1 Магические фронтенд-фреймворки

В ЭТОЙ ГЛАВЕ

- ✓ Почему стоит создать собственный фронтенд-фреймворк
- ✓ Функциональность фреймворка, которую мы построим вместе
- ✓ Как работает фронтенд-фреймворк

Вы когда-нибудь задумывались над тем, как работают фронтенд-фреймворки, которыми вы пользуетесь? Как они решают, когда следует заново отрендерить компонент, и почему обновляют только те части DOM (Document Object Model), которые изменились? Разве не любопытно, что HTML-страница может изменить свой контент без перезагрузки, а URL в адресной строке браузера меняется без запроса новой страницы с сервера? Внутреннее устройство фреймворков невероятно интересно, и нет способа изучить их лучше, чем создать такой фреймворк с нуля. Но для чего изучать, как работают фреймворки? Разве недостаточно просто уметь ими пользоваться?

Хорошие повара знают свои инструменты и умеют мастерски пользоваться ножами. Но настоящие мастера идут дальше: у них есть разные виды ножей, они знают, когда выбирать каждый из них и как поддерживать лезвия всегда острыми. Плотники умеют пользоваться пилой, чтобы распилить дерево, но опытные плотники также понимают, как работает пила, и могут починить ее, если она сломается. Электроинженеры не только знают, что электричество — это поток электронов, идущих через проводник, но и отлично разбираются в приборах,

которыми пользуются для измерения и управления этим потоком. Например, они могут собрать мультиметр, а если он сломается, то и разобрать его, чтобы выяснить причину поломки и отремонтировать.

У вас как у разработчика есть свой инструментарий, в который входит фреймворк. Знаете ли вы, как он работает, или для вас это что-то магическое? Если фреймворк сломался (допустим, вы обнаружили ошибку), сможете ли вы найти причину и устранить ее? Когда одностраничное приложение (Single Page Application, SPA) изменяет маршрут в адресной строке браузера и рендерит новую страницу, не запрашивая ее с сервера, понимаете ли вы, что при этом происходит?

1.1. ДЛЯ ЧЕГО ПИСАТЬ СВОЙ ФРОНТЕНД-ФРЕЙМВОРК

Популярность фреймворков растет — в наши дни мало кто пишет на чистом JavaScript, и это понятно. Современные фронтенд-фреймворки повышают производительность, с ними легко создавать сложные интерактивные приложения. Фреймворки стали настолько популярными, что уже появились шутки о том, что они появляются быстрее, чем их успевают изучать. (К тому моменту, когда вы будете читать эти строки, появятся новые фреймворки.) У самых популярных фреймворков есть свои фан-группы, участники которых ожесточенно спорят, почему именно их вариант лучше всех остальных. Но не стоит забывать, что фреймворк — всего лишь инструмент, то есть средство достижения цели. А цель в данном случае — создание приложений, решающих реальные задачи.

Фреймворки и библиотеки

Не путайте фреймворки с библиотеками. Когда вы используете *библиотеку*, то импортируете ее код и вызываете ее функции. Когда вы используете *фреймворк*, вы пишете код, который выполняется фреймворком. Фреймворк отвечает за запуск приложения и выполняет ваш код при возникновении соответствующих инициирующих событий (триггеров). А функции из библиотеки вы вызываете тогда, когда они вам понадобятся.

Angular — пример фронтенд-фреймворка, тогда как React заявляет о себе как о библиотеке, которая может применяться для создания пользовательских интерфейсов (UI). Но для удобства я буду называть в этой книге фреймворки и библиотеки общим термином *фреймворки*.

Все современные популярные фреймворки (Vue, Svelte, Angular, React) очень эффективны. Зачем создавать свой фреймворк, когда уже есть столько отличных вариантов? Помимо удовольствия и интереса от проектирования сложного программного продукта с нуля, приходится учитывать ряд практических соображений. Приведу небольшую историю из личного опыта.

Как-то в детстве я был в гостях у своего двоюродного брата. Он был на несколько лет старше меня и работал ремонтником. Его шкафы были завалены кабелями, отвертками и другими инструментами. Я мог часами наблюдать за тем, как он чинит разные приспособления. Однажды я принес с собой радиоуправляемую машину, чтобы мы могли с ней поиграть. Он посмотрел на нее, а потом задал вопрос, который оказался для меня совершенно неожиданным: «А ты понимаешь, как работает эта штука?» Я не понимал: я был ребенком и совершенно не разбирался в электронике. Тогда брат сказал: «Я хочу знать, как работает то, чем я пользуюсь. Давай-ка разберем ее и посмотрим, что там внутри». Я до сих пор иногда вспоминаю этот случай.

Теперь я задам вам похожий вопрос. Вы ежедневно используете фреймворки, но понимаете ли вы, как они работают? Вы пишете код для своих компонентов и передаете его фреймворку, который творит свое волшебство. Когда вы загружаете приложение в браузере, оно работает. Оно рендерит представления и обрабатывает действия пользователя, постоянно поддерживая страницу в обновленном состоянии (синхронизированной с состоянием приложения). Для большинства фронтенд-разработчиков, включая меня в прошлом, то, как происходит эта магия, остается загадкой. Остается ли фреймворк, который вы используете, такой же тайной для вас?

Разумеется, большинство из нас слышали о виртуальном DOM. Слышали также и об алгоритме согласования, который выбирает наименьший набор изменений, необходимый для обновления DOM браузера. Мы также знаем, что SPA-приложения изменяют URL в адресной строке браузера без перезагрузки страницы, и если вы относитесь к числу любознательных разработчиков, то могли прочитать о том, как API истории браузера (http://mng.bz/ZRdR) решает эту задачу. Но понимаете ли вы, как все эти части работают друг с другом? Пытались ли разобрать и отладить код используемого фреймворка? Не огорчайтесь, если не пытались. Этого не делали многие разработчики, включая очень опытных. Реверс-инжиниринг далеко не прост и требует огромных усилий и мотивации.

В этой книге мы с вами, как одна команда, будем создавать фронтенд-фреймворк с нуля. Он будет простым, но достаточно полноценным, чтобы показать, как работают фронтенд-фреймворки. После этого работа фреймворков больше не будет для вас загадкой. К тому же этот проект будет действительно увлекательным!

1.2. ФРЕЙМВОРК, КОТОРЫЙ МЫ СОЗДАДИМ

Сразу определимся с ожиданиями: в этой книге мы не будем создавать новый Vue или React. В конце книги вы сможете сделать это сами, добавив все недостающие детали и пару-тройку оптимизаций. Фреймворк, который мы сделаем, не сможет конкурировать с лидерами отрасли, но мы и не ставим такой цели. Эта книга была написана для того, чтобы показать, как вообще работают фреймворки, чтобы они

перестали казаться вам чем-то непостижимым. Для этого не обязательно делать самый мощный фреймворк в мире. Потребовалась бы книга раза в четыре толще этой, а писать такой фреймворк было бы не так интересно. (Я уже говорил, что писать свой фреймворк весело?)

Проект, который мы сделаем, заимствует идеи у реальных фреймворков, в первую очередь у Vue (https://vuejs.org), Mithril (https://mithril.js.org), Svelte (https://svelte.dev), React (https://react.dev), Preact (https://preactjs.com), Angular (https://angular.io) и Hyperapp (https://github.com/jorgebucaran/hyperapp). Наша цель — сделать фреймворк достаточно простым, но включающим типичные возможности фронтенд-фреймворка. Я также хочу, чтобы в нем были представлены некоторые актуальные концепции, заложенные в исходный код самых популярных фреймворков.

Абстракция виртуального DOM используется не во всех фреймворках, но во многих. (Например, Svelte считает ее лишней, что великолепно обосновывается: рекомендую ознакомиться с постом в блоге по адресу http://mng.bz/RmjZ.) Я решил, что в нашем проекте будет реализован виртуальный DOM, типичный для фреймворка, которым вы с большой вероятностью пользуетесь сегодня. Я выбрал подход, который, как мне показалось, будет иметь наибольшую учебную ценность. Виртуальный DOM подробно рассматривается в главе 3, но в двух словах — это облегченное представление DOM, которое используется для вычисления наименьшего набора изменений, необходимых для обновления DOM в браузере. Например, для HTML-разметки

```
<div class="name">
  <label for="name-input">Name</label>
  <input type="text" id="name-input" />
  <button>Save</button>
</div>
```

будет создано представление виртуального DOM, изображенное на рис. 1.1. (Обратите внимание: обработчик события `saveName()` на схеме отсутствует в HTML-разметке. Обработчики событий обычно не определяются в HTML, а добавляются на программном уровне.) В книге будут часто использоваться схемы, показывающие состояние виртуального DOM и работу алгоритма согласования. *Алгоритм согласования* — процесс, который решает, какие изменения нужно внести в DOM браузера для отражения изменений в виртуальном DOM. Эта тема рассматривается в главах 7 и 8.

У нашего фреймворка будут некоторые недостатки, которые делают его не вполне идеальным вариантом для сложных приложений для рабочей среды. Но он все равно остается неплохим вариантом для ваших проектов. Фреймворк поддерживает только стандартное пространство имен HTML (http://mng.bz/27Bg), это означает, что SVG-графика поддерживаться не будет. Многие популярные фреймворки поддерживают это пространство имен, но мы опустим его для простоты.

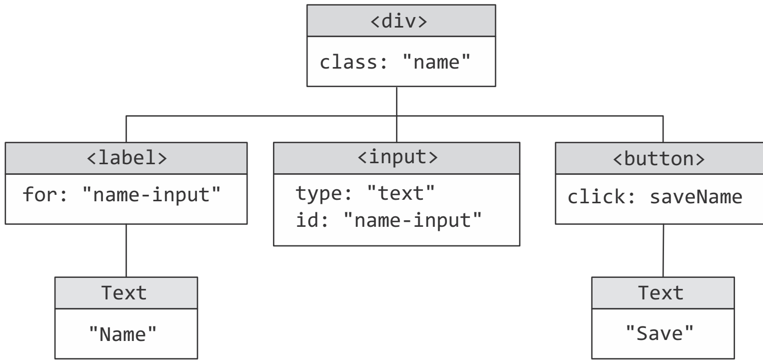

Рис. 1.1. Представление HTML-разметки в виртуальном DOM

1.2.1. Функциональность

К концу книги фреймворк будет обладать следующими возможностями, написанными с нуля:

- абстракция виртуального DOM;
- алгоритм согласования, обновляющий DOM браузера;
- компонентная архитектура, в которой каждый компонент решает следующие задачи:
  - хранение состояния;
  - управление жизненным циклом;
  - повторный рендеринг себя и своих потомков при изменении состояния.

Если захотите продолжить изучение темы, ищите дополнительные главы в репозитории книги на GitHub (приложение А). В них рассматриваются расширенные возможности:

- маршрутизатор SPA, который обновляет URL в адресной строке браузера без перезагрузки страницы;
- слоты для рендеринга контента внутри компонента;
- HTML-шаблоны, компилируемые в функции рендера JavaScript;
- рендеринг на стороне сервера;
- расширение браузера для отладки фреймворка.

Как видите, фреймворк будет относительно завершенным — не полноценным вроде Vue или React, но достаточным для демонстрации того, как работают фреймворки. Важно то, что проект будет создаваться строка за строкой, поэтому вы будете понимать, как все компоненты взаимодействуют друг с другом. В книге много иллюстраций, которые помогут понять достаточно трудные концепции.

Рекомендую писать исходный код самостоятельно в процессе чтения книги. Постарайтесь понять его строка за строкой, не жалейте времени, отлаживайте код и убедитесь, что понимаете решения и компромиссы, которые при этом будут приниматься.

На рис. 1.2 представлена архитектура фреймворка, который мы создадим. На нем изображены все части фреймворка и обозначены взаимодействия между ними. Мы будем неоднократно возвращаться к этой схеме, выделяя каждую часть

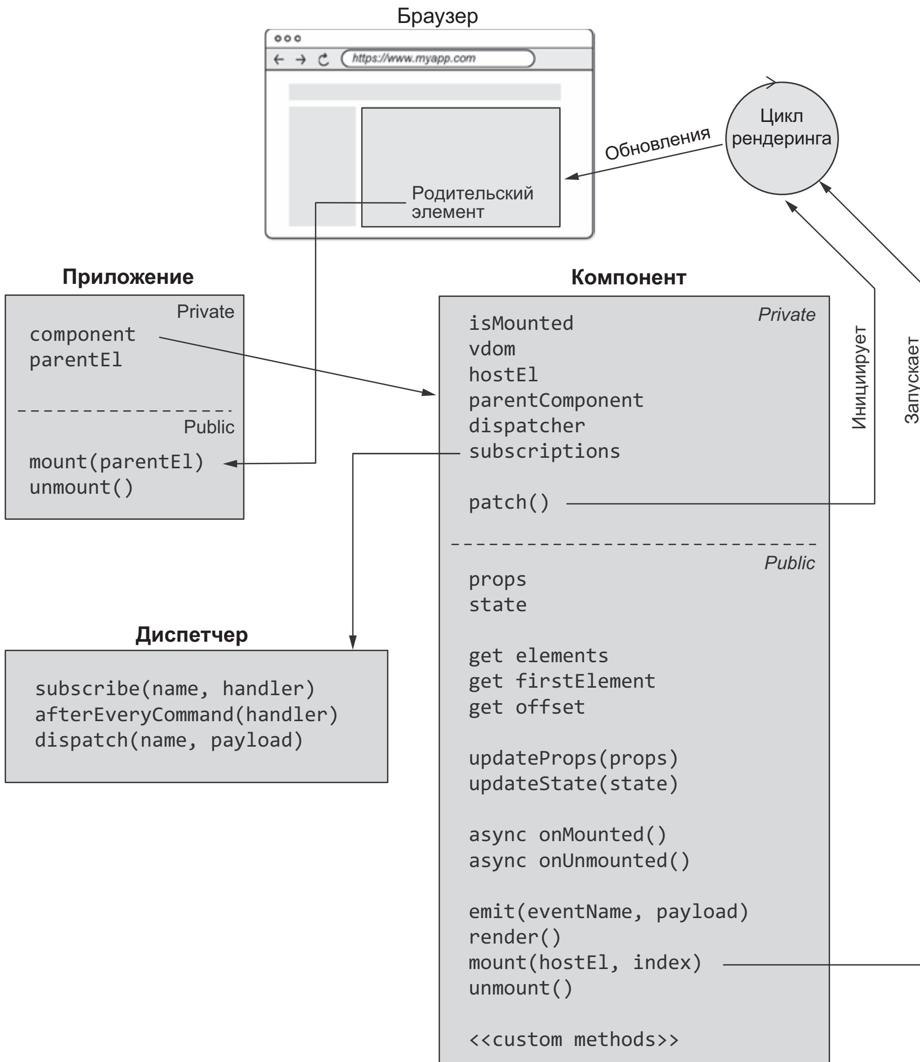

Рис. 1.2. Архитектура фреймворка

фреймворка во время работы над ее реализацией. Понимать все подробности этой архитектуры прямо сейчас необязательно, но желательно представлять ее на высоком уровне. К концу книги этот рисунок станет более понятным: все части будут узнаваемыми, поскольку вы сами их напишете.

1.2.2. План реализации

Как вы понимаете, нельзя создать все в одной главе. Задачу нужно разбить на небольшие части, чтобы действовать методично и постепенно. На рис. 1.3 изображен план реализации фреймворка: он напоминает канбан-доску, на которой каждый стикер представляет работу одной или нескольких глав. В каждой главе мы будем выбирать стикер, то есть подзадачу, которой будем заниматься в этой главе.

Начнем с реализации простого приложения на чистом JavaScript. Она поможет понять, как фреймворки упрощают код фронтенд-приложений. (После этих «страданий» вам будет проще оценить преимущества фреймворков.)

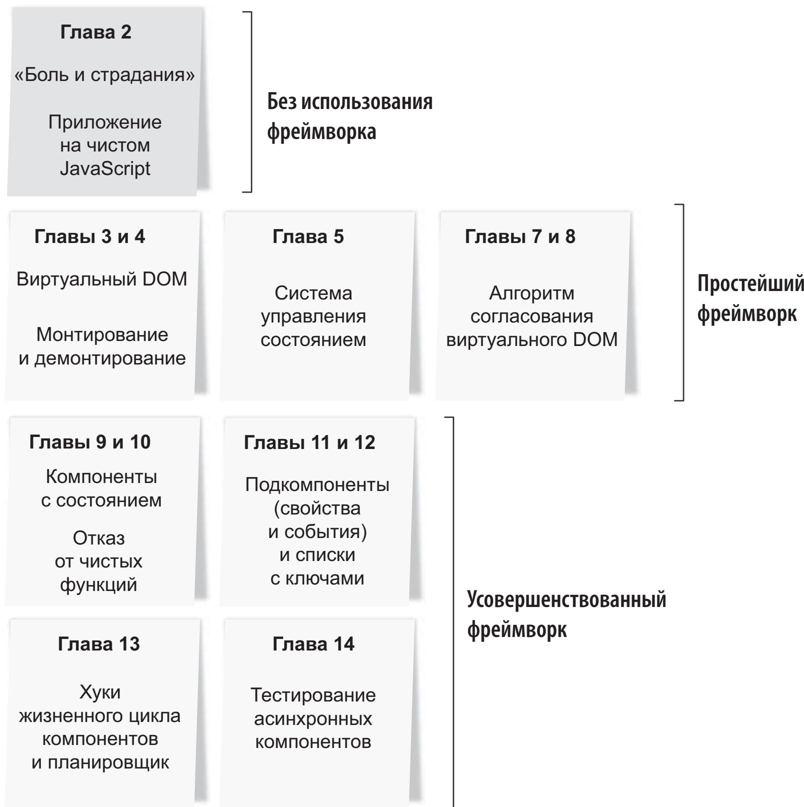

Рис. 1.3. План реализации фреймворка

Компоненты без состояния (stateless) и глобальное состояние приложения

Когда вы начнете лучше понимать преимущества фреймворков, мы выделим части представления приложения в компоненты без состояния, которые будут моделироваться чистыми функциями, возвращающими свои представления в виртуальном DOM. Приложение будет хранить всё состояние приложения и передавать его компонентам, что позволит нам сосредоточиться на алгоритме согласования DOM — вероятно, самой сложной части фреймворка.

Компоненты с состоянием (stateful)

Затем мы предоставим компонентам возможность иметь собственное состояние, что упростит управление состоянием. Приложению больше не нужно будет хранить все состояние целиком — состояние будет распределено между компонентами. Чистые функции преобразуются в классы, реализующие метод `render()`, а каждый компонент станет отдельным мини-приложением с собственным жизненным циклом.

Подкомпоненты

Мы добавим поддержку подкомпонентов, что позволит разбить приложение на небольшие части. Компонент может передавать информацию своим потомкам через *свойства* (props), сходные с аргументами функции. Компоненты также могут взаимодействовать со своими родителями через *события*, на которые родитель может подписаться.

Хуки жизненного цикла

В компоненты будут добавлены *хуки* жизненного цикла. Они позволяют выполнять код в определенные моменты — например, при монтировании компонента в DOM. Например, хук жизненного цикла может загружать данные с удаленного сервера при монтировании компонента.

Тестирование

Каждое приложение нужно тестировать. Как написать юнит-тесты для приложения с использованием нашего фреймворка? Вы узнаете, как тестировать компоненты приложения и как поступать с асинхронными событиями и хуками жизненного цикла.

1.3. ОБЩИЙ ОБЗОР РАБОТЫ ФРОНТЕНД-ФРЕЙМВОРКА

Итак, у нас есть план. Теперь вкратце разберемся, как работают фронтенд-фреймворки. В этом разделе приведен обзор работы фронтенд-фреймворка с внешней точки зрения. (Его внутреннее устройство будет рассматриваться

в оставшейся части книги.) Начнем с точки зрения разработчика — того, кто использует фреймворк для создания приложения, а потом рассмотрим происходящее с точки зрения браузера.

1.3.1. Сторона разработчика

Разработчик начинает создавать проект, используя интерфейс командной строки фреймворка (Command Line Interface, CLI) или устанавливая зависимости и настраивая проект вручную. Важно ознакомиться с документацией фреймворка, поскольку каждый фреймворк работает по-своему.

В веб-приложении разработчики создают компоненты, определяющие часть представления приложения и то, как пользователь взаимодействует с ними. Компоненты пишутся на HTML, CSS и JavaScript. Многие фреймворки используют *однофайловые компоненты* (Single File Component, SFC), у которых весь код компонента (HTML, CSS и JavaScript) размещается в одном файле. Исключение — Angular, в котором для каждого компонента используются три файла: для HTML, TypeScript и CSS. Такой способ размещения позволяет разработчику разделять языки и в теории способствует получению более качественной поддержки синтаксиса в IDE. Но, с другой стороны, разработчику может быть неудобно переключаться между файлами, чтобы увидеть весь компонент.

React и Preact используют JSX — расширение JavaScript — вместо прямого написания HTML-разметки. Другие фреймворки (включая Vue, Svelte и Angular) применяют HTML-шаблоны с директивами добавления и изменения поведения элементов DOM, такие как перебор и вывод массива элементов либо условное отображение конкретных элементов. Следующий код показывает, как можно организовать условное отображение абзаца в Vue:

```
<p v-if="hasDiscount">
  You get a discount!
</p>
```

Директива `v-if` — специальная директива, которую Vue предоставляет для условного отображения элементов. Другие фреймворки используют несколько иной синтаксис, но все они предлагают разработчику способ отображения или скрытия элементов в зависимости от состояния приложения. Для сравнения покажем, как сделать то же самое в Svelte:

```
{#if hasDiscount}
<p>
  You get a discount!
</p>
{/if}
```

и в React:

```
{hasDiscount && <p>You get a discount!</p>}
```

Когда разработчик будет удовлетворен приложением, код нужно упаковать в меньшее количество файлов, чтобы браузер мог загрузить приложение, делая меньшее число запросов к сервису. При этом файлы также могут быть *минифицированы*, то есть уменьшены за счет удаления пробелов и комментариев и назначения переменным более коротких имен. Процесс преобразования исходного кода приложения в файлы, которые поставляются пользователям, называется *сборкой* (building).

Прежде чем развертывать фронтенд-приложение в продакшен-среде, нужно произвести его сборку. Бо́льшая часть работы по сборке конкретного фреймворка выполняется самим фреймворком. Фреймворк обычно предоставляет инструменты командной строки, которые могут использоваться для сборки приложения через запуск простого npm-скрипта, например `npm run build`.

> **ПРИМЕЧАНИЕ** Приложение можно собрать разными способами, что приводит к разным форматам итоговых файлов сборки (бандлов). Здесь я объясню процесс сборки, включающий в себя некоторые из наиболее распространенных практик.

Сборка приложения состоит из нескольких шагов:

1. Компилятор шаблонов преобразует шаблон каждого компонента в JavaScript-код. Этот выполняемый в браузере код создает представление компонента.
2. Код компонентов, разбитый на множество файлов, преобразуется и упаковывается в один JavaScript-файл — app.bundle.js. (Для крупных приложений обычно создается несколько бандлов, которые загружаются по мере необходимости, то есть только тогда, когда станут видимыми для пользователя.)
3. Третий файл — vendors.bundle.js, куда объединяется код сторонних библиотек, используемых приложением. Этот файл включает в себя код самого фреймворка, а также других сторонних библиотек.
4. CSS в компонентах извлекается и упаковывается в один файл CSS: bundle.css. (В больших приложениях могут использоваться несколько пакетов CSS.)
5. HTML-файл, который будет поставляться пользователю (index.html), генерируется или копируется из каталога static-assets.
6. Статические ресурсы (графика, шрифты, аудиоклипы и т. д.) копируются в выходной каталог. Перед этим они могут проходить предварительную обработку для оптимизации графики или преобразования аудиофайлов в другой формат.

Типичная сборка включает четыре файла (или больше в случае больших приложений):

- app.bundle.js с кодом приложения;
- vendors.bundle.js со сторонним кодом;

- bundle.css с CSS приложения;
- index.html — HTML-файл, который предоставляется пользователю.

Файлы отправляются на сервер, а приложение готово к передаче пользователю. Когда пользователь обращается с запросом к сайту, файлы HTML, JS и CSS поставляются статически.

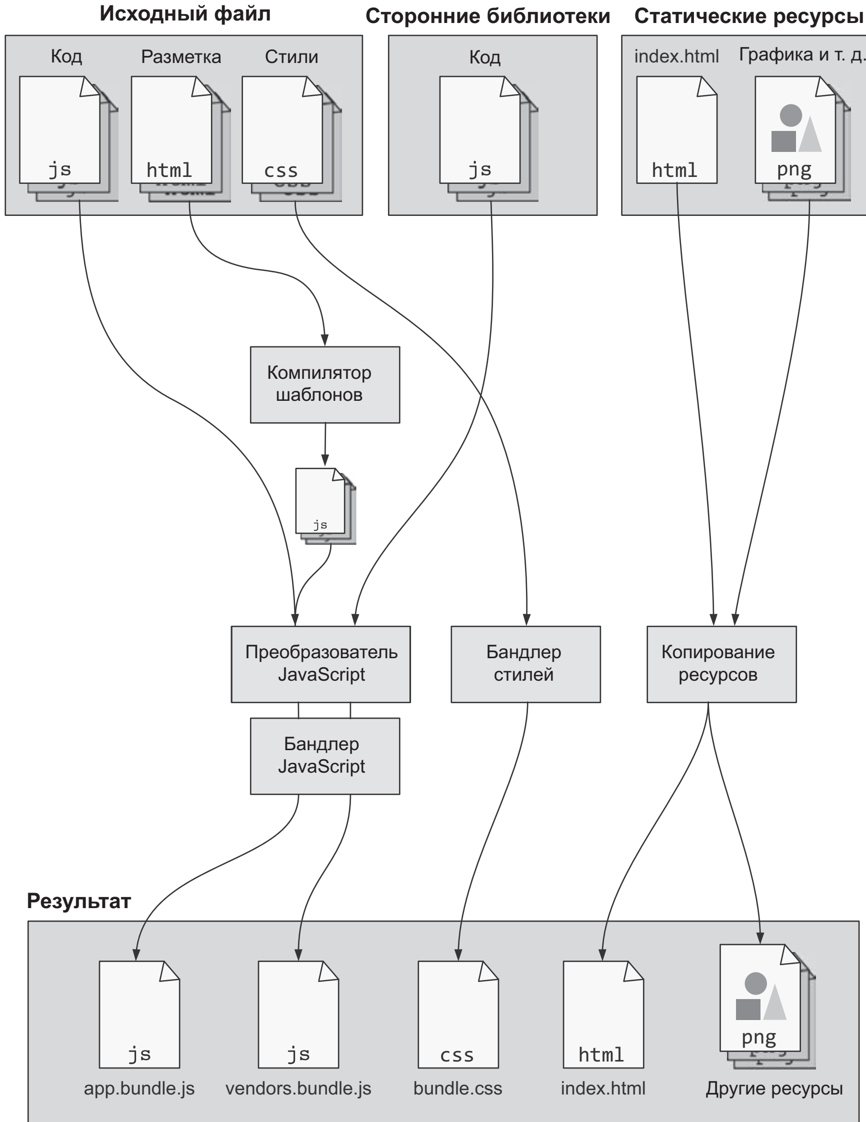

Рис. 1.4. Упрощенная схема процесса сборки фронтенд-приложения