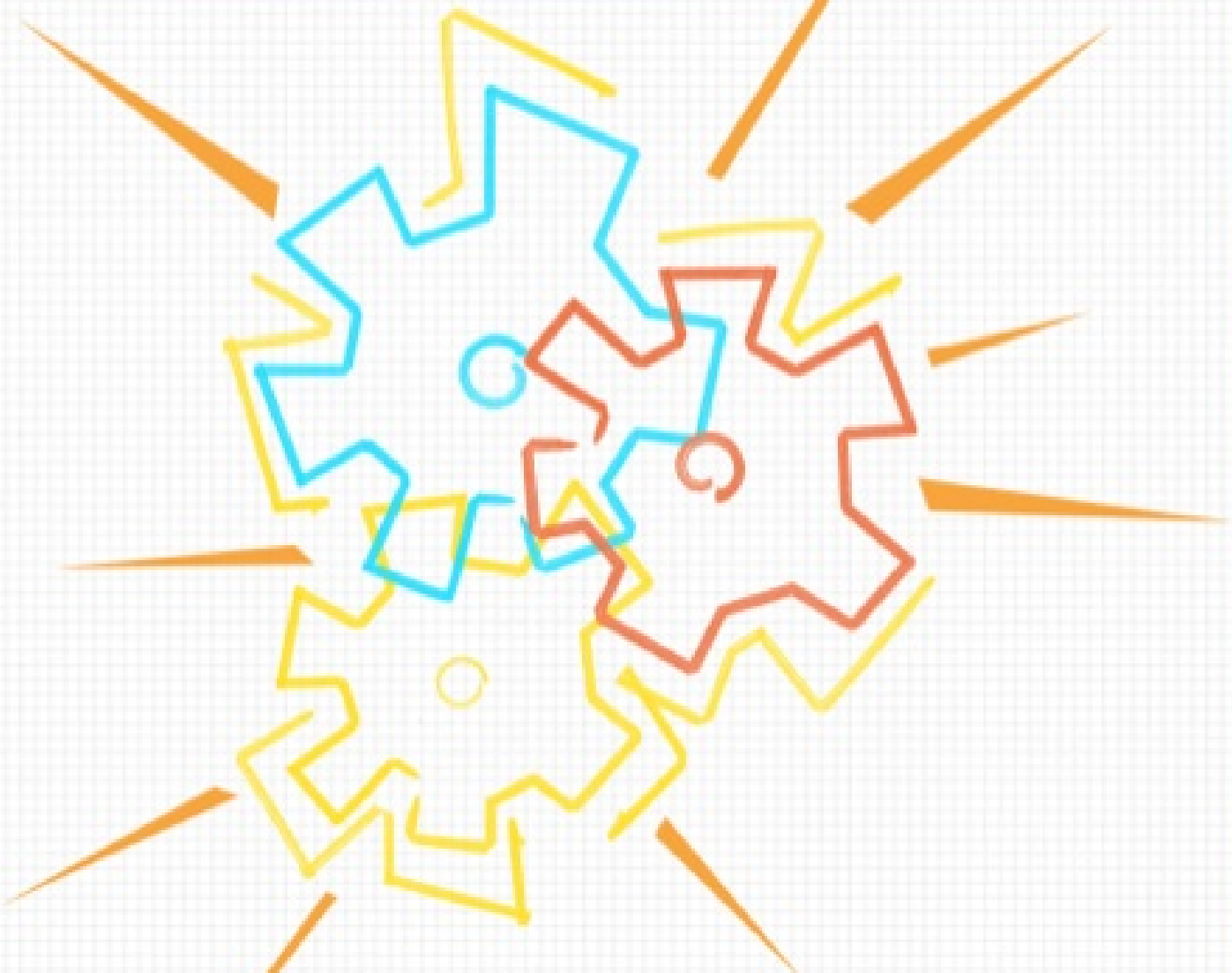
Developing a React Edge

THE JAVASCRIPT LIBRARY FOR USER INTERFACES



Written by: League of Extraordinary Developers

BLEEDING EDGE PRESS

Table of Contents

От автора

Этот перевод возник абсолютно спонтанно, когда однажды я решил углубиться в изучение React.js от Facebook. Перевод не претендует на правильность и уникальность, он был реализован just for fun, только ради лучшего усвоения мной представленного в книге материала.

Глава 1. Введение в React.js

Когда-то давно у всех был единственный путь для создания веб-приложений. Отправлять запрос на сервер и ждать пока он ответит готовой страницей. Очень просто, не надо париться что там происходит в браузере. Только и знай, что создавай одну и ту же страницу каждый раз с нуля.

С языками навроде PHP такой подход реализуется очень просто. Можно даже делать функциональные компоненты и повторно их использовать. Из-за этой простоты PHP и стал одним из самых распространенных языков для веб-разработки.

Но вот кайфа от использования таких простых приложений не было никакого. Юзеры каждый раз ждали ответа от сервера, теряя к тому же состояние страницы до перезагрузки.

Чтобы было интересней тыкать по сайтам, гики решили понаписать библиотек, которые бы помогали рендерить приложения прямо в браузере, используя JavaScript. Эти библиотеки использовали разные средства для манипуляции с DOM-деревом, начиная с простых параметризированных HTML шаблонизаторов, заканчивая системами, что контролировали все приложение. Понятно, что все это со временем превратилось в монстров, которые непонятно как вообще работают, состоя из запутанных и переплетенных между собой событий. По сравнению со старым добрым PHP это просто Адъ и Израиль какой-то.

И было это все до тех пока не начался React. А начался он с порта PHP фрэймворка названного XHP. Назвали его так парни из Facebook, которые задизайнили его чтобы рендерить одну и ту же страницу каждый раз, когда браузер этого попросит. Так вот React и был рожден, чтобы перетащить PHP подход на клиентскую сторону.

Так что же умеет этот ваш React? Целых две вещи:

1. Обновлять DOM-дерево.
2. Реагировать на события.

Он вообще ничего не знает про AJAX, роутинг, про структурирование и хранение ваших данных. Это не MVC фрэймворк. К нему можно применить только V из этой аббревиатуры. Такая узкая сфера применения дает вам свободу использования React в различных системах. По факту, он используется для рендеринга представлений в нескольких популярных MVC фрэймворках.

Отображение страницы каждый раз с сохранением состояния вот что невыносимо медленно в JavaScript. К счастью, React имеет прекрасную систему рендеринга.

Как навороченный игровой 3D движок, React построен вокруг функций рендеринга, которые берут состояние мира и транслируют его в виртуальное представление конечной страницы. Когда React узнает о смене состояние, он повторно запускает функции, чтобы определить новое виртуальное состояние страницы, потом автоматически транслирует результат в изменения DOM-дерева, чтобы отразить новое состояние.

Кажется, что это должно быть медленнее, чем обычный JavaScript подход к обновлению каждого элемента. Но по факту у React есть эффективный алгоритм для определения разницы между текущим состоянием страницы и новым. К тому же, он вносит изменения в DOM только там, где это реально необходимо.

Прирост производительности происходит из-за минимизации перекомпоновки элементов и DOM мутаций, которые обычно в плохой производительности и виновны.

Чем больше ваш интерфейс разрастается, тем скорее всего одно взаимодействие запускает обновление дерева, которое запускает другое обновление, которое запускает другое обновление... Ну, вы поняли. Когда эти наслоенные друг на друга обновление не сгруппированы правильно, производительность ухудшается весьма значительно. И что еще хуже, элементы DOM могут обновляться много раз прежде чем достигнут своего финального состояния.

Но не только виртуальное представление помогает минимизировать проблемы производительности. С React обслуживать ваше приложение гораздо проще. Когда состояние меняется в зависимости от пользовательского

ввода или внешних обновлений, вам просто нужно сообщить React о изменениях и он позаботиться об остальном автоматически. Можно забыть о микроменеджменте.

Содержание

Эта книга раскроет вам 4 главных темы, чтобы помочь начать кодить с помощью React.js.

Создание и композиция компонентов

Первые 7 глав только про создание и композицию React компонентов. Они помогут понять как использовать React.

1. Введение в React.js

Первая глава содержит предысторию и содержание книги.

2. Использование JSX и базовых компонентов React.js

JSX (JavaScript XML) это путь написания декларативного синтаксиса в XML стиле на стороне JavaScript. Вы научитесь как использовать JSX с React и как создавать базовые React.js компоненты. Большинство примеров этой книги и учебное приложение используют JSX.

3. Жизненный цикл компонента React.js

React.js обычно создает и уничтожает компоненты во время процесса рендеринга. React.js предоставляет множество функций которые вы можете использовать в течении жизненного цикла ваших компонентов. Вы научитесь пониманию как управлять жизненным циклом компонента, чтобы избежать утечек памяти вашего приложения.

4. Поток данных в React.js

Это важно знать, как данные путешествуют по дереву компонентов и какие из них можно безболезненно изменить. React.js использует строгое разделение именно данных (props) и состояние (state). Это глава научит вас как использовать данные и состояние корректно в ваших React.js компонентах.

5. Обработчики событий

React.js воплощает их в декларативной манере. Обработка событий это важная часть любого динамического интерфейса. Так научитесь же мастерски работать с ними, на счастье React.js может сделать это весьма легким.

6. Композиция компонентов

React.js заставит вас делать маленькие легковесные компоненты, выполняющие определенную работу. Вам потребуются другие компоненты для контроля ваших компонентов. Эта глава научит вас использованию ваших компонентов внутри других.

7. Миксины React.js

Миксины в React.js дают возможность использовать одинаковый функционал в разных компонентах. Использование миксин — еще один путь раздробить ваши компоненты на маленькие, более управляемые части.

Темы посложнее

8. Доступ к DOM-дереву с помощью React.js

Несмотря на всю мощь виртуального дерева DOM в React.js, вам все-таки понадобится доступ к обычным DOM

узлам вашего приложения. Эта глава научит вас в какой момент жизненного цикла ваших React.js компонентов вы можете безопасно вклиниться в DOM и где лучше перестать его контролировать, чтобы предотвратить потерю ваших DOM узлов.

9. Строим формы с React.js

HTML формы — один из лучших путей получить данные от пользователей. К тому же они легко передают состояние. React.js дает возможность использовать множество состояний элементов формы внутри React.js компонентов. А это в свою очередь дает вам нереальный контроль за элементами форм.

10. Анимации

React.js изменит ваше представление о использовании CSS анимаций. Эта глава расскажет о механизме который React.js предоставляет для управления CSS, что будут анимировать ваши компоненты.

11. Улучшение производительности ваших компонентов.

Виртуальное дерево в React.js дает вам отличную производительность прямо из коробки. Но всегда есть место для улучшений. В React.js есть возможность не ре-рендерить ваш компонент без необходимости. Это может значительно улучшить скорость ваших приложений.

12. Рендеринг на стороне сервера

Большинство приложений нуждаются в SEO, к счастью, React.js может быть быть срендерен в отличном от браузера окружении, таком как nodejs. К тому же, рендеринг на сервере может ускорить первоначальную загрузку вашего приложения. Эта глава раскроет некоторые стратегии для изоморфного рендеринга.

13. Дополнения React.js

React.js предлагает продвинутую функциональность с помощью пакета дополнений. Большинство этих аддонов поставляются в виде миксин. Навык использования этих дополнений поможет вам быстрее создавать продвинутую функциональность с React.js.

Инструменты для React.js

У React есть бомбические инструменты разработчика и инструменты для тестов тоже. Все это умещено в 3 главы.

14. Средства отладки React.js

React.js предоставляет плагин для Google Chrome, с помощью которого можно инспектировать ваши компоненты. Вы научитесь настраивать и использовать этот плагин.

15. Написание тестов для React.js

Написание тестов это возможность не наплодить ошибок в работающем коде с ростом вашего приложения. К тому же это поможет вам писать более модулярный и лучше работающий код. Глава проведет вас по всем аспектам тестирования ваших React.js компонентов.

16. Инструменты сборки

С ростом вашего JavaScript приложения вам потребуются инструменты для автоматической сборки и развертывания вашего кода. React.js поддерживает популярные инструменты для этого: Webpack и Browserify. Глава расскажет вам как настроить их для работы с React.js и JSX.

Работа с React.js

Заключительные 3 главы раскрывают важные аспекты работы с React.

17. Паттерны проектирования

React.js только “V” в аббревиатуре “MVC”. Он более гибок как плагин в других фрэймворках и системах. Эта глава поможет дизайнить большие расширяемые приложения используя React.js.

18. Другие JavaScript библиотеки в семействе React.js

Facebook разрешает свободное использование React.js. В тоже время есть и другие библиотеки, которые разработаны для работы вместе с React.js. Эта глава о них и расскажет.

19. Примеры использования React.js

Хоть React.js и предназначен для использования в веб-среде, он может быть использован везде, где поддерживается JavaScript. Глава расскажет о других примерах использования React.js помимо веб.

Глава 2. JSX

React — одна из тех библиотек, что придерживаются строгих соглашений о том, как структурировать код и управлять данными вашего приложения. JSX является результатом одного из таких соглашений. React воплощает идею, что разметка и код, который генерирует ее по своей природе связаны друг с другом. В React компонентах это реализуется генерацией разметки прямо из JavaScript, используя всю мощь языка.

С этой целью React представляет опциональный язык разметки, очень похожий на HTML. К примеру, вызов функции, создающей заголовок на чистом React может выглядеть вот так:

```
React.DOM.h1({className: 'question'}, 'Questions');
```

Но с JSX это становится похожим на более привычную разметку:

```
<h1 className="question">Questions</h1>
```

В этой главе мы рассмотрим преимущества JSX, способы его использования. Помните, вы можете использовать JSX опционально. Если вы не планируете использовать JSX, в конце главы есть рекомендации по использованию React без него.

Что есть JSX

JSX нужен для JavaScript XML — разметки в стиле XML внутри компонентов React. React работает и без JSX, но именно JSX поможет сделать ваши компоненты более читаемыми, поэтому мы рекомендуем использовать его.

По сравнению с прошлыми попытками встроить разметку в JavaScript JSX обладает несколькими преимуществами:

1. JSX это синтаксическая трансформация — каждая JSX ветвь соответствует JavaScript функции.
2. Не требует никаких библиотек для выполнения.
3. JSX ничего не добавляет и не изменяет в JavaScript — он просто вызывает функции.

По сравнению с HTML JSX дает вам больше возможностей при использовании его с React. Но в конце концов он вызывает старые добрые JavaScript функции. Далее мы расскажем о преимуществах использования JSX внутри вашего приложения, а также, основные различия между JSX и HTML.

Преимущества JSX

Большинство вопросов о JSX сводится к надобности его использования. Зачем использовать что-то еще вместо существующих языков? Почему не использовать просто чистый JavaScript? Ведь JSX просто вызывает JavaScript функции.

Вот несколько плюсов от использования JSX:

- JSX проще в визуализации чем функции JavaScript.
- Разметка понятней для дизайнеров.
- С JSX ваша разметка более семантична и осмысленна.

Простота визуализации

Проще всего показать это на примере. Ниже вы видите функцию рендеринга простого разделителя. Убедитесь, что она проще и читабельнее в формате JSX.

Чистый JavaScript:

```
render: function () {
  return React.DOM.div({className:"divider"},
    "Label Text",
    React.DOM.hr()
  );
}
```

JSX разметка:

```
render: function () {
  return <div className="divider">
    Label Text<hr />
  </div>;
}
```

Многие согласятся, что JSX разметка проще для понимания и в ней проще исправлять ошибки. Вот еще одно ключевое преимущество JSX: людям знакомым с HTML проще работать с ним.

В большинстве команд разработчиков есть люди не умеющие писать код. Начиная от UI & UX дизайнеров, что знакомы с HTML, заканчивая людьми, тестирующими конечный продукт. Этим ребятам гораздо проще читать и исправлять что-то в JSX проекте. Любой человек, знакомый с языками основанными на XML может легко перейти на JSX.

К тому же, так как компонентами React могут становиться все возможные элементы DOM дерева (подробней об этом говорится в разделе о компонентах), JSX дает удобную возможность визуализации структуры в компактной и краткой манере.

Семантика

JSX трансформирует ваш JavaScript код в более семантическую осмысленную разметку. Это позволяет использовать все преимущества при создании структуры вашего компонента в HTML подобном синтаксис, который трансформируется в простые JavaScript функции.

React определяет все возможные HTML элементы в пространстве имен React.dom. Это к тому же позволяет вам использовать вам ваши, полностью кастомные компоненты внутри разметки. Мы расскажем о создании кастомных компонентах позже, пока же просто узнайте как JSX поможет сделать ваш JavaScript более читабельным.

Вернемся к разделителю, о котором мы говорил выше. Он рендерит некоторый текст в заголовке и горизонтальную линию. Разметка для этого разделителя выглядит вот так:

```
<div className="divider">
  <h2>Questions</h2><hr />
</div>
```

Однако, после создания React компонента `<Divider>` вы можете использовать любой другой html элемент, используя все мощность разметки:

```
<Divider>Questions</Divider>
```

Составные компоненты

Вы узнали о плюсах использования JSX и увидели как можно использовать его для определения компонента в кратком формате разметки. Посмотрим как это поможет нам собирать несколько компонентов.

Эта глава расскажет о том как:

- Создавать JavaScript файлы, содержащие JSX.
- Расскажет о сборке компонентов.
- Расскажет об родительских/дочерних отношениях между компонентами.

Давайте углубимся в тему.

Сетап

Хоть JSX и позволяет выразить ваш компонент в форме разметки, на самом деле он трансформируется в JavaScript. Для того чтобы файл с JSX стал доступен для React, в начало файла нужно вставить вот такой заголовок: `/** @jsx React.DOM */`

Без этого JSX не поймет что файл нужно обработать для React.

Вот зачем это нужно:

- JSX транслятор будет игнорировать все файлы без этого коммента.
- Это позволить JSX файлам лежать рядом с не-JSX файлами (обычными JavaScript).

Как говорилось ранее, все элементы в JSX выполняют JavaScript функции. React предопределяет все HTML теги которые только есть в пространстве имен React.DOM. JSX трансформирует их в глобальные переменные, которые позволят вам использовать просто `<div>...</div>` вместо того, чтобы каждый раз нудно указывать `var div = React.dom.div` в каждом JSX файле.

Все ваши кастомные компоненты должны быть определены внутри файла, иначе React не будет знать как выполнить эти функции. Для примера мы определим кастомный компонент для использования в пределах файла.

```
/** @jsx React.DOM */
var React = require("react");

// Определим кастомный компонент, чтобы JSX мог обработать его
var Divider = require("./divider");

// Используем любой React.DOM.* элемент или Divider ниже
<div>
  <Divider>...</Divider>
  <p>...</p>
</div>
```

Вы увидели как создать JSX файл и готовы к созданию кастомного компонента с помощью JSX.

Создание кастомного компонента

К примеру, нам надо получить на выходе следующую разметку:

```
<div className="divider">
  <h2>Questions</h2><hr />
</div>
```

Чтобы сделать этот HTML компонентом React *все что вам нужно* это вызвать функцию рендеринга, возвращающую разметку.

```
var Divider = React.createClass({
  render: function () {
    return (
      <div className="divider">
        <h2>Questions</h2><hr />
      </div>
    );
  }
});
```

Это конечно одноразовый компонент. Чтобы сделать его по настоящему полезным, нужно иметь возможность динамически менять текст внутри тега h2. Но прежде чем затронуть эту тему, поговорим о наследовании.

Наследование

В HTML вы получаете заголовок, используя `<h2>Questions</h2>`, где текст “Questions” это дочерний текст элемента h2. Хочется использовать наш кастомный элемент в JSX похожим образом, например вот так:

```
<Divider>Questions</Divider>
```

React хранит все дочерние элементы между открывающимся и закрывающимся тэгами в массиве свойств компонента, `this.props.children`. В нашем примере `this.props.children === ["Questions"]`.

Вооружившись этим знанием, давайте изменим жестко заданный текст “Questions” с помощью переменной `this.props.children`. Теперь React может срендерить все что вы поместите внутрь тэга `<Divider>`:

```
var Divider = React.createClass({
  render: function () {
    return (
      <div className="divider">
        <h2>{this.props.children}</h2><hr />
      </div>
    );
  }
});
```

Теперь можно использовать компонент `<Divider>` как любой другой HTML элемент.

```
<Divider>Questions</Divider>
```

И JSX транслятор (когда будет обрабатывать) переведет это все в следующий JavaScript код:

```
var Divider = React.createClass({
  render: function () {
    return React.DOM.div(
      {className="divider"},
      React.DOM.h2(null, this.props.children),
      React.DOM.hr(null)
    );
  }
});
```

На выходе будет ровно то, что и задумывалось:

```
<div className="divider">
  <h2>Questions</h2><hr />
</div>
```

В чем отличие JSX от HTML?

JSX похож на HTML, но не в точности повторяет его. В спецификации JSX сказано:

> JSX создан как расширение ECMAScript и похож на XML только для удобства.

Рассмотрим основные отличия JSX от HTML синтаксиса.

Атрибуты

В HTML мы устанавливаем атрибуты внутри каждого элемента, вот так:

```
<div id="some-id" class="some-class-name">...</div>
```

JSX воплощает атрибуты в похожей манере, но его большое преимущество в том, что вы можете задать атрибуты динамическими JavaScript переменными. Чтобы сделать это нужно заключить переменную в фигурные скобки, вместо кавычек:

```
var surveyQuestionId = this.props.id;
var classes = 'some-class-name';
...
<div id={surveyQuestionId} className={classes}>...</div>
```

В более сложных ситуациях вы можете задавать атрибуты как результат вызова функции:

```
<div id={this.getSurveyId()} >...</div>
```

Теперь, каждый раз при рендеринге компонента будет вызвана функция и полученный в результате отобразит новое состояние.

Условные выражения

React воплощает идею о том, что разметка компонента и логика, которой он генерируется неразрывно связаны. Это означает, что вы можете использовать всю мощь логики JavaScript, таких как циклы и условные выражения.

Достаточно сложно добавить условные выражения к вашим компонентам, потому что что логика if/else сложновыполнима как разметка. Добавление if напрямую в JSX выдаст невалидный JavaScript:

```
<div className={if(isComplete) { 'is-complete' }}>...</div>
```

Решением может послужить что-то из следующего списка:

- Использовать тернарную операцию.
- Создать переменную и ссылаться на нее в атрибуте.
- Вынести логику в функцию.

Простые примеры ниже иллюстрируют, как мог бы выглядеть каждый из этих подходов.

Использование тернарной операции

```
...
render: function () {
  return <div className={
    this.state.isComplete ? 'is-complete' : ''
  }>...</div>;
}
...
```

Тернарная операция отлично работает в случае с текстом. Но она может быть весьма сложной и громоздкой в других случаях. Для этих случаев лучше использовать один из методов ниже.

Использование переменной

```
...
getIsComplete: function () {
```

```
  return this.state.isComplete ? 'is-complete' : '';
},
render: function () {
  var isComplete = this.getIsComplete();
  return <div className={isComplete}>...</div>;
}
...
```

Вызов функции

```
...
getIsComplete: function () {
  return this.state.isComplete ? 'is-complete' : '';
},
render: function () {
  return <div className={this.getIsComplete()}>...</div>;
}
..
```

Другие атрибуты (non-DOM)

Следующие атрибуты являются зарезервированными словами в JSX:

- key
- ref
- dangerouslySetInnerHTML

Keys

`key` это опциональный уникальный идентификатор. В процессе выполнение программы компонент может перемещаться вверх и вниз по дереву компонентов. Когда это происходит, ваш компонент может потребовать уничтожения и создания по-новой.

С установкой уникальной ключа, остающегося неизменным в процессе рендеринга вы сообщаете React что нужно более разумно уничтожать и использовать по новой этот компонент, что повышает производительность. В общем, когда два элемента в DOM одновременно меняют свое положение, React может сравнить ключи и переместить их без полной перерисовки их DOM.

References

`ref` позволяет родительским компонентам сохранять ссылку на дочерние компоненты за пределами функции рендеринга.

Вы можете определить `ref` установив атрибут.

```
...
render: function () {
  return <div>
    <input ref="myInput" ... />
  </div>;
}
...
```

Позже вы можете получить доступ к этому `ref` используя `this.refs.myInput` в любом месте вашего компонента. Объект, который возвращает этот `ref` называется **резервным экземпляром** (backing instance). Это не настоящий DOM, но описание компонента, которое React использует, когда нужно будет создать DOM элемент. Для доступа к настоящему DOM элементу нужно использовать `this.refs.myInput.getDOMNode()`.

Более подробное описание наследования дается в 4-ой главе.

Чистый HTML

`dangerouslySetInnerHTML` — используется тогда, когда нужно вывести HTML контент в виде строки, специально для тех случаев когда используются библиотеки, которые манипулируют HTML с помощью строк. Для этого можно использовать HTML в виде строк, но это не рекомендуется, когда можно обойтись без этого. Для использования этого свойства, установите ключ `__html`, например вот так:

```
...
render: function () {
  var htmlString = {
    __html: "<span>an html string</span>"
  };
  return <div dangerouslySetInnerHTML={htmlString} ></div>;
}
...
```

События

Имена событий обычно кроссбраузерны и задаются с помощью camelCase. Для примера `change` становится `onChange`, `click` становится `onClick`. Чтобы захватить событие в JSX нужно просто применить свойство к методу компонента:

```
...
handleClick: function (event) {...},
render: function () {
  return <div onClick={this.handleClick}>...</div>
}
...
```

Помните, что React автоматически связывает все методы компонента, вам не нужно вручную определять контекст:

```
...
handleClick: function (event) {...},
render: function () {
  // анти-паттерн - ручное привязывание контекста функции
  // к инстансу компонента не нужно
  return <div onClick={this.handleClick.bind(this)}>...</div>
}
...
```

Более подробно система событий будет рассмотрена в главе о формах.

Специальные атрибуты

Из-за того что JSX трансформируется в чистый JavaScript есть несколько ключевых слов, которые нельзя использовать — `class` и `for`.

Для создания label у формы с атрибутом `for` используйте `htmlFor`:

```
<label htmlFor="for-text" ... >
```

Для создания класса используйте `className`.

```
<div className={classes} ... >
```

Стили

В завершении мы поговорим о inline стилях. В React все стили записаны в camelCase, в соответствие со свойствами DOM, которые используются в JavaScript.

Для определения кастомного атрибута `style` просто определите объект JavaScript с указанием имен свойств в camelCase и нужными CSS-свойствами.

```
var styles = {
  borderColor: "#999",
  borderThickness: "1px"
};
React.renderComponent(<div style={styles}>...</div>, node);
```

React без JSX

Вся JSX разметка трансформируется в простые вызовы JavaScript функций. И, JSX *необязателен* для использования React.

Это все равно работает

Паттерн для создания компонентов очень прост. Это всего лишь функция, вот и все. Названия HTML-тегов определяются с помощью пространства имен `React.DOM`. Создание dib — это всего лишь вызов функции:

```
React.DOM.div();

// <div></div>
```

Дочерние элементы просто вставляются как аргументы, после определения объекта `props`. Мы может срендерить наш divider из примеров выше, просто вставив дочерние элементы во втором и третьем аргументе, при вызове функции:

```
React.DOM.div(
  {className: "divider"},
  "Label Text",
  React.DOM.hr()
)

// <div class="divider">Label Text<hr/></div>
```

Сокращения

Писать `React.DOM.div` крайне неудобно раз от разу. Вы можете смело использовать короткую переменную, например так:

```
var R = React.DOM;
//...
R.div(
  {className: "divider"},
  "Label Text",
  R.hr()
);
```

Или даже так:

```
var div = React.DOM.div;
var hr = React.DOM.hr;
//...
div(
  {className: "divider"},
  "Label Text",
  hr()
);
```

Дальнейшее чтение и ссылки

Если вы до сих пор не понимаете каким образом работает JSX по ссылкам ниже вы можете найти дополнительные материалы, которые помогут вам разобраться что к чему.

Официальная спецификация JSX

В сентябре 2014 года Facebook представили официальную спецификацию JSX, которая включает обоснование использования JSX, а также технические детали синтаксиса.

Вы можете почитать ее вот здесь http://facebook.github.io/jsx/

Эксперименты в браузере

Есть несколько инструментов для экспериментов с JSX. Страница Getting Started документации React содержит ссылки на JSFiddle, где рассматриваются примеры как с JSX, так и без него.

http://facebook.github.io/react/docs/getting-started.html

К тому же у React есть JSX Compiler Service который транслирует JSX в чистый JavaScript прямо в браузере.

http://facebook.github.io/react/jsx-compiler.html

Глава 3. Жизненный цикл компонента React.js

Компоненты React это простейшие автоматы. Их вывод это просто DOM с определенными свойствами и состоянием. С жизненным циклом компоненты свойства и состояние компонентов может изменяться и их DOM отображение меняется тоже. Как было сказано в главе про JSX, компоненты — это просто функции JavaScript.

React дает возможность определять различные моменты жизненного цикла компонентов — их создание, время жизни и демонтаж. Мы рассмотрим их по мере их появления — сначала инициализация, потом процесс жизни и наконец удаление компонента.

Методы жизненного цикла

В React есть относительно немного методов жизненного цикла, но все они необычайно мощные. React дает вам все необходимые методы для контроля свойств и состояния вашего приложения в процессе его жизненного цикла. Давайте взглянем на них в том порядке в котором они вызываются в вашем компоненте.

Когда вы впервые используете класс компонента, методы вызываются в следующем порядке:

- getDefaultProps
- getInitialState
- componentWillMount
- render
- componentDidMount

При всех последующих использованиях компонента вызываются следующие методы (как видите, `getDefaultProps` больше нет в этом списке):

- getInitialState
- componentWillMount
- render
- componentDidMount

Если состояние приложения изменилось и ваш компонент изменен, вызовутся следующие методы именно в этом порядке:

- componentWillRecieveProps
- shouldComponentUpdate
- componentWillUpdate
- render
- componentDidUpdate

И, когда вы закончили использование компонента вызывается метод `componentWillUnmount`, который убирает мусор после использования компонента.

Рассмотрим подробней каждую из этий стадий: инициализация, время жизни и сборка мусора.

Инициализация

Для создания каждого нового компонента и первого рендера есть серия методов, которые вы можете использовать для настройки и подготовки вашего компонента. У каждого из этих методов есть свое назначение, как описано ниже.

getDefaultProps

Метод вызывается единожды для класса компонента. Возвращаемый объект используется для свойств по умолчанию, если они не определены родительским компонентом.

Важно помнить, что любые комплексные значения, такие как объекты и массивы будут доступны всем инстанциям — они не копируются и не клонируются.

getInitialState

Вызывается единожды для каждой инстанции вашего компонента, давая вам возможность инициализировать кастомное состояние каждой инстанции. В отличии от `getDefaultProps` этот метод вызывается каждый раз, когда создается инстанция компонента.

С этого момента вы можете получить доступ к `this.props`.

componentWillMount

Вызывается сразу же после начального рендеринга. Это последний момент воздействовать на состояние компонента, перед вызовом метода `render`.

render

Здесь начинается создание виртуального DOM которое отображает вывод вашего компонента. `Render` это всего лишь обязательный метод для компонента, подчиняющийся определенным правилам. Следующие условия являются обязательным для метода `render`:

- У него есть доступ только к данным `this.props` и `this.state`
- Вы можете вернуть `null`, `false` или любой компонент React.
- Возвращемый компонент должен быть глобальным (вы не можете вернуть массив элементов).
- Он должен быть *чистым*, это значит что он не должен влиять на вывод DOM.

Результат, возвращаемый функцией `render` это не настоящий DOM, но виртуальное представление которое React впоследствии встроит в реальное DOM дерево для определения любых изменений.

componentDidMount

Если рендеринг прошел успешно, и настоящий DOM срендерен, вы можете получить доступ к нему в методе `componentDidMount` через вызов `this.getDOMNode()`.

Вы можете использовать этот прием для доступа к сырому DOM. Для примера, если вам требуется измерить высоту срендеренного элемента, изменять ее по таймеру, или запустить какой-нибудь плагин jQuery.

Помните, что этот метод на работает на сервере

Время жизни

Ваш компонент уже показан пользователю, который может взаимодействовать с ним. Обычно, это означает вызов каких-нибудь обработчиков событыий, таких как click, tap, или key event. Как только пользователь меняет состояние компонента, или приложения, новое состояние спускается по дереву компонентов и у вас есть шанс влиять на это.

componentWillReceiveProps

Свойство компонента может измениться в любое время, через родительский компонент. Когда это происходит вызывается метод `componentWillReceiveProps` и у вас есть возможность назначить новое свойство и обновить состояние объекта.

Для примера, в нашем простом приложении для опросов есть компонент `AnswerRadioInput` который позволяет пользователям менять состояние radio input. Родительский компонент может менять это булево свойство и мы можем отвечать на это, меняя внутренне состояние, основанное на изменении родительского свойства.

```
componentWillReceiveProps: function (nextProps) {
  if(nextProps.checked !== undefined) {
    this.setState({
      checked: nextProps.checked
    });
```

```
  }
}
```