Конечно, мы ласково называем ее толстой, потому что она выглядит как стрелка с толстым основанием – =>.

Чтобы использовать толстую стрелку в приведенном ниже листинге, мы передаем функцию-стрелку, содержащую выражение, которая вызывает наш метод вместо самой функции.

**Листинг A.24 Использование толстой стрелки для сохранения области видимости**

```
class ScopeTest {
   constructor() {
       this.message = 'hi';
       setInterval(() => this.onTimer(), 1000);
   }

   onTimer() {
       console.log(this.message);
   }
}
let test = new ScopeTest();
```

Передаем выражение из толстой стрелки вместо функции

Область видимости класса сохранена; сообщение со словом «hi» зарегистрировано

Используя здесь толстую стрелку, мы можем вывести содержимое `this.message`, и это не будет `undefined`; что еще более важно, мы сохраняем область видимости нашего класса даже посредством функции обратного вызова. Часто вы не увидите пустых скобок в выражении толстой стрелки. Здесь это означает, что мы не передаем параметры, поскольку обратные вызовы `setInterval` их не принимают.

Если бы были параметры для данного метода, эти круглые скобки были бы заполнены, как вы, возможно, и ожидали:

```
callback((x, y, z) => this.onCallback(x, y, z));
```

В качестве примера, который непосредственно относится к нам, давайте вернемся к нашему слушателю события перемещения мыши:

```
this.addEventListener('mousemove', e => this.onMouseMove(e));
```

В этом случае слушатель событий передает событие в функцию обратного вызова, и мы решили использовать это. Мы по-прежнему можем игнорировать это событие и использовать функцию с пустыми скобками, например так:

```
this.addEventListener('mousemove', () => this.onMouseMove());
```

В любом случае, мы можем использовать толстую стрелку, чтобы правильно сохранить область видимости, как показано на рис. А.8, где мы противопоставляем это другим методам, которые мы обсуждали. Что еще более важно для сценариев использования нашего веб-компонента, мы можем сохранить область видимости непосредственно для самого веб-компонента, а написание кода для нашего компонента останется простым, не превращаясь в лабиринт из смешанных областей, которыми мы должны управлять.

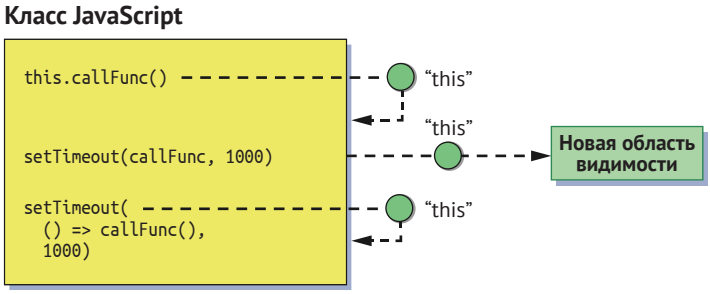



**Рис. А.8 Потеря области видимости или контекста при использовании функции обратного вызова без возможности возврата, и как толстая стрелка может отобразить область видимости туда, где была вызвана функция, точно так же, как при вызове обычной функции**

# Предметный указатель

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу в книготорговой компании «Галактика»
(представляет интересы издательств «ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.

Бен Фаррелл

**Веб-компоненты в действии**

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Беликов Д. А.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура PT Serif. Печать офсетная.
Усл. печ. л. 37,54. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**