1 Знакомство с React

В ЭТОЙ ГЛАВЕ

- ✓ Что такое React?
- ✓ React для решения задач
- ✓ Интеграция React в веб-приложения
- ✓ Пишем первое приложение React: Hello World

React — воистину революционный инструмент. Многие разработчики даже не знают о том, что он им необходим, но стоит один раз попробовать, и они уже не могут с ним расстаться. Именно так случилось с авторами этой книги и многими другими энтузиастами веб-разработки. React пользуется невероятной популярностью — и на то есть веские причины.

Чтобы заниматься веб-разработкой в начале 2000-х, все, что было нужно, — HTML и серверный язык (такой, как Perl или PHP). Старые добрые времена, когда для обычной отладки кода на фронтенд приходилось набивать его вызовами `alert()`! Время шло, интернет-технологии развивались, и сложность сайтов возросла в разы. Сайты превратились в веб-приложения со сложными интерфейсами, бизнес-логикой и уровнями данных, которые требовали постоянных изменений и обновлений, часто в реальном времени.

Для решения проблем с построением сложных пользовательских интерфейсов (UI, User Interface) было написано много библиотек шаблонов JavaScript. Однако все они требовали, чтобы разработчики придерживались классического разделения обязанностей, то есть разделения стилей (CSS), данных и структуры (HTML) и динамических взаимодействий (JavaScript), и не удовлетворяли современных потребностей. (Помните DHTML?)

React же предлагает новый подход, при правильном применении упрощающий разработку клиентской части. React — мощная библиотека пользовательского интерфейса — предоставляет альтернативу, принятую многими крупными компаниями, такими как Facebook, Netflix и Airbnb, которые сочли ее перспективной. Вместо создания одноразовых шаблонов для пользовательских интерфейсов React позволяет создавать на JavaScript UI-компоненты, которые можно снова и снова использовать на сайтах.

Вам нужен элемент для вывода капчи или выбора даты? Определите в React компонент `<Captcha />` или `<DatePicker />`, который можно добавить к форме: простой подключаемый компонент, содержащий всю функциональность и логику для взаимодействия с бэкендом. Понадобилось поле с автозаполнением, которое обращается с асинхронным запросом к базе данных после того, как пользователь введет четыре буквы или больше? Определите компонент `<Autocomplete charNum="4"/>` для асинхронного запроса. Вы даже можете выбрать, будет ли такое поле иметь интерфейс текстового поля или же вместо него будет использоваться другой нестандартный элемент формы, возможно, `<Autocomplete textbox="..." />`.

Такой подход не нов. Идея построения пользовательских интерфейсов из компонентов появилась давно, но React стал первым, кто предоставил такую возможность с использованием чистого JavaScript без шаблонов. И такой метод оказалось проще обслуживать, повторно использовать и расширять.

React — превосходная библиотека для создания пользовательских интерфейсов, и ее просто необходимо включить в свой веб-инструментарий. Тем не менее она не решает всех задач фронтенд-разработки. В этой главе рассматриваются плюсы и минусы использования React в приложениях и возможность интеграции React в существующий стек веб-разработки.

В этой книге разбираются лишь основы React. Мы хотим, чтобы читатель прочно усвоил базовые концепции и принципы библиотеки React, не углубляясь в изучение сопутствующих или более сложных тем. Сосредоточившись исключительно на React, читатели смогут как следует разобраться в возможностях библиотеки и будут готовы применить полученные знания в разработке разнообразных веб-проектов.

1.1. ПРЕИМУЩЕСТВА REACT

Создатели каждой новой библиотеки или фреймворка утверждают, что их детище по определенным параметрам превосходит своих предшественников. Вначале была библиотека jQuery, и она стала огромным шагом вперед, позволив писать межбраузерный код на чистом JavaScript. Если вам доводилось работать с JavaScript на заре веб-программирования, вы знаете, что один вызов AJAX мог занимать много строк кода, так как разработчику приходилось учитывать специфику Internet Explorer и WebKit-браузеров. С jQuery же стало достаточно одного-единственного вызова: `$.ajax()`. В те дни jQuery иногда даже называли фреймворком — но не сейчас! Теперь под фреймворком понимают что-то более масштабное и мощное.

Точно так же дело обстояло с Backbone, а затем и с Angular; каждое новое поколение фреймворков JavaScript приносило что-то новое. В этом React не уникален. Новое в нем то, что React переосмысливает некоторые базовые концепции, используемые многими популярными фреймворками: например, идею о необходимости шаблонов.

Ниже перечислены некоторые преимущества React перед другими библиотеками и фреймворками, существовавшими на момент появления React:

- *Простые приложения* — React использует компонентную архитектуру с чистым JavaScript, декларативный стиль программирования и мощные, удобные для разработчика абстракции DOM (и не только DOM, но и iOS, Android и т. д.).
- *Быстрый пользовательский интерфейс* — React обеспечивает выдающуюся производительность благодаря своей виртуальной модели DOM и алгоритму интеллектуального согласования (smart-reconciliation algorithm). Одно из дополнительных преимуществ — возможность тестирования без запуска браузера с отсутствующим графическим интерфейсом.
- *Сокращение объема кода* — огромное сообщество React и гигантская экосистема компонентов предоставляют в распоряжение разработчика множество разнообразных библиотек и компонентов. Это важный момент при выборе фреймворка, используемого для разработки.

Благодаря целому ряду особенностей React был проще в использовании по сравнению с другими фреймворками для фронтенда, доступными на первых порах его существования. Однако с момента выхода React появилось много новых фреймворков. Отчасти из-за популярности React разработчики некоторых из них

руководствовались аналогичными концепциями или стремились реализовать те же преимущества, слегка изменяя их параметры. Одни фреймворки были вдохновлены той же идеей, но работали совершенно иначе, тогда как другие были очень похожи на React, но обладали более узкой функциональностью и иногда требовали написания дополнительного кода, но в других случаях заметно сокращали объем кода приложения.

Разберем, что сделало React таким популярным. Это особенности, которые были уникальны для этого фреймворка на момент его выхода, а сейчас появились и у других современных фреймворков. Разберем их подробнее, один за другим. Начнем с невероятной простоты использования.

1.1.1. Простота

Концепция простоты в компьютерных технологиях высоко ценится как разработчиками, так и пользователями. Тем не менее она подразумевает не только простоту использования. Простое решение может быть более сложным в реализации, но в конечном итоге оно оказывается более элегантным и эффективным, а кажущееся на первый взгляд простым нередко оказывается сложным. Простота тесно связана с принципом KISS (Keep It Simple, Stupid, то есть «не усложняй»[1]). Суть в том, что простые системы лучше работают.

Подход React позволяет строить более простые решения с применением кардинально улучшенного процесса веб-разработки. Когда мы начинали работать с React, это стало для нас масштабным сдвигом, который можно было сравнить с переходом от простого JavaScript без фреймворков на jQuery.

В React эта простота достигается благодаря следующим особенностям:

- *Декларативный стиль* (в отличие от императивного) — React отдает предпочтение декларативному стилю вместо императивного, автоматически обновляя представления.
- *Компонентная архитектура, использующая чистый JavaScript*, — React не использует для своих компонентов предметно-ориентированные языки (DSL, Domain-Specific Languages), только чистый JavaScript. Кроме того, отсутствует искусственное разделение работы над одной функциональностью.

[1] https://ru.wikipedia.org/wiki/KISS_(принцип).

- *Мощные абстракции* — в React существует упрощенный механизм взаимодействия с DOM, который позволяет нормализовать обработку событий и другие интерфейсы, работающие одинаково в разных браузерах.

Рассмотрим все эти особенности по очереди.

Декларативный стиль

Декларативный стиль означает, что разработчик определяет, *что должно получиться*, а не *как это делать* шаг за шагом (как в императивном стиле). Но почему декларативный стиль лучше? Его главное преимущество — уменьшение сложности, упрощение чтения и понимания кода.

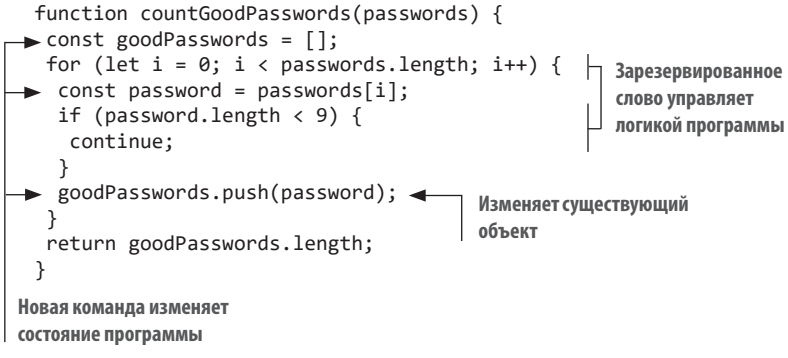
Различия между императивным и декларативным стилем программирования быстро переходят в область чистой теории. Декларативное программирование, в своем максимальном проявлении, становится слишком сложным для понимания без знания таких сложных абстрактных концепций, как монады и функторы. Вот несколько возможных различий между двумя стилями:

- *Выражения вместо команд* — программирование в императивном стиле часто работает с независимыми командами, которые сами по себе изменяют состояние программы, тогда как в декларативном программировании используются выражения, которые в сочетании друг с другом направляют логику выполнения.
- *Зарезервированные слова* — в императивном стиле программирования часто используются многочисленные зарезервированные слова (`for`, `while`, `switch`, `if`, `else` и т. д.), тогда как в декларативном программировании для достижения тех же результатов используются методы массивов, стрелочные функции, обращения к объектам, булевы выражения и тернарные операторы.
- *Композиция функций* — в императивном стиле программирования часто используются независимые вызовы функций и методов, тогда как программирование в декларативном стиле использует композицию функций для определения новых выражений на базе существующих и создания небольших обобщенных логических блоков, композиция которых приводит к нужному результату.
- *Изменяемость* — в императивном программировании часто используются изменяемые объекты и операции с существующими структурами, тогда как в декларативном программировании используются неизменяемые данные, а новые структуры создаются на базе уже существующих (вместо их модификации).

Продемонстрируем различия между стилями на простом примере. Требуется создать функцию `countGoodPasswords`, которая для заданного списка паролей возвращает количество «*хороших*» (сильных) паролей. Хорошим будем считать пароль длиной не менее 9 символов.

Это очень простая задача, которую можно решить на любом языке программирования несколькими разными способами. Для некоторых языков естественнее и удобнее какой-то определенный стиль, но JavaScript занимает особое место, так как он принадлежит обоим мирам. В нем задача может решаться как императивно, так и декларативно.

Начнем с (очень) наивного императивного решения:



Конечно, этот пример немного искусственный, и даже в полностью императивной парадигме программирования программа может быть намного короче.

Реализуем тот же пример в парадигме декларативного программирования:

```
function countGoodPasswords(passwords) {
 return passwords.filter(p => p.length >= 9).length;
}
```

Мы приходим прямо к цели всего одной командой, которая манипулирует с объектом за несколько шагов, применяя композицию функций для достижения нужного результата. Исходный массив фильтруется с переходом к временному значению — массиву, содержащему только хорошие пароли. Однако этот массив нигде не сохраняется; мы переходим сразу к следующему шагу, на котором определяется длина полученного массива.

Впрочем, это был довольно типичный код JavaScript. А как насчет React? React применяет тот же декларативный подход при построении пользовательских интерфейсов. Сначала разработчик описывает элементы интерфейса

в декларативном стиле. А если в представлениях, генерируемых этими элементами, произойдут изменения, React позаботится об обновлении. Вот так!

Удобство декларативного стиля React в полной мере проявляется при необходимости вносить изменения в представление. Они называются изменениями *внутреннего состояния*. При изменении состояния React соответствующим образом обновляет представление.

ПРИМЕЧАНИЕ О том, как работают состояния, говорится в главе 4.

Компонентная архитектура с использованием чистого JavaScript

Компонентная архитектура (CBA, component-based architecture) существовала еще до появления React. Разделение обязанностей, слабая связанность (coupling) и повторное использование кода лежат в основе этого подхода, имеющего массу преимуществ: разработчики, в том числе веб-разработчики, обожают компонентную архитектуру. Структурным элементом CBA в React является класс компонента. Как и другие CBA, он обладает многими преимуществами, главное из которых — повторное использование (ведь так вы пишете меньше кода!).

Главное, чего не хватало до появления React, — реализации этой архитектуры на чистом JavaScript. Работая с Angular, Backbone, Ember и большинством других MVC-подобных фреймворков для фронтенда, вы используете один файл для JavaScript, а другой файл для шаблона (в Angular употребляется термин *директивы* для компонентов).

Использование двух языков (и двух и более файлов) для одного компонента создает ряд проблем. Разделение HTML и JavaScript хорошо работает, если разметка HTML рендерится на сервере, а JavaScript используется только для эффектов типа мигающего текста. Теперь *одностраничные приложения* (SPA, Single Page Applications) обрабатывают сложный пользовательский ввод и выполняют рендер в браузере. Это означает, что HTML и JavaScript жестко связываются в функциональном отношении. Для разработчиков было бы удобнее не разделять HTML и JavaScript в ходе работы над частью проекта (компонентом).

Во внутренней реализации React использует *виртуальную модель DOM* для определения различий (дельты) между текущим содержимым браузера и новым представлением. Этот процесс называется *сравнением* (diffing) в DOM или *соотнесением состояния с представлением* (когда они перестают различаться). Это означает, что разработчикам не нужно явно изменять представление; им достаточно обновить состояние, а представление будет обновляться автоматически по мере надобности. Вы увидите, как эта концепция снова и снова неявно

используется в книге. Мы никогда не манипулируем DOM напрямую, а поручаем React выполнять эту работу за нас.

И наоборот, с jQuery обновления приходится реализовывать в императивном стиле. Манипулируя DOM, разработчик может на программном уровне изменять веб-страницу или ее отдельные части (более вероятный сценарий) без повторной перерисовки всей страницы. Собственно, при вызове методов jQuery происходят именно операции с DOM.

Чтобы оценить, какую поддержку обеспечивает используемый фреймворк, взгляните на рис. 1.1. На одном конце шкалы находится «фреймворк», который вообще вам не помогает. Напишите свое приложение на простом JavaScript, и вы окажетесь в этой точке. Использование jQuery несколько упрощает манипуляции с DOM, но при обновлениях вы все равно не получите никакой поддержки от фреймворка. Вам придется вручную обновлять представления jQuery при обновлении данных jQuery.

Рис. 1.1. Какую помощь оказывает фреймворк? jQuery не делает ничего; Angular делает все. Для некоторых разработчиков React занимает золотую середину между этими крайностями

На другом конце шкалы находятся такие фреймворки, как Angular, — еще один очень популярный фреймворк, сравнимый с React во всех отношениях. Однако Angular работает по совершенно иному принципу, а за кулисами происходит гораздо больше «волшебства». Часто вы просто описываете, как компоненты взаимодействуют друг с другом, а Angular пытается правильно связать их. Проблема в том, что если что-то работает не так, вы часто теряете возможность точного управления. Вы многого не видите, и это приводит к излишнему усложнению.

React находит ту «золотую середину», в которой фреймворк берет на себя большую часть рутинной работы по связыванию компонентов, но не лишает вас возможности точного управления, необходимого для создания сложных веб-приложений. Разумеется, это мнение субъективно, но его разделяют многие разработчики.

Мощные абстракции

React предоставляет ряд замечательных абстракций, которые значительно упрощают жизнь разработчика:

- Синтетические события, абстрагирующие браузерные различия в платформенных событиях.
- JavaScript XML (JSX) абстрагирует JS DOM.
- Независимость от браузера обеспечивает возможность рендера в небраузерных средах (например, на сервере).

В React реализована мощная абстракция модели событий браузера. Иначе говоря, React скрывает нижележащие интерфейсы и предоставляет нормализованные/синтезированные методы и свойства. Например, при создании события `onClick` в React обработчик события получает не платформенный объект события для конкретного браузера, а синтетический объект события, который является оберткой для платформенных объектов событий. Вы можете рассчитывать на одинаковое поведение синтетических событий независимо от того, в каком браузере будет выполняться код. React также содержит набор синтетических событий для событий касания, которые отлично подходят для построения веб-приложений для мобильных устройств.

JSX — один из неоднозначных элементов React. Для некоторых разработчиков абстракция JSX — сильный аргумент в пользу React, тогда как для других она становится препятствием, иногда даже непреодолимым.

Если вы знакомы с Angular, значит, вам уже доводилось писать большой объем кода JavaScript в коде шаблона, потому что в современной веб-разработке простая разметка HTML слишком статична и вряд ли будет применяться на практике сама по себе. Мы рекомендуем дать React шанс и протестировать JSX на практике.

JSX представляет собой *синтаксический сахар* на базе JavaScript, позволяющий записывать элементы React на JavaScript в HTML-подобной записи с <>. React хорошо сочетается с JSX, поскольку разработчикам становится проще реализовать и прочитать код. JSX можно считать своего рода мини-языком, который компилируется в платформенный JavaScript. Таким образом, JSX не выполняется в браузере, а используется как исходный код для компиляции. Рассмотрим компактный фрагмент, написанный на JSX:

```
if (user.session) {
 return <a href="/logout">Logout</a>;
} else {
 return <a href="/login">Login</a>;
}
```

Даже если вы загрузите файл JSX в браузере при помощи библиотеки среды выполнения, которая сразу компилирует JSX в платформенный JavaScript, вы все равно выполняете не JSX, а JavaScript. В этом смысле JSX напоминает

CoffeeScript. Эти языки компилируются в платформенный JavaScript с улучшенным синтаксисом и функциональностью по сравнению с обычным JavaScript.

Мы понимаем, что для кого-то сама идея чередования HTML с кодом JavaScript выглядит странно. Всем новичкам React (как и нам когда-то) требуется время, чтобы свыкнуться с этой идеей, потому что они подсознательно ожидают увидеть лавину сообщений о синтаксических ошибках. Да, использовать JSX не обязательно. По этим двум причинам мы не рассматриваем JSX до главы 3. Однако поверьте на слово: это очень мощный язык. Стоит к нему привыкнуть, и вам будет трудно с ним расстаться.

Другой пример абстракции DOM в React — возможность рендера элементов React на сервере. Она может пригодиться для улучшения поисковой оптимизации (SEO) и/или производительности.

При рендере компонентов React на сервере варианты не ограничиваются DOM или строками HTML. Возможны гибридные решения, в которых шаблоны с контентом проходят рендер на сервере, после чего заново заполняются живыми данными в браузере. Эта опция более подробно рассматривается в разделе 1.3. И если говорить о DOM, одним из самых заманчивых преимуществ React является превосходная производительность.

1.1.2. Скорость и удобство тестирования

Помимо необходимых обновлений DOM, фреймворк может выполнить ненужные обновления, которые только снижают производительность сложных пользовательских интерфейсов. Этот эффект особенно заметен и неприятен для пользователя при большом количестве динамических UI-элементов на веб-странице.

С другой стороны, виртуальная модель DOM React существует только в памяти JavaScript. Каждый раз, когда происходит изменение в данных, React сначала сравнивает различия по своей виртуальной модели DOM; только когда библиотека знает, что в рендере произошли изменения, она обновляет фактическую модель DOM. На рис. 1.2 изображена высокоуровневая схема работы виртуальной модели DOM React при изменении данных.

В конечном итоге React обновляет только те части, для которых это абсолютно необходимо, чтобы внутреннее состояние (виртуальная модель DOM) и представление (реальная модель DOM) не отличались. Например, если присутствует элемент `<p>` и текст дополняется состоянием компонента, то обновлен

будет только текст (то есть `innerHTML`), а не сам элемент. Таким образом обеспечивается повышение производительности по сравнению с повторным рендерингом целых наборов элементов или даже целых страниц (рендеринг на стороне сервера).

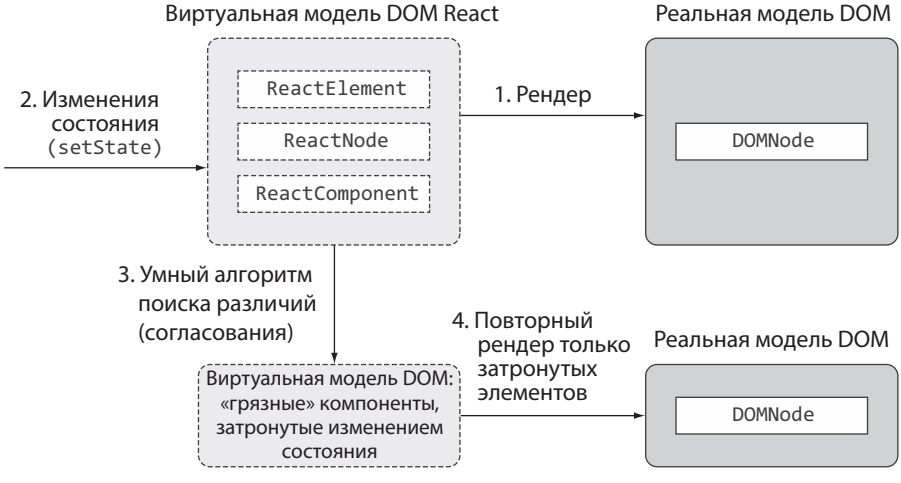


Рис. 1.2. После того как рендеринг компонента завершен, при изменении состояния он сравнивается с виртуальной моделью DOM, находящейся в памяти, и в случае необходимости рендерится заново

ПРИМЕЧАНИЕ

Если вас интересует техническая сторона алгоритмов и нотация «O-большое», следующие две статьи отлично объясняют, как команде React удалось превратить задачу $O(n^3)$ в задачу $O(n)$:

- Reconciliation на сайте React (http://mng.bz/PQ9X);
- React's Diff Algorithm Кристофера Шедо (Christopher Chedeau) (http://mng.bz/68L4).

Еще одно преимущество виртуальной модели DOM — возможность проведения модульного тестирования без браузеров, не имеющих графического интерфейса, например PhantomJS (http://phantomjs.org). Существует ряд библиотек, включая Jest и React Testing Library, которые позволяют тестировать компоненты React прямо из командной строки! Модульное тестирование компонентов React более подробно рассматривается в следующих главах.