Бесплатная электронная книга

УЧУСЬ React

Free unaffiliated eBook created from
Stack Overflow contributors.

#reactjs

Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: react

It is an unofficial and free React ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official React.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

глава 1: Начало работы с реакцией

замечания

React - это декларативная, основанная на компонентах JavaScript-библиотека, используемая для создания пользовательских интерфейсов.

Для достижения MVC-структуры, такой как функциональность в React, разработчики используют ее в сочетании с вкусом Flux , например Redux .

Версии

Версия	Дата выхода
0.3.0	2013-05-29
0.4.0	2013-07-17
0.5.0	2013-10-16
0.8.0	2013-12-19
0.9.0	2014-02-20
0.10.0	2014-03-21
0.11.0	2014-07-17
0.12.0	2014-10-28
0.13.0	2015-03-10
0.14.0	2015-10-07
15.0.0	2016-04-07
15.1.0	2016-05-20
15.2.0	2016-07-01
15.2.1	2016-07-08
15.3.0	2016-07-29
15.3.1	2016-08-19
15.3.2	2016-09-19

Версия	Дата выхода
15.4.0	2016-11-16
15.4.1	2016-11-23
15.4.2	2017-01-06
15.5.0	2017-04-07
15.6.0	2017-06-13

Examples

Установка или настройка

ReactJS - это библиотека JavaScript, содержащаяся в одном файле `react-<version>.js` который может быть включен в любую HTML-страницу. Люди также обычно устанавливают React DOM library `react-dom-<version>.js` вместе с основным файлом React:

Основное включение

```
<!DOCTYPE html>
<html>
    <head></head>
    <body>
    <script type="text/javascript" src="/path/to/react.js"></script>
    <script type="text/javascript" src="/path/to/react-dom.js"></script>
    <script type="text/javascript">
        // Use react JavaScript code here or in a separate file
    </script>
    </body>
</html>
```

Чтобы получить файлы JavaScript, перейдите на страницу установки официальной документации React.

React также поддерживает синтаксис JSX . JSX - это расширение, созданное Facebook, которое добавляет синтаксис XML к JavaScript. Чтобы использовать JSX, вам нужно включить библиотеку Babel и изменить `<script type="text/javascript">` на `<script type="text/babel">` , чтобы перевести JSX на Javascript-код.

```
<!DOCTYPE html>
<html>
    <head></head>
    <body>
    <script type="text/javascript" src="/path/to/react.js"></script>
    <script type="text/javascript" src="/path/to/react-dom.js"></script>
    <script src="https://npmcdn.com/babel-core@5.8.38/browser.min.js"></script>
    <script type="text/babel">
        // Use react JSX code here or in a separate file
```

```
    </script>
    </body>
</html>
```

Установка через npm

Вы также можете установить React с помощью npm , выполнив следующие действия:

```
npm install --save react react-dom
```

Чтобы использовать React в вашем проекте JavaScript, вы можете сделать следующее:

```
var React = require('react');
var ReactDOM = require('react-dom');
ReactDOM.render(<App />, ...);
```

Установка через пряжу

Facebook выпустил собственный менеджер пакетов с именем Yarn , который также можно использовать для установки React. После установки пряжи вам просто нужно запустить эту команду:

```
yarn add react react-dom
```

Затем вы можете использовать React в своем проекте точно так же, как если бы вы установили React через npm.

Привет, мир

Компонент React может быть определен как класс ES6, который расширяет базовый класс `React.Component` . В своей минимальной форме компонент *должен* определить метод `render` который указывает, как компонент отображает DOM. Метод `render` возвращает React node, который может быть определен с использованием синтаксиса JSX как HTML-подобных тегов. В следующем примере показано, как определить минимальный компонент:

```
import React from 'react'

class HelloWorld extends React.Component {
    render() {
        return <h1>Hello, World!</h1>
    }
}

export default HelloWorld
```

Компонент также может получать `props` . Это свойства, переданные его родителем, чтобы указать некоторые значения, которые компонент не может знать сам по себе; свойство может также содержать функцию, которая может быть вызвана компонентом после возникновения определенных событий - например, кнопка может получить функцию для

своего свойства `onClick` и вызывать ее всякий раз, когда нажимается. При написании компонента его `props` можно получить через объект `props` самого Компонента:

```
import React from 'react'

class Hello extends React.Component {
    render() {
        return <h1>Hello, {this.props.name}!</h1>
    }
}

export default Hello
```

В приведенном выше примере показано, как компонент может отображать произвольную строку, переданную в `name` prop своим родителем. Обратите внимание, что компонент не может изменять получаемые реквизиты.

Компонент может отображаться в любом другом компоненте или непосредственно в DOM, если он является самым верхним компонентом, используя `ReactDOM.render` и предоставляя ему как компонент, так и узел DOM, в котором вы хотите, чтобы дерево React было отображено:

```
import React from 'react'
import ReactDOM from 'react-dom'
import Hello from './Hello'

ReactDOM.render(<Hello name="Billy James" />, document.getElementById('main'))
```

К настоящему времени вы знаете, как создать базовый компонент и принять `props` . Давайте сделаем еще один шаг и представим `state` .

Для демонстрации давайте сделаем наше приложение Hello World, отображаем только первое имя, если дано полное имя.

```
import React from 'react'

class Hello extends React.Component {

    constructor(props){

        //Since we are extending the default constructor,
        //handle default activities first.
        super(props);

        //Extract the first-name from the prop
        let firstName = this.props.name.split(" ")[0];

        //In the constructor, feel free to modify the
        //state property on the current context.
        this.state = {
            name: firstName
        }
```

```
    } //Look maa, no comma required in JSX based class defs!

    render() {
        return <h1>Hello, {this.state.name}!</h1>
    }
}

export default Hello
```

Примечание. Каждый компонент может иметь собственное состояние или принимать состояние родителя в качестве опоры.

Ссылка Codepen на пример.

Привет, мир

Без JSX

Вот базовый пример, который использует основной API React для создания элемента React и React DOM API для визуализации элемента React в браузере.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Hello React!</title>

    <!-- Include the React and ReactDOM libraries -->
    <script src="https://fb.me/react-15.2.1.js"></script>
    <script src="https://fb.me/react-dom-15.2.1.js"></script>

  </head>
  <body>
    <div id="example"></div>

    <script type="text/javascript">

      // create a React element rElement
      var rElement = React.createElement('h1', null, 'Hello, world!');

      // dElement is a DOM container
      var dElement = document.getElementById('example');

      // render the React element in the DOM container
      ReactDOM.render(rElement, dElement);

    </script>

  </body>
</html>
```

С JSX

Вместо создания элемента React из строк можно использовать JSX (расширение Javascript,

созданное Facebook для добавления синтаксиса XML к JavaScript), что позволяет писать

```
var rElement = React.createElement('h1', null, 'Hello, world!');
```

как эквивалент (и более легкий для чтения для кого-то знакомого с HTML)

```
var rElement = <h1>Hello, world!</h1>;
```

Код, содержащий JSX, должен быть заключен в `<script type="text/babel">` . Все внутри этого тега будет преобразовано в обычный Javascript, используя библиотеку Babel (которая должна быть включена в дополнение к библиотекам React).

Итак, наконец, приведенный выше пример:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Hello React!</title>

    <!-- Include the React and ReactDOM libraries -->
    <script src="https://fb.me/react-15.2.1.js"></script>
    <script src="https://fb.me/react-dom-15.2.1.js"></script>
    <!-- Include the Babel library -->
    <script src="https://npmcdn.com/babel-core@5.8.38/browser.min.js"></script>

  </head>
  <body>
   <div id="example"></div>

    <script type="text/babel">

      // create a React element rElement using JSX
      var rElement = <h1>Hello, world!</h1>;

      // dElement is a DOM container
      var dElement = document.getElementById('example');

      // render the React element in the DOM container
      ReactDOM.render(rElement, dElement);

    </script>

  </body>
</html>
```

Что такое ReactJS?

ReactJS - это библиотека с открытым исходным кодом, основанная на компонентах, отвечающая только за **уровень** представления приложения. Он поддерживается Facebook.

ReactJS использует виртуальный механизм DOM для заполнения данных (представлений) в

HTML DOM. Виртуальный DOM работает быстро, владея тем фактом, что он меняет только отдельные элементы DOM, а не перезагружает полную DOM каждый раз

> Приложение React состоит из нескольких **компонентов** , каждый из которых отвечает за вывод небольшого многократно используемого фрагмента HTML. Компоненты могут быть вложены в другие компоненты, чтобы создавать сложные приложения из простых строительных блоков. Компонент может также поддерживать внутреннее состояние - например, компонент TabList может хранить переменную, соответствующую текущей открытой вкладке.

React позволяет писать компоненты, используя язык, специфичный для домена, который называется JSX. JSX позволяет нам писать наши компоненты с помощью HTML, одновременно смешивая события JavaScript. React будет внутренне преобразовывать это в виртуальный DOM и в конечном итоге выводит наш HTML для нас.

Реагируйте « *реагирует* », чтобы быстро и автоматически изменять изменения в ваших компонентах, чтобы повторно перенести компоненты в DOM HTML, используя виртуальную DOM. Виртуальный DOM представляет собой представление в реальном времени DOM. Выполняя большую часть обработки внутри виртуальной DOM, а не непосредственно в DOM браузера, React может действовать быстро и только добавлять, обновлять и удалять компоненты, которые изменились с момента последнего цикла рендеринга.

Hello World с функциями безгражданства

Компоненты без гражданства получают свою философию от функционального программирования. Это означает, что: функция возвращает все время одинаково точно на то, что ему дано.

Например:

```
const statelessSum = (a, b) => a + b;

let a = 0;
const statefulSum = () => a++;
```

Как видно из приведенного выше примера, statelessSum всегда возвращает те же значения, что и a и b. Однако функция statefulSum не будет возвращать те же значения, которые даны даже без параметров. Такое поведение функции также называется *побочным эффектом* . Так как компонент влияет на что-то за пределами.

Поэтому рекомендуется чаще использовать компоненты без гражданства, поскольку они *свободны от побочных эффектов* и всегда будут создавать одинаковое поведение. Это то, что вы хотите использовать в своих приложениях, потому что колебательное состояние - наихудший сценарий для поддерживаемой программы.

Самый основной тип реагирующего компонента - один без состояния. Реагировать на компоненты, которые являются чистыми функциями своих реквизитов и не требуют какого-либо внутреннего управления состоянием, могут быть написаны как простые функции JavaScript. Они говорят, что `Stateless Functional Components` , поскольку они являются функцией только `props` , не имея `state` , чтобы следить.

Вот простой пример, иллюстрирующий концепцию `Stateless Functional Component` :

```
// In HTML
<div id="element"></div>

// In React
const MyComponent = props => {
    return <h1>Hello, {props.name}!</h1>;
};

ReactDOM.render(<MyComponent name="Arun" />, element);
// Will render <h1>Hello, Arun!</h1>
```

Обратите внимание, что все, что делает этот компонент, представляет собой элемент `h1` содержащий `name` prop. Этот компонент не отслеживает состояние. Вот пример ES6:

```
import React from 'react'

const HelloWorld = props => (
    <h1>Hello, {props.name}!</h1>
)

HelloWorld.propTypes = {
    name: React.PropTypes.string.isRequired
}

export default HelloWorld
```

Поскольку эти компоненты не требуют экземпляра поддержки для управления состоянием, React имеет больше возможностей для оптимизации. Реализация чиста, но пока не реализована такая оптимизация для компонентов без состояния .

Создать приложение React

create-react-app - это генератор шаблонов приложения React, созданный Facebook. Он обеспечивает среду разработки, настроенную для простоты использования с минимальной настройкой, в том числе:

- ES6 и трансляция JSX
- Сервер Dev с горячей перезагрузкой модуля
- Листинг кода
- Автоматическое префикс CSS
- Сценарий сборки с использованием JS, CSS и компоновки изображений и исходных карт

- Схема тестирования Jest

Монтаж

Во-первых, установите приложение create-response-app глобально с менеджером пакетов узлов (npm).

```
npm install -g create-react-app
```

Затем запустите генератор в выбранном вами каталоге.

```
create-react-app my-app
```

Перейдите во вновь созданный каталог и запустите стартовый скрипт.

```
cd my-app/
npm start
```

конфигурация

По умолчанию приложение create-react-приложение намеренно не настраивается. Если требуется использовать нестандартное использование, например, для использования скомпилированного языка CSS, такого как Sass, тогда можно использовать команду eject.

```
npm run eject
```

Это позволяет редактировать все файлы конфигурации. NB это необратимый процесс.

альтернативы

Альтернативные контрольные плиты включают:

- анклав
- NWB
- движение
- rackt-кли
- Будо
- РБГ
- QUIK
- Sagui
- птица рух

Приложение Build React

Чтобы готовить приложение для производства, выполните следующую команду

```
npm run build
```

Абсолютные основы создания многоразовых компонентов

Компоненты и реквизит

Поскольку React затрагивает себя только с точки зрения приложения, основной частью разработки в React будет создание компонентов. Компонент представляет часть представления вашего приложения. «Опоры» - это просто атрибуты, используемые на узле JSX (например, `<SomeComponent someProp="some prop's value" />`) и являются основным способом взаимодействия нашего приложения с нашими компонентами. В фрагменте выше, внутри SomeComponent, у нас будет доступ к `this.props` , значением которого будет объект `{someProp: "some prop's value"}` .

Может быть полезно подумать о компонентах React как о простых функциях - они принимают входные данные в виде «реквизита» и производят вывод в виде разметки. Многие простые компоненты делают это еще дальше, делая себя «Чистыми функциями», что означает, что они не выдают побочных эффектов и являются идемпотентными (с учетом набора входов компонент всегда будет воспроизводить один и тот же результат). Эта цель может быть формально введена в действие, фактически создавая компоненты как функции, а не «классы». Существует три способа создания компонента React:

- **Функциональные («Безстоящие») компоненты**

```
const FirstComponent = props => (
    <div>{props.content}</div>
);
```

- **React.createClass ()**

```
const SecondComponent = React.createClass({
    render: function () {
        return (
            <div>{this.props.content}</div>
        );
    }
});
```

- **Классы ES2015**

```
class ThirdComponent extends React.Component {
    render() {
        return (
            <div>{this.props.content}</div>
        );
```

```
    }
}
```

Эти компоненты используются точно так же:

```
const ParentComponent = function (props) {
    const someText = "FooBar";
    return (
        <FirstComponent content={someText} />
        <SecondComponent content={someText} />
        <ThirdComponent content={someText} />
    );
}
```

Вышеприведенные примеры приведут к одинаковой разметке.

Функциональные компоненты не могут иметь «состояние» внутри них. Поэтому, если ваш компонент должен иметь состояние, то перейдите для компонентов на основе классов. Дополнительную информацию см. В разделе « Создание компонентов» .

В качестве последней заметки, реакция реквизита неизменна после того, как они были переданы, то есть они не могут быть изменены изнутри компонента. Если родительский компонент компонента изменяет значение prop, React обрабатывает замену старых реквизитов на новый, компонент будет повторно выполнять себя с использованием новых значений.

См. « Мышление в реакциях и многоразовых компонентах» для более глубоких погружений в отношение реквизита к компонентам.

Прочитайте Начало работы с реакцией онлайн: https://riptutorial.com/ru/reactjs/topic/797/начало-работы-с-реакцией

глава 2: JSX

замечания

JSX - это **шаг препроцессора,** который добавляет синтаксис XML к JavaScript. Вы можете определенно использовать React без JSX, но JSX делает React намного более элегантным.

Подобно XML, теги JSX имеют имя, атрибуты и дочерние элементы. Если значение атрибута заключено в кавычки, значение представляет собой строку. В противном случае оберните значение в фигурные скобки, а значение - вложенное выражение JavaScript.

По сути, JSX просто предоставляет синтаксический сахар для функции `React.createElement(component, props, ...children)` .

Итак, следующий код JSX:

```
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}

ReactDOM.render(<HelloMessage name="Kalo" />, mountNode);
```

Скомпилируется до следующего кода JavaScript:

```
class HelloMessage extends React.Component {
  render() {
    return React.createElement(
      "div",
      null,
      "Hello ",
      this.props.name
    );
  }
}

ReactDOM.render(React.createElement(HelloMessage, { name: "Kalo" }), mountNode);
```

В заключение отметим, что **следующая строка в JSX не является ни строкой, ни HTML** :

```
const element = <h1>Hello, world!</h1>;
```

Он называется JSX, и это **расширение синтаксиса JavaScript** . JSX может напомнить вам о языке шаблонов, но он поставляется с полной мощью JavaScript.

Команда React говорит в своих документах, что они рекомендуют использовать ее для описания того, как должен выглядеть пользовательский интерфейс.

Examples

Подставки в JSX

Существует несколько способов указать реквизиты в JSX.

Выражения JavaScript

Вы можете передать **любое выражение JavaScript** в качестве опоры, окружив его `{}` . Например, в этом JSX:

```
<MyComponent count={1 + 2 + 3 + 4} />
```

Внутри `MyComponent` значение `props.count` будет равно `10` , потому что вычисляется выражение `1 + 2 + 3 + 4` .

Если утверждения и циклы не являются выражениями в JavaScript, поэтому они не могут использоваться в JSX напрямую.

Строковые литералы

Конечно, вы можете просто передать любой `string literal` в качестве `prop` . Эти два выражения JSX эквивалентны:

```
<MyComponent message="hello world" />

<MyComponent message={'hello world'} />
```

Когда вы передаете строковый литерал, его значение не привязано к HTML. Таким образом, эти два выражения JSX эквивалентны:

```
<MyComponent message="&lt;3" />

<MyComponent message={'<3'} />
```

Такое поведение обычно не имеет значения. Это только упоминается здесь для полноты.

Значение по умолчанию для реквизита

Если вы не передадите значение для пропеллера, **оно по умолчанию** равно `true` . Эти два

выражения JSX эквивалентны:

```
<MyTextBox autocomplete />

<MyTextBox autocomplete={true} />
```

Тем не менее, команда React говорит в своих документах, **используя этот подход, не рекомендуется** , потому что его можно путать с сокращением объекта ES6 `{foo}` который меньше для `{foo: foo}` а не `{foo: true}` . Они говорят, что это поведение существует именно так, чтобы оно соответствовало поведению HTML.

Атрибуты распространения

Если у вас уже есть реквизит в качестве объекта, и вы хотите передать его в JSX, вы можете использовать `...` в качестве оператора распространения для передачи всего объекта реквизита. Эти два компонента эквивалентны:

```
function Case1() {
  return <Greeting firstName="Kaloyab" lastName="Kosev" />;
}

function Case2() {
  const person = {firstName: 'Kaloyan', lastName: 'Kosev'};
  return <Greeting {...person} />;
}
```

Дети в JSX

В выражениях JSX, которые содержат как открытый тег, так и закрывающий тег, контент между этими тегами передается как специальный prop: `props.children` . Есть несколько способов передачи детей:

Строковые литералы

Вы можете поместить строку между открывающими и закрывающими тегами, а `props.children` будет только той строкой. Это полезно для многих встроенных элементов HTML. Например:

```
<MyComponent>
    <h1>Hello world!</h1>
</MyComponent>
```

Это действительный JSX, а `props.children` в MyComponent будет просто `<h1>Hello world!</h1>` .

Обратите внимание, что **HTML не привязан** , поэтому вы можете вообще писать JSX так же, как и писать HTML.

Заметим, что в этом случае JSX:

- удаляет пробелы в начале и в конце строки;
- удаляет пустые строки;
- удаляются новые строки, смежные с тегами;
- новые строки, которые встречаются в середине строковых литералов, сводятся в одно пространство.

Дети JSX

Вы можете предоставить больше JSX-элементов в качестве детей. Это полезно для отображения вложенных компонентов:

```
<MyContainer>
  <MyFirstComponent />
  <MySecondComponent />
</MyContainer>
```

Вы можете **смешивать разные типы детей, поэтому вы можете использовать строковые литералы вместе с детьми JSX** . Это еще один способ, в котором JSX похож на HTML, так что это как действительный JSX, так и действительный HTML:

```
<div>
  <h2>Here is a list</h2>
  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
  </ul>
</div>
```

Обратите внимание, что компонент React **не может возвращать несколько элементов React, но одно выражение JSX может иметь несколько дочерних элементов** . Поэтому, если вы хотите, чтобы компонент отображал несколько вещей, вы можете обернуть их в `div` как пример выше.

Выражения JavaScript

Вы можете передать любое выражение JavaScript в виде дочерних элементов, заключая его в `{}` . Например, эти выражения эквивалентны:

```
<MyComponent>foo</MyComponent>

<MyComponent>{'foo'}</MyComponent>
```

Это часто полезно для отображения списка выражений JSX произвольной длины. Например, это отображает список HTML:

```
const Item = ({ message }) => (
  <li>{ message }</li>
);

const TodoList = () => {
  const todos = ['finish doc', 'submit review', 'wait stackoverflow review'];
  return (
    <ul>
      { todos.map(message => (<Item key={message} message={message} />)) }
    </ul>
  );
};
```

Обратите внимание, что выражения JavaScript могут быть смешаны с другими типами детей.

Функции как дети

Обычно выражения JavaScript, вставленные в JSX, будут оценивать строку, элемент React или список этих вещей. Тем не менее, `props.children` работает так же, как и любая другая поддержка в том, что он может передавать любые данные, а не только те виды, которые React знает, как визуализировать. Например, если у вас есть пользовательский компонент, вы можете заставить его выполнить обратный вызов как `props.children` :

```
const ListOfTenThings = () => (
  <Repeat numTimes={10}>
    {(index) => <div key={index}>This is item {index} in the list</div>}
  </Repeat>
);

// Calls the children callback numTimes to produce a repeated component
const Repeat = ({ numTimes, children }) => {
  let items = [];
  for (let i = 0; i < numTimes; i++) {
    items.push(children(i));
  }
  return <div>{items}</div>;
};
```

Дети, переданные в пользовательский компонент, могут быть чем угодно, если этот компонент преобразует их во что-то, что Реакт может понять перед рендерингом. Это использование не является обычным явлением, но оно работает, если вы хотите расширить

возможности JSX.

Игнорируемые значения

Обратите внимание, что `false` , `null` , `undefined` и `true` являются действительными дочерними. Но они просто не делают. Эти выражения JSX будут отображать одно и то же:

```
<MyComponent />

<MyComponent></MyComponent>

<MyComponent>{false}</MyComponent>

<MyComponent>{null}</MyComponent>

<MyComponent>{true}</MyComponent>
```

Это чрезвычайно полезно для условного рендеринга элементов React. Этот JSX отображает только if, если `showHeader` имеет значение true:

```
<div>
  {showHeader && <Header />}
  <Content />
</div>
```

Одним из важных предостережений является то, что некоторые «фальшивые» значения, такие как `0` , все еще выдаются React. Например, этот код не будет вести себя так, как вы могли ожидать, потому что `0` будет напечатан, если `props.messages` - пустой массив:

```
<div>
  {props.messages.length &&
    <MessageList messages={props.messages} />
  }
</div>
```

Один из подходов к исправлению заключается в том, чтобы убедиться, что выражение перед `&&` всегда является логическим:

```
<div>
  {props.messages.length > 0 &&
    <MessageList messages={props.messages} />
  }
</div>
```

Наконец, не забывайте, что если вы хотите, чтобы в выводе было указано значение `false` , `true` , `null` или `undefined` , вы должны сначала преобразовать его в строку:

```
<div>
```

```
  My JavaScript variable is {String(myVariable)}.
</div>
```

Прочитайте JSX онлайн: https://riptutorial.com/ru/reactjs/topic/8027/jsx

глава 3: React.createClass vs extends React.Component

Синтаксис

- Случай 1: React.createClass ({})
- Случай 2: класс MyComponent расширяет React.Component {}

замечания

`React.createClass` устарел в версии 15.5 и, как ожидается, будет удален в v16 . Для тех, которые все еще требуют этого, есть пакет замены замены . Примеры, использующие его, должны быть обновлены.

Examples

Создать реактивный компонент

Давайте рассмотрим различия в синтаксисе, сравнив два примера кода.

React.createClass (устаревший)

Здесь у нас есть `const` с классом класса React, с последующей функцией `render` чтобы выполнить типичное определение базового компонента.

```
import React from 'react';

const MyComponent = React.createClass({
  render() {
    return (
      <div></div>
    );
  }
});

export default MyComponent;
```

React.Component

Давайте рассмотрим вышеописанное определение React.createClass и преобразуем его в класс ES6.

```
import React from 'react';

class MyComponent extends React.Component {
  render() {
    return (
      <div></div>
    );
  }
}

export default MyComponent;
```

В этом примере мы теперь используем классы ES6. Для изменений React теперь мы создаем класс **MyComponent** и распространяем его из React.Component, а не напрямую на React.createClass. Таким образом, мы используем меньше шаблонов React и больше JavaScript.

PS: Обычно это будет использоваться с чем-то вроде Babel для компиляции ES6 на ES5 для работы в других браузерах.

Объявлять реквизиты и пропеллеры по умолчанию

Существуют важные изменения в том, как мы используем и объявляем реквизиты по умолчанию и их типы.

React.createClass

В этой версии свойство `propTypes` представляет собой объект, в котором мы можем объявить тип для каждой опоры. Свойство `getDefaultProps` - это функция, которая возвращает объект для создания исходных реквизитов.

```
import React from 'react';

const MyComponent = React.createClass({
  propTypes: {
    name: React.PropTypes.string,
    position: React.PropTypes.number
  },
  getDefaultProps() {
    return {
      name: 'Home',
      position: 1
    };
  },
  render() {
    return (
      <div></div>
    );
  }
});

export default MyComponent;
```

React.Component

Эта версия использует `propTypes` как свойство в самом классе **MyComponent** вместо свойства как части `createClass` определения `createClass` .

Теперь `getDefaultProps` изменился на просто свойство Object в классе, называемом defaultProps, поскольку он больше не является функцией «get», а просто Object. Это позволяет избежать повторного шаблона React, это простой JavaScript.

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div></div>
    );
  }
}
MyComponent.propTypes = {
  name: React.PropTypes.string,
  position: React.PropTypes.number
};
MyComponent.defaultProps = {
  name: 'Home',
  position: 1
};

export default MyComponent;
```

Кроме того, существует еще один синтаксис для `propTypes` и `defaultProps` . Это ярлык, если в вашей сборке включены инициализаторы свойств ES7:

```
import React from 'react';

class MyComponent extends React.Component {
  static propTypes = {
    name: React.PropTypes.string,
    position: React.PropTypes.number
  };
  static defaultProps = {
    name: 'Home',
    position: 1
  };
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div></div>
    );
  }
}
```

```
export default MyComponent;
```

Установить начальное состояние

Существуют изменения в том, как мы устанавливаем начальные состояния.

React.createClass

У нас есть функция `getInitialState` , которая просто возвращает объект начальных состояний.

```
import React from 'react';

const MyComponent = React.createClass({
  getInitialState () {
    return {
      activePage: 1
    };
  },
  render() {
    return (
      <div></div>
    );
  }
});

export default MyComponent;
```

React.Component

В этой версии мы объявляем все состояние как простое **свойство инициализации в конструкторе** , вместо использования функции `getInitialState` . Он чувствует себя менее « React API», потому что это простой JavaScript.

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      activePage: 1
    };
  }
  render() {
    return (
      <div></div>
    );
  }
}
```

```
export default MyComponent;
```

Примеси

Мы можем использовать `mixins` только с методом React.createClass.

React.createClass

В этой версии мы можем добавлять `mixins` к компонентам, используя свойство mixins, которое принимает массив доступных миксинов. Затем они расширяют класс компонента.

```
import React from 'react';

var MyMixin = {
  doSomething() {

  }
};
const MyComponent = React.createClass({
  mixins: [MyMixin],
  handleClick() {
    this.doSomething(); // invoke mixin's method
  },
  render() {
    return (
      <button onClick={this.handleClick}>Do Something</button>
    );
  }
});

export default MyComponent;
```

React.Component

Реакция миксинов не поддерживается при использовании компонентов React, написанных на ES6. Более того, у них не будет поддержки классов ES6 в React. Причина в том, что они считаются вредными .

«этот» контекст

Использование React.createClass автоматически привяжет `this` контекст (значения) правильно, но это не относится к использованию классов ES6.

React.createClass

Обратите внимание на `onClick` декларацию с `this.handleClick` методом связанного. Когда этот метод будет вызван, React применит правильный контекст выполнения к `handleClick` .

```
import React from 'react';

const MyComponent = React.createClass({
  handleClick() {
    console.log(this); // the React Component instance
  },
  render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
});

export default MyComponent;
```

React.Component

С классами ES6 `this` по умолчанию равно `null` , свойства этого класса автоматически не привязываются к экземпляру класса React (компонент).

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  handleClick() {
    console.log(this); // null
  }
  render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
}

export default MyComponent;
```

Есть несколько способов связать право `this` контекста.

Случай 1: Связать встроенный:

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  handleClick() {
```

```
    console.log(this); // the React Component instance
  }
  render() {
    return (
      <div onClick={this.handleClick.bind(this)}></div>
    );
  }
}

export default MyComponent;
```

Случай 2: привязка к конструктору класса

Другой подход заключается в изменении контекста `this.handleClick` внутри `constructor` . Таким образом мы избегаем повторного повторения. Многие считают, что это лучший подход, который позволяет вообще не прикасаться к JSX:

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    console.log(this); // the React Component instance
  }
  render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
}

export default MyComponent;
```

Случай 3: использование анонимной функции ES6

Вы также можете использовать анонимную функцию ES6, не связывая явно:

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  handleClick = () => {
    console.log(this); // the React Component instance
  }
  render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
```

```
}

export default MyComponent;
```

ES6 / Реагировать «это» ключевое слово с помощью ajax для получения данных с сервера

```
import React from 'react';

class SearchEs6 extends React.Component{
    constructor(props) {
        super(props);
        this.state = {
            searchResults: []
        };
    }

    showResults(response){
        this.setState({
            searchResults: response.results
        })
    }

    search(url){
        $.ajax({
            type: "GET",
            dataType: 'jsonp',
            url: url,
            success: (data) => {
                this.showResults(data);
            },
            error: (xhr, status, err) => {
                console.error(url, status, err.toString());
            }
        });
    }

    render() {
        return (
            <div>
                <SearchBox search={this.search.bind(this)} />
                <Results searchResults={this.state.searchResults} />
            </div>
        );
    }
}
```

Прочитайте React.createClass vs extends React.Component онлайн:
https://riptutorial.com/ru/reactjs/topic/6371/react-createclass-vs-extends-react-component

глава 4: Введение в серверную визуализацию

Examples

Рендеринг компонентов

Существует два варианта рендеринга компонентов на сервере: `renderToString` и `renderToStaticMarkup` .

renderToString

Это сделает компоненты React для HTML на сервере. Эта функция будет также добавить `data-react-` свойство HTML элементов так Реагировать на клиенте не будет снова элементы визуализации.

```
import { renderToString } from "react-dom/server";
renderToString(<App />);
```

renderToStaticMarkup

Это позволит отображать компоненты React в HTML, но без свойств `data-react-` не рекомендуется использовать компоненты, которые будут отображаться на клиенте, потому что компоненты будут повторно выполняться.

```
import { renderToStaticMarkup } from "react-dom/server";
renderToStaticMarkup(<App />);
```

Прочитайте Введение в серверную визуализацию онлайн:
https://riptutorial.com/ru/reactjs/topic/7478/введение-в-серверную-визуализацию

глава 5: Использование ReactJS в потоке Flux

Вступление

Очень удобно использовать подход Flux, когда ваше приложение с ReactJS на frontend планируется увеличить, из-за ограниченных структур и небольшого количества нового кода, чтобы сделать изменения состояния во время выполнения более легкими.

замечания

Flux - это архитектура приложений, используемая Facebook для создания клиентских веб-приложений. Он дополняет составные компоненты представления React, используя однонаправленный поток данных. Это скорее шаблон, чем формальная структура, и вы можете сразу начать использовать Flux без большого количества нового кода.

Приложения Flux имеют три основные части: *диспетчер* , *магазины* и *виды* (компоненты React). Их не следует путать с Model-View-Controller. Контроллеры существуют в приложении Flux, но они представляют собой представления контроллера - представления, которые часто встречаются в верхней части иерархии, которые извлекают данные из хранилищ и передают эти данные своим детям. Кроме того, создатели действий - вспомогательные методы диспетчера - используются для поддержки семантического API, который описывает все изменения, которые возможны в приложении. Полезно подумать о них как о четвертой части цикла обновления потока.

Flux избегает MVC в пользу однонаправленного потока данных. Когда пользователь взаимодействует с представлением React, представление передает действие через центральный диспетчер в различные магазины, в которых хранятся данные и бизнес-логика приложения, которые обновляют все просмотренные виды. Это особенно хорошо работает с декларативным стилем программирования React, который позволяет магазину отправлять обновления без указания способа перехода между состояниями.

Examples

Поток данных

> Это общий обзор .

Образец потока предполагает использование однонаправленного потока данных.

1. **Действие** - простой объект, описывающий `type` действия и другие входные данные.

2. **Диспетчер** - контроллер приемника с одним действием и обратный вызов. Представьте, что это центральный центр вашего приложения.

3. **Store** - содержит состояние приложения и логику. Он регистрирует обратный вызов в диспетчере и испускает событие для просмотра, когда произошло изменение уровня данных.

4. **View** - React компонент, который получает событие изменения и данные из хранилища. Это вызывает повторный рендеринг, когда что-то изменяется.

    > По потоку потока данных представления также могут **создавать действия** и передавать их диспетчеру для взаимодействия с пользователем.

Отменено

Чтобы сделать его более ясным, мы можем начать с самого конца.

- Различные компоненты React (*views*) получают данные из разных магазинов о внесенных изменениях.

    > Немногие компоненты можно назвать **видами контроллеров** , потому что они предоставляют код клея для получения данных из магазинов и передачи данных по цепочке их потомков. Представления контроллера представляют собой любой важный раздел страницы.

- *Магазины* можно отметить как обратные вызовы, которые сравнивают тип действия и другие входные данные для бизнес-логики вашего приложения.

- *Диспетчер* является общим приемом приемника и callbacks контейнера.

- *Действия* - это не что иное, как простые объекты с требуемым свойством `type` .

    > Раньше вы хотели бы использовать константы для типов действий и вспомогательных методов (называемых **создателями действий**).

Прочитайте Использование ReactJS в потоке Flux онлайн:
https://riptutorial.com/ru/reactjs/topic/8158/использование-reactjs-в-потоке-flux

глава 6: Использование ReactJS с jQuery

Examples

ReactJS с jQuery

Во-первых, вам нужно импортировать библиотеку jquery. Нам также нужно импортировать findDOmNode, поскольку мы будем манипулировать dom. И, очевидно, мы также импортируем React.

```
import React from 'react';
import { findDOMNode } from 'react-dom';
import $ from 'jquery';
```

Мы устанавливаем функцию стрелки «handleToggle», которая срабатывает при щелчке значка. Мы просто показываем и скрываем div с ссылкой, обозначающей «переключить» onClick над значком.

```
handleToggle = () => {
    const el = findDOMNode(this.refs.toggle);
    $(el).slideToggle();
};
```

Давайте теперь установим ссылку «toggle»,

```
<ul className="profile-info additional-profile-info-list" ref="toggle">
  <li>
    <span className="info-email">Office Email</span>   me@shuvohabib.com
  </li>
</ul>
```

Элемент div, в котором мы будем запускать «handleToggle» onClick.

```
 <div className="ellipsis-click" onClick={this.handleToggle}>
   <i className="fa-ellipsis-h"/>
 </div>
```

Давайте рассмотрим полный код ниже, как он выглядит.

```
import React from 'react';
import { findDOMNode } from 'react-dom';
import $ from 'jquery';

export default class FullDesc extends React.Component {
    constructor() {
        super();
    }
```

```
    handleToggle = () => {
        const el = findDOMNode(this.refs.toggle);
        $(el).slideToggle();
    };

    render() {
        return (
            <div className="long-desc">
                <ul className="profile-info">
                    <li>
                        <span className="info-title">User Name : </span> Shuvo Habib
                    </li>
                </ul>

                <ul className="profile-info additional-profile-info-list" ref="toggle">
                    <li>
                        <span className="info-email">Office Email</span> me@shuvohabib.com
                    </li>
                </ul>

                <div className="ellipsis-click" onClick={this.handleToggle}>
                    <i className="fa-ellipsis-h"/>
                </div>
            </div>
        );
    }
}
```

Мы сделали! Именно так мы можем использовать **jQuery в** компоненте **React** .

Прочитайте Использование ReactJS с jQuery онлайн:
https://riptutorial.com/ru/reactjs/topic/6009/использование-reactjs-c-jquery

глава 7: Использование ReactJS с машинописным текстом

Examples

Компонент ReactJS, написанный на машинописном машинописном документе

Фактически вы можете использовать компоненты ReactJS в Typcript, как в примере на facebook. Просто замените расширение файла «jsx» на «tsx»:

```
//helloMessage.tsx:
var HelloMessage = React.createClass({
  render: function() {
    return <div>Hello {this.props.name}</div>;
  }
});
ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

Но для того, чтобы в полной мере использовать главную функцию Typcript (проверка статического типа), нужно выполнить пару вещей:

1) конвертировать пример React.createClass в ES6 Класс:

```
//helloMessage.tsx:
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}
ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

2) затем добавить интерфейсы реквизита и состояния:

```
interface IHelloMessageProps {
    name:string;
}

interface IHelloMessageState {
  //empty in our case
}

class HelloMessage extends React.Component<IHelloMessageProps, IHelloMessageState> {
  constructor(){
    super();
  }
  render() {
    return <div>Hello {this.props.name}</div>;
  }
```

```
}
ReactDOM.render(<HelloMessage name="Sebastian" />, mountNode);
```

Теперь TypScript отобразит ошибку, если программист забудет передать реквизиты. Или если они добавили реквизиты, которые не определены в интерфейсе.

Агрегаты без учета состояния в машинописном тексте

Реагировать на компоненты, которые являются чистыми функциями своих реквизитов и не требуют какого-либо внутреннего состояния, могут быть записаны как функции JavaScript вместо использования стандартного синтаксиса класса, а именно:

```
import React from 'react'

const HelloWorld = (props) => (
    <h1>Hello, {props.name}!</h1>
);
```

То же самое можно сделать в TypScript с использованием класса `React.SFC` :

```
import * as React from 'react';

class GreeterProps {
   name: string
}

const Greeter : React.SFC<GreeterProps> = props =>
    <h1>Hello, {props.name}!</h1>;
```

Обратите внимание, что имя `React.SFC` является псевдонимом для `React.StatelessComponent` Таким образом, либо можно использовать.

Установка и настройка

Чтобы использовать машинопись с реакцией в проекте узла, вы должны сначала создать каталог проекта, инициализированный с помощью npm. Чтобы инициализировать каталог с помощью `npm init`

Установка через npm или пряжу

Вы можете установить React с помощью npm , выполнив следующие действия:

```
npm install --save react react-dom
```

Facebook выпустил собственный менеджер пакетов с именем Yarn , который также можно использовать для установки React. После установки пряжи вам просто нужно запустить эту команду:

```
yarn add react react-dom
```

Затем вы можете использовать React в своем проекте точно так же, как если бы вы установили React через npm.

Установка определений типа реакции в TypScript 2.0+

Чтобы скомпилировать свой код с помощью машинописных файлов, добавьте / установите типы определения файлов с использованием npm или пряжи.

```
npm install --save-dev @types/react @types/react-dom
```

или, используя пряжу

```
yarn add --dev @types/react @types/react-dom
```

Установка определений типа реакции в более старых версиях Typcript

Вы должны использовать отдельный пакет под названием tsd

```
tsd install react react-dom --save
```

Добавление или изменение конфигурации Виджета

Чтобы использовать JSX , язык, смешивающий javascript с html / xml, вам нужно изменить конфигурацию компилятора typescript. В файле конфигурации машинописного файла проекта (обычно называемом `tsconfig.json`) вам нужно добавить параметр JSX как:

```
"compilerOptions": {
    "jsx": "react"
},
```

Этот параметр компилятора в основном сообщает компилятору typescript, чтобы перевести теги JSX в код на вызовы функций javascript.

Чтобы избежать компиляции скриптов, конвертирующих JSX в обычные вызовы функций javascript, используйте

```
"compilerOptions": {
    "jsx": "preserve"
},
```

Безстоящие и не имеющие отношения компоненты

Простейший компонент реакции без состояния и свойств не может быть записан как:

```
import * as React from 'react';
```

```
const Greeter = () => <span>Hello, World!</span>
```

Однако этот компонент не может получить доступ к `this.props` поскольку машинописный текст не может определить, является ли он реактивным компонентом. Чтобы получить доступ к своим реквизитам, используйте:

```
import * as React from 'react';

const Greeter: React.SFC<{}> = props => () => <span>Hello, World!</span>
```

Даже если компонент не имеет явно определенных свойств, он теперь может обращаться к **`props.children`** поскольку у всех компонентов есть дети.

Другим аналогичным хорошим использованием компонентов без учета состояния и без свойств является простое шаблонирование страниц. Ниже приведен пример простой простой компонент `Page` , если в проекте есть гипотетические компоненты `Container` , `NavTop` и `NavBottom` :

```
import * as React from 'react';

const Page: React.SFC<{}> = props => () =>
    <Container>
        <NavTop />
        {props.children}
        <NavBottom />
    </Container>

const LoginPage: React.SFC<{}> = props => () =>
    <Page>
        Login Pass: <input type="password" />
    </Page>
```

В этом примере компонент `Page` позже может использоваться любой другой фактической страницей в качестве базового шаблона.

Прочитайте Использование ReactJS с машинописным текстом онлайн:
https://riptutorial.com/ru/reactjs/topic/1419/использование-reactjs-с-машинописным-текстом

глава 8: Использование реакции с потоком

Вступление

Как использовать проверку типа потока для проверки типов в компонентах React.

замечания

Поток | реагировать

Examples

Использование потока для проверки типов поддержки безстоящих функциональных компонентов

```
type Props = {
  posts: Array<Article>,
  dispatch: Function,
  children: ReactElement
}

const AppContainer =
  ({ posts, dispatch, children }: Props) => (
    <div className="main-app">
      <Header {...{ posts, dispatch }} />
      {children}
    </div>
  )
```

Использование потока для проверки типов опоры

```
import React, { Component } from 'react';

type Props = {
  posts: Array<Article>,
  dispatch: Function,
  children: ReactElement
}

class Posts extends Component {
  props: Props;

  render () {
    // rest of the code goes here
  }
}
```

Прочитайте Использование реакции с потоком онлайн:

https://riptutorial.com/ru/reactjs/topic/7918/использование-реакции-с-потоком

глава 9: Как и почему использовать ключи в React

Вступление

Всякий раз, когда вы создаете список компонентов React, каждый компонент должен иметь `key` атрибут. Ключ может быть любым значением, но он должен быть уникальным для этого списка.

Когда React должен отображать изменения в списке элементов, React просто выполняет итерацию по обоим спискам детей одновременно и генерирует мутацию всякий раз, когда есть разница. Если для детей нет ключей, React сканирует каждого ребенка. В противном случае React сравнивает ключи, чтобы узнать, какие из них были добавлены или удалены из списка

замечания

Для получения дополнительной информации перейдите по этой ссылке, чтобы прочитать, как использовать ключи: https://facebook.github.io/react/docs/lists-and-keys.html

Посетите эту ссылку, чтобы узнать, почему рекомендуется использовать ключи: https://facebook.github.io/react/docs/reconciliation.html#recursing-on-children

Examples

Основной пример

Для неактивного компонента:

```
function SomeComponent(props){

    const ITEMS = ['cat', 'dog', 'rat']
    function getItemsList(){
        return ITEMS.map(item => <li key={item}>{item}</i>);
    }

    return (
        <ul>
            {getItemsList()}
        </ul>
    );
}
```

В этом примере вышеуказанный компонент разрешает:

```
<ul>
    <li key='cat'>cat</li>
    <li key='dog'>dog</li>
    <li key='rat'>rat</li>
<ul>
```

Прочитайте Как и почему использовать ключи в React онлайн:
https://riptutorial.com/ru/reactjs/topic/9665/как-и-почему-использовать-ключи-в-react

глава 10: Как настроить базовый веб-пакет, реагировать и загружать среду

замечания

Этот конвейер сборки не совсем то, что вы бы назвали «готовым продуктом», но это дает вам твердый старт для вас, чтобы добавить к нему то, что вам нужно, чтобы получить опыт разработки, который вы ищете. Подход, который некоторые люди берут (в том числе и я сам по себе), - это взять полностью построенный трубопровод Yeoman.io или где-то в другом месте, а затем снять с себя то, что им не нужно, пока оно не устраивает стиль. В этом нет ничего плохого, но, возможно, с приведенным выше примером вы могли бы выбрать противоположный подход и сформироваться из голых костей.

Некоторые вещи, которые вы могли бы добавить, - это такие вещи, как структура тестирования и статистика покрытия, такие как карма с моккой или жасмином. Переливание с помощью ESLint. Горячая замена модуля в webpack-dev-сервере, чтобы вы могли получить опыт разработки Ctrl + S, F5. Также текущий конвейер работает только в режиме dev, поэтому задача создания сборки будет хорошей.

Gotchas!

Обратите внимание, что в свойстве контекста `webpack.config.js` мы использовали модуль пути узла, чтобы определить наш путь, а не просто конкатенировать `__dirname` в строке `/src` потому что окна ненавидят перекосы . Поэтому, чтобы сделать решение более совместимым с перекрестными платформами, используйте инструмент рычагов, чтобы помочь нам.

Объяснение свойств webpack.config.js

контекст

Это путь к файлу, для которого webpack будет использовать его как корневой путь для решения относительных путей файлов. Поэтому в index.jsx, где мы используем `require('./index.html')` эта точка фактически разрешает директорию `src/` потому что мы определили ее как таковую в этом свойстве.

запись

Где webpack выглядит сначала, чтобы начать связывать решение. Вот почему вы увидите, что в index.jsx мы сшиваем решение с требованиями и импортом.

выход

Здесь мы определяем, где webpack должен отбрасывать файлы файлов, которые он обнаружил для объединения. Мы также определили имя для файла, в котором наши связанные javascript и стили будут удалены.

devServer

Это настройки, специфичные для webpack-dev-сервера. `contentBase` определяет, где сервер должен сделать его root, мы определили `dist/` folder как нашу базу здесь. `port` - это порт, на котором будет размещаться сервер. `open` - это то, что используется, чтобы проинструктировать webpack-dev-сервер, чтобы открыть браузер по умолчанию для вас, как только он развернется на сервере.

модуль> погрузчики

Это определяет сопоставление для использования webpack, так что он знает, что делать, когда он находит разные файлы. Свойство `test` дает регулярное выражение для использования webpack для определения того, должен ли он применяться к этому модулю, в большинстве случаев мы имеем совпадения с расширениями файлов. `loader` или `loaders` дают имя модуля загрузчика, которое мы хотели бы использовать для загрузки файла в webpack, и пусть этот загрузчик позаботится о наборе этого типа файла. В javascript также есть свойство `query` , это просто предоставляет строку запроса загрузчику, поэтому мы могли бы использовать свойство запроса в html-загрузчике, если хотим. Это просто другой способ сделать что-то.

Examples

Как построить конвейер для настроенного «Hello world» с изображениями.

Шаг 1: Установите Node.js

Строка сборки, которую вы собираетесь строить, основана на Node.js, поэтому вы должны убедиться, что в первом случае это установлено. Для получения инструкций по установке Node.js вы можете проверить SO-документы для этого здесь.

Шаг 2. Инициализация вашего проекта как узла

Откройте папку проекта в командной строке и используйте следующую команду:

```
npm init
```

Для целей этого примера вы можете свободно принимать значения по умолчанию или если вам нужна дополнительная информация о том, что все это значит, вы можете проверить этот SO-Doc на настройке конфигурации пакета.

Шаг 3: Установите необходимые пакеты npm

Выполните следующую команду в командной строке, чтобы установить пакеты, необходимые для этого примера:

```
npm install --save react react-dom
```

Затем для зависимостей dev выполните следующую команду:

```
npm install --save-dev babel-core babel-preset-react babel-preset-es2015 webpack babel-loader
css-loader style-loader file-loader image-webpack-loader
```

Наконец, webpack и webpack-dev-сервер - это вещи, которые стоит устанавливать во всем мире, а не как зависимость от вашего проекта, если вы предпочтете добавить его в качестве зависимости, то это будет работать, я этого не сделаю. Вот команда запуска:

```
npm install --global webpack webpack-dev-server
```

Шаг 3. Добавьте файл .babelrc в корень вашего проекта.

Это позволит настроить babel для использования только что установленных пресетов. Ваш файл .babelrc должен выглядеть следующим образом:

```
{
  "presets": ["react", "es2015"]
}
```

Шаг 4. Настройка структуры каталога проекта.

Задайте себе структуру каталога, которая выглядит как ниже в корне вашего каталога:

```
|- node_modules
|- src/
  |- components/
  |- images/
  |- styles/
  |- index.html
  |- index.jsx
|- .babelrc
|- package.json
```

ПРИМЕЧАНИЕ. `node_modules` , `.babelrc` и `package.json` должен быть уже с предыдущих шагов, я просто включил их, чтобы вы могли видеть, где они подходят.

Шаг 5: Заполните проект с помощью файлов проекта Hello World

Это не очень важно для процесса построения конвейера, поэтому я просто дам вам код для них, и вы можете скопировать их в:

SRC / компоненты / HelloWorldComponent.jsx

```
import React, { Component } from 'react';

class HelloWorldComponent extends Component {
  constructor(props) {
    super(props);
    this.state = {name: 'Student'};
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange(e) {
    this.setState({name: e.target.value});
  }

  render() {
    return (
      <div>
        <div className="image-container">
          <img src="./images/myImage.gif" />
        </div>
        <div className="form">
          <input type="text" onChange={this.handleChange} />
          <div>
            My name is {this.state.name} and I'm a clever cloggs because I built a React build
pipeline
          </div>
        </div>
      </div>
    );
  }
}

export default HelloWorldComponent;
```

SRC / изображения / myImage.gif

Не стесняйтесь подставлять это любым изображением, которое вам нужно просто, чтобы доказать, что мы также можем связать изображения. Если вы предоставите свой собственный образ, и вы назовете его чем-то другим, вам придется обновить `HelloWorldComponent.jsx` чтобы отразить ваши изменения. Аналогично, если вы выбираете изображение с другим расширением файла, вам необходимо изменить свойство `test` загрузчика изображений в файле webpack.config.js с соответствующим регулярным выражением, чтобы оно соответствовало вашему новому расширению файла.

ЦСИ / стили / styles.css

```
.form {
  margin: 25px;
  padding: 25px;
  border: 1px solid #ddd;
  background-color: #eaeaea;
  border-radius: 10px;
}

.form div {
```

```
  padding-top: 25px;
}

.image-container {
  display: flex;
  justify-content: center;
}
```

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Learning to build a react pipeline</title>
</head>
<body>
  <div id="content"></div>
  <script src="app.js"></script>
</body>
</html>
```

index.jsx

```
import React from 'react';
import { render } from 'react-dom';
import HelloWorldComponent from './components/HelloWorldComponent.jsx';

require('./images/myImage.gif');
require('./styles/styles.css');
require('./index.html');

render(<HelloWorldComponent />, document.getElementById('content'));
```

Шаг 6: Создайте конфигурацию webpack

Создайте файл с именем webpack.config.js в корневом каталоге вашего проекта и скопируйте в него этот код:

webpack.config.js

```
var path = require('path');

var config = {
  context: path.resolve(__dirname + '/src'),
  entry: './index.jsx',
  output: {
    filename: 'app.js',
    path: path.resolve(__dirname + '/dist'),
  },
  devServer: {
    contentBase: path.join(__dirname + '/dist'),
    port: 3000,
    open: true,
  },
  module: {
```

```
    loaders: [
      {
        test: /\.(js|jsx)$/,
        exclude: /node_modules/,
        loader: 'babel-loader'
      },
      {
        test: /\.css$/,
        loader: "style!css"
      },
      {
        test: /\.gif$/,
        loaders: [
          'file?name=[path][name].[ext]',
          'image-webpack',
        ]
      },
      { test: /\.(html)$/,
        loader: "file?name=[path][name].[ext]"
      }
    ],
  },
};

module.exports = config;
```

Шаг 7. Создание задач npm для вашего конвейера.

Для этого вам нужно добавить два свойства в скриптовый ключ JSON, определенный в файле package.json в корне вашего проекта. Сделайте свой скрипт таким образом:

```
  "scripts": {
    "start": "webpack-dev-server",
    "build": "webpack",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
```

`test` скрипт уже был там, и вы можете выбрать, сохранять его или нет, это не важно для этого примера.

Шаг 8: Используйте трубопровод

Из командной строки, если вы находитесь в корневом каталоге проекта, вы должны теперь выполнить команду:

```
npm run build
```

Это свяжет небольшое приложение, которое вы создали, и поместите его в каталог `dist/` который будет создан в корневой папке вашего проекта.

Если вы запустите команду:

```
npm start
```

Затем созданное вами приложение будет обслуживаться в вашем веб-браузере по умолчанию внутри экземпляра сервера webpack dev.

Прочитайте Как настроить базовый веб-пакет, реагировать и загружать среду онлайн: https://riptutorial.com/ru/reactjs/topic/6294/как-настроить-базовый-веб-пакет--реагировать-и-загружать-среду

глава 11: Ключи реагирования

Вступление

Ключи реагирования используются для идентификации списка элементов DOM из одной и той же иерархии.

Поэтому, если вы выполняете итерацию по массиву, чтобы показать список элементов li, каждому элементу li нужен уникальный идентификатор, заданный свойством ключа. Обычно это может быть идентификатор вашего элемента базы данных или индекс массива.

замечания

Использование индекса массива в качестве ключа обычно не рекомендуется, когда массив будет меняться со временем. Из документов React:

> В крайнем случае вы можете передать индекс элемента в массиве в качестве ключа. Это может хорошо работать, если элементы никогда не переупорядочиваются, но переупорядочивание будет медленным.

Хороший пример: https://medium.com/@robinpokorny/index-as-a-key-is-an-anti-pattern-e0349aece318

Examples

Использование идентификатора элемента

Здесь у нас есть список предметов todo, которые передаются в реквизиты нашего компонента.

Каждый объект todo имеет свойство text и id. Представьте, что свойство id происходит из бэкэнд-хранилища данных и является уникальным числовым значением:

```
todos = [
  {
    id: 1,
    text: 'value 1'
  },
  {
    id: 2,
    text: 'value 2'
  },
  {
    id: 3,
    text: 'value 3'
```

```
  },
  {
    id: 4,
    text: 'value 4'
  },
];
```

Мы устанавливаем ключевой атрибут каждого итерированного элемента списка `todo-${todo.id}` чтобы реакция могла идентифицировать его внутри:

```
render() {
  const { todos } = this.props;
  return (
    <ul>
      { todos.map((todo) =>
        <li key={ `todo-${todo.id}` }>
          { todo.text }
        </li>
      }
    </ul>
  );
}
```

Использование индекса массива

Если у вас нет уникальных идентификаторов базы данных, вы также можете использовать числовой индекс вашего массива следующим образом:

```
render() {
  const { todos } = this.props;
  return (
    <ul>
      { todos.map((todo, index) =>
        <li key={ `todo-${index}` }>
          { todo.text }
        </li>
      }
    </ul>
  );
}
```

Прочитайте Ключи реагирования онлайн: https://riptutorial.com/ru/reactjs/topic/9805/ключи-реагирования

глава 12: Компоненты

замечания

`React.createClass` устарел в версии 15.5 и, как ожидается, будет удален в v16 . Для тех, которые все еще требуют этого, есть пакет замены замены . Примеры, использующие его, должны быть обновлены.

Examples

Основной компонент

Учитывая следующий HTML-файл:

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>React Tutorial</title>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react-dom.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-
core/5.8.34/browser.min.js"></script>
  </head>
  <body>
    <div id="content"></div>
    <script type="text/babel" src="scripts/example.js"></script>
  </body>
</html>
```

Вы можете создать базовый компонент, используя следующий код в отдельном файле:

скрипты / example.js

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';

class FirstComponent extends Component {
  render() {
    return (
      <div className="firstComponent">
        Hello, world! I am a FirstComponent.
      </div>
    );
  }
}
ReactDOM.render(
  <FirstComponent />, // Note that this is the same as the variable you stored above
  document.getElementById('content')
```

```
);
```

Вы получите следующий результат (обратите внимание, что внутри `div#content`):

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>React Tutorial</title>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react-dom.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-
core/5.8.34/browser.min.js"></script>
  </head>
  <body>
    <div id="content">
        <div className="firstComponent">
            Hello, world! I am a FirstComponent.
        </div>
    </div>
    <script type="text/babel" src="scripts/example.js"></script>
  </body>
</html>
```

Компоненты вложенности

Большая часть возможностей ReactJS - это способность разрешать вложенность компонентов. Возьмите следующие два компонента:

```
var React = require('react');
var createReactClass = require('create-react-class');

var CommentList = reactCreateClass({
  render: function() {
    return (
      <div className="commentList">
        Hello, world! I am a CommentList.
      </div>
    );
  }
});

var CommentForm = reactCreateClass({
  render: function() {
    return (
      <div className="commentForm">
        Hello, world! I am a CommentForm.
      </div>
    );
  }
});
```

Вы можете вставлять и ссылаться на эти компоненты в определении другого компонента:

```
var React = require('react');
```

```
var createReactClass = require('create-react-class');

var CommentBox = reactCreateClass({
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList /> // Which was defined above and can be reused
        <CommentForm /> // Same here
      </div>
    );
  }
});
```

Дальнейшая вложенность может быть выполнена тремя способами: у всех есть свои места для использования.

1. Вложение без использования детей

(продолжение сверху)

```
var CommentList = reactCreateClass({
  render: function() {
    return (
      <div className="commentList">
        <ListTitle/>
        Hello, world! I am a CommentList.
      </div>
    );
  }
});
```

Это стиль, в котором A составляет B и B составляет C.

Pros

- Легко и быстро отделяет элементы пользовательского интерфейса
- Легко вводить реквизиты до детей на основе состояния родительского компонента

Cons

- Меньшая видимость архитектуры композиции
- Меньшее повторное использование

Хорошо, если

- B и C представляют собой только презентационные компоненты
- B должен отвечать за жизненный цикл C

2. Вложение с использованием детей

(продолжение сверху)

```
var CommentBox = reactCreateClass({
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList>
            <ListTitle/> // child
        </CommentList>
        <CommentForm />
      </div>
    );
  }
});
```

Это стиль, в котором A составляет B и A указывает B на создание C. Больше мощности для родительских компонентов.

Pros

- Лучшее управление жизненным циклом компонентов
- Лучшая видимость архитектуры композиции
- Лучшая повторная эксплуатация

Cons

- Инъекции реквизита могут стать немного дороже
- Меньшая гибкость и мощность в детских компонентах

Хорошо, если

- B должен согласиться составить что-то другое, чем C в будущем или где-то еще
- A должен контролировать жизненный цикл C

B будет отображать C, используя `this.props.children` , и нет `this.props.children` структурированного способа, чтобы B знал, для чего предназначены эти дети. Таким образом, B может обогатить дочерние компоненты, предоставив дополнительные реквизиты, но если B должен точно знать, что они собой представляют, вариант № 3 может быть лучшим вариантом.

---

3. Вложение с помощью реквизита

(продолжение сверху)

```
var CommentBox = reactCreateClass({
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList title={ListTitle}/> //prop
        <CommentForm />
      </div>
    );
  }
});
```

Это стиль, в котором A составы B и B предоставляют возможность для A передать что-то, чтобы составить для определенной цели. Более структурированная композиция.

Pros

- Композиция как функция
- Простая проверка
- Лучшая композиционная способность

Cons

- Инъекции реквизита могут стать немного дороже
- Меньшая гибкость и мощность в детских компонентах

Хорошо, если

- B имеет специфические особенности, определенные для составления чего-либо
- B должен знать только, как визуализировать не то, что визуализировать

3 обычно является обязательным для создания публичной библиотеки компонентов, но также хорошей практикой в целом для создания составных компонентов и четкого определения функций композиции. # 1 - самый простой и быстрый способ сделать что-то, что работает, но # 2 и # 3 должны предоставлять определенные преимущества в различных случаях использования.

Создание компонентов

Это расширение базового примера:

Базовая структура

```
import React, { Component } from 'react';
import { render } from 'react-dom';

class FirstComponent extends Component {
    render() {
        return (
            <div>
                Hello, {this.props.name}! I am a FirstComponent.
            </div>
        );
    }
}

render(
    <FirstComponent name={ 'User' } />,
    document.getElementById('content')
);
```

Вышеприведенный пример называется **безстоящим** компонентом, так как он не содержит состояния (в смысле ответа).

В этом случае некоторые люди считают предпочтительным использовать функциональные компоненты Stateless, которые основаны на функциях стрелок ES6 .

Функциональные компоненты без учета состояния

Во многих приложениях есть интеллектуальные компоненты, которые содержат состояние, но отображают немые компоненты, которые просто получают реквизит и возвращают HTML как JSX. Функциональные компоненты без учета состояния намного более многоразовые и оказывают положительное влияние на ваше приложение.

Они имеют две основные характеристики:

1. При визуализации они получают объект со всеми реквизитами, которые были переданы
2. Они должны вернуть JSX для отображения

```
// When using JSX inside a module you must import React
import React from 'react';
import PropTypes from 'prop-types';

const FirstComponent = props => (
    <div>
        Hello, {props.name}! I am a FirstComponent.
    </div>
);

//arrow components also may have props validation
FirstComponent.propTypes = {
```

```
    name: PropTypes.string.isRequired,
}

// To use FirstComponent in another file it must be exposed through an export call:
export default FirstComponent;
```

Компоненты состояния

В отличие от вышеперечисленных компонентов «без состояния» компоненты «stateful» имеют объект состояния, который может быть обновлен с `setState` метода `setState` . Состояние должно быть инициализировано в `constructor` до его установки:

```
import React, { Component } from 'react';

class SecondComponent extends Component {
    constructor(props) {
        super(props);

        this.state = {
            toggle: true
        };

        // This is to bind context when passing onClick as a callback
        this.onClick = this.onClick.bind(this);
    }

    onClick() {
        this.setState((prevState, props) => ({
            toggle: !prevState.toggle
        }));
    }

    render() {
        return (
            <div onClick={this.onClick}>
                Hello, {this.props.name}! I am a SecondComponent.
                <br />
                Toggle is: {this.state.toggle}
            </div>
        );
    }
}
```

Расширение компонента с помощью `PureComponent` вместо `Component` будет автоматически реализовывать метод жизненного цикла `shouldComponentUpdate()` с неглубокой поддержкой и сравнением состояний. Это позволяет повысить эффективность вашего приложения за счет уменьшения количества ненужных визуализаций, которые происходят. Это предполагает, что ваши компоненты являются «чистыми» и всегда отображают один и тот же результат с одним и тем же состоянием и реквизитом.

Компоненты более высокого порядка

Компоненты более высокого порядка (HOC) позволяют совместно использовать функциональность компонента.

```
import React, { Component } from 'react';

const PrintHello = ComposedComponent => class extends Component {
    onClick() {
        console.log('hello');
    }

    /* The higher order component takes another component as a parameter
    and then renders it with additional props */
    render() {
        return <ComposedComponent {...this.props } onClick={this.onClick} />
    }
}

const FirstComponent = props => (
    <div onClick={ props.onClick }>
        Hello, {props.name}! I am a FirstComponent.
    </div>
);

const ExtendedComponent = PrintHello(FirstComponent);
```

Компоненты более высокого порядка используются, когда вы хотите делиться логикой между несколькими компонентами независимо от того, насколько они отличаются.

Недостатки setState

Следует соблюдать осторожность при использовании `setState` в асинхронном контексте. Например, вы можете попытаться вызвать `setState` в `setState` вызове `setState` get:

```
class MyClass extends React.Component {
    constructor() {
        super();

        this.state = {
            user: {}
        };
    }

    componentDidMount() {
        this.fetchUser();
    }

    fetchUser() {
        $.get('/api/users/self')
            .then((user) => {
                this.setState({user: user});
            });
    }

    render() {
        return <h1>{this.state.user}</h1>;
    }
```

```
}
```

Это может вызвать проблемы - если вызываемый вызов вызывается после демонтажа `Component` , то `this.setState` не будет функцией. Всякий раз, когда это так, вы должны быть осторожны, чтобы ваше использование `setState` .

В этом примере вы можете отменить запрос XHR, когда компонент отключится:

```
class MyClass extends React.Component {
    constructor() {
        super();

        this.state = {
            user: {},
            xhr: null
        };
    }

    componentWillUnmount() {
        let xhr = this.state.xhr;

        // Cancel the xhr request, so the callback is never called
        if (xhr && xhr.readyState != 4) {
            xhr.abort();
        }
    }

    componentDidMount() {
        this.fetchUser();
    }

    fetchUser() {
        let xhr = $.get('/api/users/self')
            .then((user) => {
                this.setState({user: user});
            });

        this.setState({xhr: xhr});
    }
}
```

Асинхронный метод сохраняется как состояние. В `componentWillUnmount` вы выполняете всю свою очистку - включая отмену запроса XHR.

Вы могли бы также сделать что-то более сложное. В этом примере я создаю функцию stateSetter, которая принимает этот объект в качестве аргумента и предотвращает `this.setState` когда `this.setState` функция `cancel` :

```
function stateSetter(context) {
    var cancelled = false;
    return {
        cancel: function () {
            cancelled = true;
        },
        setState(newState) {
```

```
            if (!cancelled) {
                context.setState(newState);
            }
        }
    }
}

class Component extends React.Component {
    constructor(props) {
        super(props);
        this.setter = stateSetter(this);
        this.state = {
            user: 'loading'
        };
    }
    componentWillUnmount() {
        this.setter.cancel();
    }
    componentDidMount() {
        this.fetchUser();
    }
    fetchUser() {
        $.get('/api/users/self')
            .then((user) => {
                this.setter.setState({user: user});
            });
    }
    render() {
        return <h1>{this.state.user}</h1>
    }
}
```

Это работает, потому что `cancelled` переменная видна в закрытии `setState` мы создали.

Реквизит

Реквизиты - это способ передачи информации в компонент React, у них может быть любой тип, включая функции, иногда называемые обратными вызовами.

В реквизитах JSX передаются синтаксис атрибута

```
<MyComponent userID={123} />
```

Внутри определения для MyComponent userID теперь будет доступен из объекта реквизита

```
// The render function inside MyComponent
render() {
    return (
        <span>The user's ID is {this.props.userID}</span>
    )
}
```

Важно определить все `props` , их типы и, где применимо, их значение по умолчанию:

```
// defined at the bottom of MyComponent
MyComponent.propTypes = {
    someObject: React.PropTypes.object,
    userID: React.PropTypes.number.isRequired,
    title: React.PropTypes.string
};

MyComponent.defaultProps = {
    someObject: {},
    title: 'My Default Title'
}
```

В этом примере prop `someObject` является необязательным, но требуется `userID` prop. Если вы не предоставите `userID` `MyComponent` , во время выполнения движок React отобразит консоль, предупреждающую вас о том, что требуемая поддержка не была предоставлена. Остерегайтесь, однако, это предупреждение отображается только в версии версии библиотеки React, производственная версия не будет регистрировать никаких предупреждений.

Использование `defaultProps` позволяет упростить

```
const { title = 'My Default Title' } = this.props;
console.log(title);
```

в

```
console.log(this.props.title);
```

Это также является гарантией использования `array` `object` и `functions` . Если вы не предоставляете опору по умолчанию для объекта, следующее будет вызывать ошибку, если оповещение не передается:

```
if (this.props.someObject.someKey)
```

В приведенном выше примере `this.props.someObject` не `undefined` и поэтому проверка `someKey` вызовет ошибку, и код сломается. С помощью `defaultProps` вы можете безопасно использовать вышеуказанную проверку.

Состояние компонентов - Динамический пользовательский интерфейс

Предположим, что мы хотим иметь следующее поведение: у нас есть заголовок (например, элемент h3) и при нажатии на него мы хотим, чтобы он стал полем ввода, чтобы мы могли изменить название заголовка. React делает это очень простым и интуитивно понятным с использованием состояний компонентов, а если иными словами. (Пояснение ниже)

```
// I have used ReactBootstrap elements. But the code works with regular html elements also
var Button = ReactBootstrap.Button;
var Form = ReactBootstrap.Form;
```

```
var FormGroup = ReactBootstrap.FormGroup;
var FormControl = ReactBootstrap.FormControl;

var Comment = reactCreateClass({
  getInitialState: function() {
    return {show: false, newTitle: ''};
  },

  handleTitleSubmit: function() {
     //code to handle input box submit - for example, issue an ajax request to change name in
database
  },

  handleTitleChange: function(e) {
    //code to change the name in form input box. newTitle is initialized as empty string. We
need to update it with the string currently entered by user in the form
    this.setState({newTitle: e.target.value});
  },

  changeComponent: function() {
    // this toggles the show variable which is used  for dynamic UI
    this.setState({show: !this.state.show)};
  },

  render: function() {

    var clickableTitle;

    if(this.state.show) {
        clickableTitle = <Form inline onSubmit={this.handleTitleSubmit}>
                            <FormGroup controlId="formInlineTitle">
                                <FormControl type="text" onChange={this.handleTitleChange}>
                            </FormGroup>
                        </Form>;
    } else {
        clickabletitle = <div>
                           <Button bsStyle="link" onClick={this.changeComponent}>
                               <h3> Default Text </h3>
                           </Button>
                        </div>;
    }

    return (
        <div className="comment">
            {clickableTitle}
        </div>
    );
  }
});

ReactDOM.render(
    <Comment />, document.getElementById('content')
);
```

Основной частью кода является переменная **clickableTitle** . На основании переменного состояния **шоу,** это может быть либо элемент формы или элемент Button. Реакция позволяет вложенность компонентов.

Таким образом, мы можем добавить элемент {clickableTitle} в функции рендеринга. Он ищет

переменную clickableTitle. Основываясь на значении «this.state.show», он отображает соответствующий элемент.

Вариации функциональных компонентов без учета состояния

```
const languages = [
  'JavaScript',
  'Python',
  'Java',
  'Elm',
  'TypeScript',
  'C#',
  'F#'
]
```

```
// one liner
const Language = ({language}) => <li>{language}</li>

Language.propTypes = {
  message: React.PropTypes.string.isRequired
}
```

```
/**
* If there are more than one line.
* Please notice that round brackets are optional here,
* However it's better to use them for readability
*/
const LanguagesList = ({languages}) => {
  <ul>
    {languages.map(language => <Language language={language} />)}
  </ul>
}

LanguagesList.PropTypes = {
  languages: React.PropTypes.array.isRequired
}
```

```
/**
 * This syntax is used if there are more work beside just JSX presentation
 * For instance some data manipulations needs to be done.
 * Please notice that round brackets after return are required,
 * Otherwise return will return nothing (undefined)
 */
const LanguageSection = ({header, languages}) => {
  // do some work
  const formattedLanguages = languages.map(language => language.toUpperCase())
  return (
    <fieldset>
      <legend>{header}</legend>
      <LanguagesList languages={formattedLanguages} />
    </fieldset>
  )
}
```

```
LanguageSection.PropTypes = {
  header: React.PropTypes.string.isRequired,
  languages: React.PropTypes.array.isRequired
}
```

```
ReactDOM.render(
  <LanguageSection
    header="Languages"
    languages={languages} />,
  document.getElementById('app')
)
```

Здесь вы можете найти рабочий пример.

Прочитайте Компоненты онлайн: https://riptutorial.com/ru/reactjs/topic/1185/компоненты

глава 13: Компоненты более высокого порядка

Вступление

Компоненты более высокого порядка («HOC» вкратце) представляет собой шаблон дизайна реагирующего приложения, который используется для улучшения компонентов с использованием многоразового кода. Они позволяют добавлять функциональные возможности и поведение к существующим классам компонентов.

HOC - это чистая функция javascript, которая принимает компонент как аргумент и возвращает новый компонент с расширенной функциональностью.

замечания

HOC довольно часто используются в сторонних библиотеках. Например, функция подключения Redux.

Examples

Простой компонент более высокого порядка

Предположим, мы хотим, чтобы console.log каждый раз, когда компонент монтируется:

hocLogger.js

```
export default function hocLogger(Component) {
  return class extends React.Component {
    componentDidMount() {
      console.log('Hey, we are mounted!');
    }
    render() {
      return <Component {...this.props} />;
    }
  }
}
```

Используйте этот HOC в своем коде:

MyLoggedComponent.js

```
import React from "react";
import {hocLogger} from "./hocLogger";

export class MyLoggedComponent extends React.Component {
```

```
    render() {
        return (
            <div>
                This component get's logged to console on each mount.
            </div>
        );
    }
}

// Now wrap MyLoggedComponent with the hocLogger function
export default hocLogger(MyLoggedComponent);
```

Компонент более высокого порядка, который проверяет подлинность

Допустим, у нас есть компонент, который должен отображаться только в том случае, если пользователь вошел в систему.

Поэтому мы создаем HOC, который проверяет подлинность для каждого render ():

AuthenticatedComponent.js

```
import React from "react";

export function requireAuthentication(Component) {
    return class AuthenticatedComponent extends React.Component {

        /**
         * Check if the user is authenticated, this.props.isAuthenticated
         * has to be set from your application logic (or use react-redux to retrieve it from
global state).
         */
        isAuthenticated() {
            return this.props.isAuthenticated;
        }

        /**
         * Render
         */
        render() {
            const loginErrorMessage = (
                <div>
                    Please <a href="/login">login</a> in order to view this part of the
application.
                </div>
            );

            return (
                <div>
                    { this.isAuthenticated === true ? <Component {...this.props} /> :
loginErrorMessage }
                </div>
            );
        }
    };
}

export default requireAuthentication;
```

Затем мы просто используем этот компонент более высокого порядка в наших компонентах, которые должны быть скрыты от анонимных пользователей:

MyPrivateComponent.js

```
import React from "react";
import {requireAuthentication} from "./AuthenticatedComponent";

export class MyPrivateComponent extends React.Component {
    /**
     * Render
     */
    render() {
        return (
            <div>
                My secret search, that is only viewable by authenticated users.
            </div>
        );
    }
}

// Now wrap MyPrivateComponent with the requireAuthentication function
export default requireAuthentication(MyPrivateComponent);
```

Этот пример более подробно описан здесь .

Прочитайте Компоненты более высокого порядка онлайн:
https://riptutorial.com/ru/reactjs/topic/9819/компоненты-более-высокого-порядка

глава 14: Монтаж

Examples

Простая настройка

Настройка папок

В этом примере предполагается, что код находится в `src/` и выводится вывод `out/` . Таким образом, структура папок должна выглядеть примерно так:

```
example/
|-- src/
|   |-- index.js
|   `-- ...
|-- out/
`-- package.json
```

Настройка пакетов

Предполагая среду установки npm, мы сначала должны установить babel, чтобы перевести код React в es5-совместимый код.

```
$npm install --save-dev babel-core babel-loader babel-preset-es2015 babel-preset-react
```

Вышеупомянутая команда будет инструктировать npm для установки основных библиотек бабелей, а также модуля загрузчика для использования с webpack. Мы также устанавливаем es6 и реагируем на предустановки для babel, чтобы понять код модуля JSX и es6. (Более подробную информацию о пресетах можно найти здесь в настройках Babel)

```
$npm i -D webpack
```

Эта команда установит webpack как зависимость от разработки. (**i** является сокращением для установки и **-D** сокращением для -save-dev)

Вы также можете установить любые дополнительные пакеты веб-пакетов (например, дополнительные загрузчики или расширение webpack-dev-server)

Наконец, нам понадобится фактический код реакции

```
$npm i -D react react-dom
```

Настройка веб-пакета

При настройке зависимостей нам понадобится файл webpack.config.js, чтобы сообщить webpack, что делать

простой webpack.config.js:

```
var path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'out'),
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      {
        test: /\.js$/,
        exclude: /(node_modules)/,
        loader: 'babel-loader',
        query: {
          presets: ['es2015', 'react']
        }
      }
    ]
  }
};
```

Этот файл сообщает webpack начать с файла index.js (предполагается, что он находится в src /) и преобразовать его в один файл bundle.js в каталог `out` .

`module` блокирует webpack для проверки всех файлов, встречающихся с регулярным выражением, и если они совпадают, вызывается указанный загрузчик. (`babel-loader` в данном случае) Кроме того, `exclude` регулярное выражение говорит WebPack игнорировать этот специальный загрузчик для всех модулей в `node_modules` папке, это помогает ускорить процесс transpilation. Наконец, опция `query` сообщает webpack, какие параметры передаются в babel, и используется для передачи по предустановленным ранее установкам.

Тестирование настройки

`src/index.js` файл `src/index.js` и попробовать упаковать приложение

SRC / index.js:

```
'use strict'

import React from 'react'
import { render } from 'react-dom'
```

```
const App = () => {
  return <h1>Hello world!</h1>
}

render(
  <App />,
  document. getElementById('app')
)
```

Этот файл обычно выводит простой заголовок `<h1>Hello world!</h1>` в тег html с идентификатором 'app', но пока этого должно быть достаточно, чтобы перевести код один раз.

`$./node_modules/.bin/webpack` . Запустит локально установленную версию webpack (используйте `$webpack` если вы установили webpack глобально с -g)

Это должно создать файл `out/bundle.js` с преобразованным кодом внутри и завершает пример.

Использование webpack-dev-сервера

Настроить

После настройки простого проекта для использования webpack, babel и реакции на выпуск `$npm i -g webpack-dev-server` будет устанавливать HTTP-сервер разработки для более быстрой разработки.

Изменение webpack.config.js

```
var path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'out'),
    publicPath: '/public/',
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      {
        test: /\.js$/,
        exclude: /(node_modules)/,
        loader: 'babel',
        query: {
          presets: ['es2015', 'react']
        }
      }
    ]
```

```
  },
  devServer: {
    contentBase: path.resolve(__dirname, 'public'),
    hot: true
  }
};
```

Изменения в

- `output.publicPath` который устанавливает путь для обслуживания нашего пакета (см. файлы конфигурации Webpack для получения дополнительной информации)
- `devServer`
    - `contentBase` базовый путь для обслуживания статических файлов (например, `index.html`)
    - `hot` устанавливает webpack-dev-сервер для горячей перезагрузки при внесении изменений в файлы на диске

И, наконец, нам просто нужен простой index.html для тестирования нашего приложения.

index.html:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>React Sandbox</title>
  </head>
  <body>

    <div id="app" />

    <script src="public/bundle.js"></script>
  </body>
</html>
```

С этой настройкой, запущенной `$webpack-dev-server` следует запустить локальный HTTP-сервер на порту 8080, а при подключении должен отобразить страницу, содержащую `<h1>Hello world!</h1>` .

Прочитайте Монтаж онлайн: https://riptutorial.com/ru/reactjs/topic/6441/монтаж

глава 15: Настройка среды реагирования

Examples

Простой реакционный компонент

Мы хотим иметь возможность компилировать ниже компонент и отображать его на нашей веб-странице.

Имя файла : src / index.jsx

```
import React from 'react';
import ReactDOM from 'react-dom';

class ToDo extends React.Component {
    render() {
        return (<div>I am working</div>);
    }
}

ReactDOM.render(<ToDo />, document.getElementById('App'));
```

Установка всех зависимостей

```
# install react and react-dom
$ npm i react react-dom --save

# install webpack for bundling
$ npm i webpack -g

# install babel for module loading, bundling and transpiling
$ npm i babel-core babel-loader --save

# install babel presets for react and es6
$ npm i babel-preset-react babel-preset-es2015 --save
```

Настроить webpack

Создайте файл `webpack.config.js` в корне вашей рабочей директории

Имя файла : webpack.config.js

```
module.exports = {
    entry: __dirname + "/src/index.jsx",
    devtool: "source-map",
    output: {
        path: __dirname + "/build",
        filename: "bundle.js"
    },
```

```
    module: {
        loaders: [
            {test: /\.jsx?$/, exclude: /node_modules/, loader: "babel-loader"}
        ]
    }
}
```

Настроить babel

Создайте файл `.babelrc` в корне нашего рабочего каталога

Имя файла : .babelrc

```
{
    "presets": ["es2015","react"]
}
```

HTML-файл для использования компонента реакции

Настройте простой html-файл в корне каталога проекта

Имя файла : index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <div id="App"></div>
    <script src="build/bundle.js" charset="utf-8"></script>
  </body>
</html>
```

Транспортируйте и свяжите свой компонент

Используя webpack, вы можете связать свой компонент:

```
$ webpack
```

Это создаст наш выходной файл в каталоге `build` .

Откройте HTML-страницу в браузере, чтобы увидеть компонент в действии

Прочитайте Настройка среды реагирования онлайн:
https://riptutorial.com/ru/reactjs/topic/7480/настройка-среды-реагирования

глава 16: Общение между компонентами

Examples

Связь между функциональными компонентами без состояния

В этом примере мы будем использовать модули `Redux` и `React Redux` для обработки состояния приложения и для автоматического повторного отображения наших функциональных компонентов. И, конечно же, `React` and `React Dom`

Вы можете проверить завершенную демо-версию здесь

В приведенном ниже примере мы имеем три разных компонента и один подключенный компонент

- **UserInputForm** : этот компонент отображает поле ввода. Когда значение поля изменяется, он вызывает метод `inputChange` на `props` (который предоставляется родительским компонентом), и если данные также предоставляются, он отображает это в поле ввода.
- **UserDashboard:** Этот компонент выводит простое сообщение , а также гнездится `UserInputForm` компонента, он также проходит `inputChange` метод `UserInputForm` компонент, `UserInputForm` компонент INTURN использует этот метод для связи с родительским компонентом.
    - **UserDashboardConnected** : этот компонент просто обертывает компонент `UserDashboard` с помощью `ReactRedux connect` . Это упрощает управление состоянием компонента и обновление компонента при изменении состояния.
- **Приложение** . Этот компонент просто отображает компонент `UserDashboardConnected` .

```
const UserInputForm = (props) => {

  let handleSubmit = (e) => {
    e.preventDefault();
  }

  return(
    <form action="" onSubmit={handleSubmit}>
      <label htmlFor="name">Please enter your name</label>
      <br />
      <input type="text" id="name" defaultValue={props.data.name || ''} onChange={
props.inputChange } />
    </form>
  )

}
```

```
const UserDashboard = (props) => {

  let inputChangeHandler = (event) => {
    props.updateName(event.target.value);
  }

  return(
    <div>
      <h1>Hi { props.user.name || 'User' }</h1>
      <UserInputForm data={props.user} inputChange={inputChangeHandler} />
    </div>
  )

}

const mapStateToProps = (state) => {
  return {
    user: state
  };
}
const mapDispatchToProps = (dispatch) => {
  return {
    updateName: (data) => dispatch( Action.updateName(data) ),
  };
};

const { connect, Provider } = ReactRedux;
const UserDashboardConnected = connect(
  mapStateToProps,
  mapDispatchToProps
)(UserDashboard);



const App = (props) => {
  return(
    <div>
      <h1>Communication between Stateless Functional Components</h1>
      <UserDashboardConnected />
    </div>
  )
}



const user = (state={name: 'John'}, action) => {
  switch (action.type) {
    case 'UPDATE_NAME':
      return Object.assign( {}, state, {name: action.payload}  );

    default:
      return state;
  }
};

const { createStore } = Redux;
const store = createStore(user);
const Action = {
  updateName: (data) => {
    return { type : 'UPDATE_NAME', payload: data }
```

```
  },
}


ReactDOM.render(
  <Provider store={ store }>
    <App />
  </Provider>,
  document.getElementById('application')
);
```

URL-адрес JS Bin

Прочитайте Общение между компонентами онлайн: https://riptutorial.com/ru/reactjs/topic/6137/общение-между-компонентами

глава 17: Реагирование, установка Webpack & TypScript

замечания

Чтобы получить подсветку синтаксиса в вашем редакторе (например, код VS), вам нужно будет загрузить информацию для ввода модулей, которые вы используете в своем проекте.

Скажем, например, вы используете React и ReactDOM в своем проекте, и вы хотите получить подсветку и Intellisense для них. Вам нужно будет добавить типы в свой проект, используя следующую команду:

```
npm install --save @types/react @types/react-dom
```

Теперь ваш редактор должен автоматически забрать эту информацию и предоставить вам автозаполнение и Intellisense для этих модулей.

Examples

webpack.config.js

```
module.exports = {
    entry: './src/index',
    output: {
        path: __dirname + '/build',
        filename: 'bundle.js'
    },
    module: {
        rules: [{
            test: /\.tsx?$/,
            loader: 'ts-loader',
            exclude: /node_modules/
        }]
    },
    resolve: {
        extensions: ['.ts', '.tsx']
    }
};
```

Основные компоненты являются (в дополнении к стандартному `entry` , `output` и другим свойствам WebPack):

Погрузчик

Для этого вам нужно создать правило, которое проверяет расширения файлов `.ts` и `.tsx` , в качестве загрузчика укажите `ts-loader` loader.

Устранение расширений TS

Вам также нужно добавить расширения `.ts` и `.tsx` в массив `resolve` , или webpack их не увидит.

tsconfig.json

Это минимальный tsconfig, чтобы вы могли работать.

```
{
    "include": [
        "src/*"
    ],
    "compilerOptions": {
        "target": "es5",
        "jsx": "react",
        "allowSyntheticDefaultImports": true
    }
}
```

Давайте рассмотрим свойства один за другим:

include

Это массив исходного кода. Здесь у нас есть только одна запись, `src/*` , которая указывает, что все в каталоге `src` должно быть включено в компиляцию.

compilerOptions.target

Указывает, что мы хотим скомпилировать целевой объект ES5

compilerOptions.jsx

Установка этого значения в `true` заставит TypeScript автоматически скомпилировать ваш синтаксис `React.createElement("div")` из `<div />` в `React.createElement("div")` .

compilerOptions.allowSyntheticDefaultImports

Удобное свойство, которое позволит вам импортировать модули узла, как если бы они были модулями ES6, поэтому вместо того, чтобы делать

```
import * as React from 'react'
const { Component } = React
```

вы можете просто сделать

```
import React, { Component } from 'react'
```

без каких-либо ошибок, сообщая вам, что React не имеет экспорта по умолчанию.

Мой первый компонент

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';


interface AppProps {
    name: string;
}
interface AppState {
    words: string[];
}

class App extends Component<AppProps, AppState> {
    constructor() {
        super();
        this.state = {
            words: ['foo', 'bar']
        };
    }

    render() {
        const { name } = this.props;
        return (<h1>Hello {name}!</h1>);
    }
}

const root = document.getElementById('root');
ReactDOM.render(<App name="Foo Bar" />, root);
```

При использовании TypeScript с React, как только вы загрузите определения типа React DefinitelyTyped (`npm install --save @types/react`), каждый компонент потребует добавления аннотаций типа.

Вы делаете это так:

```
class App extends Component<AppProps, AppState> { }
```

где `AppProps` и `AppState` - это интерфейсы (или псевдонимы типов) для реквизита ваших компонентов и состояния соответственно.

Прочитайте Реагирование, установка Webpack & TypScript онлайн:
https://riptutorial.com/ru/reactjs/topic/9590/реагирование--установка-webpack--amp--typscript

глава 18: Реагировать жизненный цикл компонентов

Вступление

Методы жизненного цикла должны использоваться для запуска кода и взаимодействия с вашим компонентом в разных точках жизни компонентов. Эти методы основаны на компоненте «Монтаж, обновление и размонтирование».

Examples

Создание компонентов

Когда компонент React создается, вызывается целый ряд функций:

- Если вы используете `React.createClass` (ES5), вызывается 5 пользовательских функций
- Если вы используете `class Component extends React.Component` (ES6), вызывается 3 пользовательских функции

`getDefaultProps()` (только для ES5)

Это **первый** вызванный метод.

Значения Prop, возвращаемые этой функцией, будут использоваться как значения по умолчанию, если они не определены при создании экземпляра компонента.

В следующем примере `this.props.name` будет по умолчанию для `Bob` если не указано иное:

```
getDefaultProps() {
  return {
    initialCount: 0,
    name: 'Bob'
  };
}
```

`getInitialState()` (только для ES5)

Это **второй** метод.

Возвращаемое значение `getInitialState()` определяет начальное состояние компонента React. Структура React вызовет эту функцию и назначит возвращаемое значение `this.state`

.

В следующем примере `this.state.count` будет проиндексирован со значением `this.props.initialCount` :

```
getInitialState() {
  return {
    count : this.props.initialCount
  };
}
```

`componentWillMount()` (ES5 и ES6)

Это **третий** вызванный метод.

Эта функция может быть использована для внесения окончательных изменений в компонент до его добавления в DOM.

```
componentWillMount() {
  ...
}
```

`render()` (ES5 и ES6)

Это **четвертый** метод.

Функция `render()` должна быть чистой функцией состояния компонента и реквизита. Он возвращает один элемент, который представляет компонент во время процесса рендеринга, и должен быть либо представлением собственного компонента DOM (например, `<p />`), либо составного компонента. Если ничего не нужно сделать, он может вернуть значение `null` или `undefined` .

Эта функция будет вызвана после любого изменения реквизита или состояния компонента.

```
render() {
  return (
    <div>
      Hello, {this.props.name}!
    </div>
  );
}
```

`componentDidMount()` (ES5 и ES6)

Это **пятый** метод.

Компонент смонтирован, и теперь вы можете получить доступ к узлам DOM компонента, например, посредством `refs` .

Этот метод следует использовать для:

- Подготовка таймеров
- Получение данных
- Добавление прослушивателей событий
- Манипулирование элементами DOM

```
componentDidMount() {
  ...
}
```

Синтаксис ES6

Если компонент определен с использованием синтаксиса класса ES6, функции `getDefaultProps()` и `getInitialState()` не могут использоваться.

Вместо этого мы объявляем наши `defaultProps` как статическое свойство в классе и объявляем форму состояния и начальное состояние в конструкторе нашего класса. Они устанавливаются в экземпляре класса во время построения, до того, как вызывается любая другая функция жизненного цикла React.

Следующий пример демонстрирует этот альтернативный подход:

```
class MyReactClass extends React.Component {
  constructor(props){
    super(props);

    this.state = {
      count: this.props.initialCount
    };
  }

  upCount() {
    this.setState((prevState) => ({
      count: prevState.count + 1
    }));
  }

  render() {
    return (
      <div>
        Hello, {this.props.name}!<br />
        You clicked the button {this.state.count} times.<br />
        <button onClick={this.upCount}>Click here!</button>
      </div>
    );
```

```
  }
}

MyReactClass.defaultProps = {
  name: 'Bob',
  initialCount: 0
};
```

Замена `getDefaultProps()`

Значения по умолчанию для реквизита компонента определяются установкой свойства `defaultProps` класса:

```
MyReactClass.defaultProps = {
  name: 'Bob',
  initialCount: 0
};
```

Замена `getInitialState()`

Идиоматическим способом настройки начального состояния компонента является установка `this.state` в конструкторе:

```
constructor(props){
  super(props);

  this.state = {
    count: this.props.initialCount
  };
}
```

Обновление компонента

`componentWillReceiveProps(nextProps)`

Это **первая функция, называемая изменением свойств** .

Когда **свойства компонента изменяются** , React вызовет эту функцию с **новыми свойствами** . Вы можете получить доступ к старым реквизитам с помощью *this.props* и к новым реквизитам с *nextProps* .

С помощью этих переменных вы можете выполнять некоторые операции сравнения между старым и новым реквизитом или функцией вызова, потому что изменение свойства и т. Д.

```
componentWillReceiveProps(nextProps){
  if (nextProps.initialCount && nextProps.initialCount > this.state.count){
    this.setState({
      count : nextProps.initialCount
    });
```

```
  }
}
```

shouldComponentUpdate(nextProps, nextState)

Это **вторая функция, называемая изменением свойств, и первая смена состояния** .

По умолчанию, если другой компонент / ваш компонент изменяет свойство / состояние вашего компонента, **React** отобразит новую версию вашего компонента. В этом случае эта функция всегда возвращает true.

Вы можете переопределить эту функцию и **выбрать более точно, если ваш компонент должен обновить или нет** .

Эта функция в основном используется для **оптимизации** .

В случае, если функция возвращает **false** , **конвейер обновлений останавливается немедленно** .

```
componentShouldUpdate(nextProps, nextState){
  return this.props.name !== nextProps.name ||
    this.state.count !== nextState.count;
}
```

componentWillUpdate(nextProps, nextState)

Эта функция работает как `componentWillMount()` . **Изменения не внесены в DOM** , поэтому вы можете внести некоторые изменения непосредственно перед обновлением.

/! \: вы не можете использовать **this.setState ()** .

```
componentWillUpdate(nextProps, nextState){}
```

render()

Есть некоторые изменения, поэтому повторно отрисуйте компонент.

componentDidUpdate(prevProps, prevState)

Тот же материал, что и `componentDidMount()` : **DOM обновлен** , поэтому вы можете выполнить некоторую работу над DOM здесь.

```
componentDidUpdate(prevProps, prevState){}
```

Удаление компонентов

componentWillUnmount()

Этот метод вызывается **до того,** как компонент будет удален из DOM.

Это хорошее место для выполнения операций очистки, таких как:

- Удаление прослушивателей событий.
- Очистка таймеров.
- Остановка гнезд.
- Очистка состояний редукции.

```
componentWillUnmount(){
  ...
}
```

Пример удаления подключенного прослушивателя событий в `componentWillUnMount`

```
import React, { Component } from 'react';

export default class SideMenu extends Component {

  constructor(props) {
    super(props);
    this.state = {
        ...
      };
    this.openMenu = this.openMenu.bind(this);
    this.closeMenu = this.closeMenu.bind(this);
  }

  componentDidMount() {
    document.addEventListener("click", this.closeMenu);
  }

  componentWillUnmount() {
    document.removeEventListener("click", this.closeMenu);
  }

  openMenu() {
    ...
  }

  closeMenu() {
    ...
  }

  render() {
    return (
      <div>
        <a
          href      = "javascript:void(0)"
          className = "closebtn"
          onClick   = {this.closeMenu}
        >
          x
        </a>
```

```
        <div>
          Some other structure
        </div>
      </div>
    );
  }
}
```

Контейнер для реактивов

При создании приложения React часто желательно разделить компоненты на основе их основной ответственности, в компоненты Presentation и Container.
Презентационные компоненты касаются только отображения данных - их можно рассматривать и часто реализуются как функции, которые преобразуют модель в представление. Обычно они не поддерживают никакого внутреннего состояния.
Контейнерные компоненты связаны с управлением данными. Это может быть сделано внутренне через их собственное государство, или действуя как посредники с государственной административной библиотекой, такой как Redux. Контейнерный компонент не будет отображать данные напрямую, а передает данные в презентационный компонент.

```
// Container component
import React, { Component } from 'react';
import Api from 'path/to/api';

class CommentsListContainer extends Component {
    constructor() {
        super();
        // Set initial state
        this.state = { comments: [] }
    }

    componentDidMount() {
        // Make API call and update state with returned comments
        Api.getComments().then(comments => this.setState({ comments }));
    }

    render() {
        // Pass our state comments to the presentational component
        return (
            <CommentsList comments={this.state.comments} />;
        );
    }
}

// Presentational Component
const CommentsList = ({ comments }) => (
    <div>
        {comments.map(comment => (
            <div>{comment}</div>
        )}
    </div>
);

CommentsList.propTypes = {
```

```
    comments: React.PropTypes.arrayOf(React.PropTypes.string)
}
```

Метод вызова жизненного цикла в разных состояниях

Этот пример служит дополнением к другим примерам, в которых рассказывается о том, как использовать методы жизненного цикла и когда будет вызван метод.

В этом примере суммировать, какие методы (componentWillMount, componentWillReceiveProps и т. Д.) Будут вызываться и в какой последовательности будет отличаться для компонента **в разных состояниях** :

Когда компонент инициализируется:

1. getDefaultProps
2. getInitialState
3. componentWillMount
4. оказывать
5. componentDidMount

При изменении состояния компонента:

1. shouldComponentUpdate
2. componentWillUpdate
3. оказывать
4. componentDidUpdate

Когда компонент имеет реквизиты, изменилось:

1. componentWillReceiveProps
2. shouldComponentUpdate
3. componentWillUpdate
4. оказывать
5. componentDidUpdate

Когда компонент размонтирует:

1. componentWillUnmount

Прочитайте Реагировать жизненный цикл компонентов онлайн:
https://riptutorial.com/ru/reactjs/topic/2750/реагировать-жизненный-цикл-компонентов

глава 19: Реагировать на вызов AJAX

Examples

Запрос HTTP GET

Иногда компонент должен отображать некоторые данные с удаленной конечной точки (например, REST API). Стандартная практика заключается в том, чтобы делать такие вызовы в методе `componentDidMount` .

Ниже приведен пример использования суперагента в качестве помощника AJAX:

```
import React from 'react'
import request from 'superagent'

class App extends React.Component {
  constructor () {
    super()
    this.state = {}
  }
  componentDidMount () {
    request
      .get('/search')
      .query({ query: 'Manny' })
      .query({ range: '1..5' })
      .query({ order: 'desc' })
      .set('API-Key', 'foobar')
      .set('Accept', 'application/json')
      .end((err, resp) => {
        if (!err) {
          this.setState({someData: resp.text})
        }
      })
  },
  render() {
    return (
      <div>{this.state.someData || 'waiting for response...'}</div>
    )
  }
}

React.render(<App />, document.getElementById('root'))
```

Запрос может быть инициирован путем вызова соответствующего метода в объекте `request` , а затем вызова `.end()` для отправки запроса. Установка полей заголовка проста, вызывается `.set()` с именем и значением поля.

Метод `.query()` принимает объекты, которые при использовании с методом GET образуют строку запроса. Далее будет `/search?query=Manny&range=1..5&order=desc` путь `/search?query=Manny&range=1..5&order=desc` .

Запросы **POST**

```
request.post('/user')
  .set('Content-Type', 'application/json')
  .send('{"name":"tj","pet":"tobi"}')
  .end(callback)
```

Дополнительную информацию см. В документах Superagent .

Ajax in React без сторонней библиотеки - он же с VanillaJS

Следующие действия будут работать в IE9 +

```
import React from 'react'

class App extends React.Component {
  constructor () {
    super()
    this.state = {someData: null}
  }
  componentDidMount () {
    var request = new XMLHttpRequest();
    request.open('GET', '/my/url', true);

    request.onload = () => {
      if (request.status >= 200 && request.status < 400) {
        // Success!
        this.setState({someData: request.responseText})
      } else {
        // We reached our target server, but it returned an error
        // Possibly handle the error by changing your state.
      }
    };

    request.onerror = () => {
      // There was a connection error of some sort.
      // Possibly handle the error by changing your state.
    };

    request.send();
  },
  render() {
    return (
      <div>{this.state.someData || 'waiting for response...'}</div>
    )
  }
}

React.render(<App />, document.getElementById('root'))
```

HTTP GET-запрос и обработка данных

В следующем примере показано, как набор данных, полученных из удаленного источника, может быть отображен в компонент.

Мы делаем запрос AJAX с использованием `fetch` , который встроен в большинство браузеров. Используйте полис для `fetch` чтобы поддерживать старые браузеры. Вы также можете использовать любую другую библиотеку для запросов (например, axios , SuperAgent или даже простой Javascript).

Мы устанавливаем данные, которые мы получаем как состояние компонента, поэтому мы можем получить доступ к нему внутри метода рендеринга. Там мы просматриваем данные с помощью `map` . Не забудьте всегда добавлять уникальный атрибут `key` (или прокрутку) к элементу с петлями, что важно для производительности рендеринга React.

```
import React from 'react';

class Users extends React.Component {
  constructor() {
    super();
    this.state = { users: [] };
  }

  componentDidMount() {
    fetch('/api/users')
      .then(response => response.json())
      .then(json => this.setState({ users: json.data }));
  }

  render() {
    return (
      <div>
        <h1>Users</h1>
        {
          this.state.users.length == 0
            ? 'Loading users...'
            : this.state.users.map(user => (
              <figure key={user.id}>
                <img src={user.avatar} />
                <figcaption>
                  {user.name}
                </figcaption>
              </figure>
            ))
        }
      </div>
    );
  }
}

ReactDOM.render(<Users />, document.getElementById('root'));
```

Рабочий пример в JSBin .

Прочитайте Реагировать на вызов AJAX онлайн: https://riptutorial.com/ru/reactjs/topic/6432/реагировать-на-вызов-ajax

глава 20: Реагировать на котел [React + Babel + Webpack]

Examples

Настройка проекта

Для установки зависимостей проекта необходим диспетчер пакетов узлов. Загрузите узел для своей операционной системы из Nodejs.org . Диспетчер пакетов узлов имеет узел.

Вы также можете использовать Node Version Manager для лучшего управления версиями узлов и версий npm. Это отлично подходит для тестирования вашего проекта на разных версиях узлов. Однако это не рекомендуется для производственной среды.

После того, как вы установили узел в своей системе, пойдите и установите некоторые необходимые пакеты, чтобы взорвать свой первый проект React с помощью Babel и Webpack.

Прежде чем мы на самом деле начнем ударять команды в терминале. Посмотрите, для чего используются Babel и Webpack .

Вы можете запустить свой проект, выполнив `npm init` в своем терминале. Следуйте первоначальной настройке. После этого выполните следующие команды в терминале:

зависимости:

```
npm install react react-dom --save
```

Dev Dependecies:

```
npm install babel-core babel-loader babel-preset-es2015 babel-preset-react babel-preset-stage-0
webpack webpack-dev-server react-hot-loader --save-dev
```

Дополнительные зависимости Dev:

```
npm install eslint eslint-plugin-react babel-eslint --save-dev
```

Вы можете обратиться к этому образцу package.json

Создайте `.babelrc` в корне вашего проекта со следующим содержимым:

```
{
  "presets": ["es2015", "stage-0", "react"]
}
```

При необходимости создайте `.eslintrc` в корне вашего проекта со следующим содержимым:

```
{
  "ecmaFeatures": {
    "jsx": true,
    "modules": true
  },
  "env": {
    "browser": true,
    "node": true
  },
  "parser": "babel-eslint",
  "rules": {
    "quotes": [2, "single"],
    "strict": [2, "never"],
  },
  "plugins": [
    "react"
  ]
}
```

Создайте файл `.gitignore` чтобы предотвратить загрузку сгенерированных файлов в `.gitignore` git.

```
node_modules
npm-debug.log
.DS_Store
dist
```

Создайте файл `webpack.config.js` со следующим минимальным содержимым.

```
var path = require('path');
var webpack = require('webpack');

module.exports = {
  devtool: 'eval',
  entry: [
    'webpack-dev-server/client?http://localhost:3000',
    'webpack/hot/only-dev-server',
    './src/index'
  ],
  output: {
    path: path.join(__dirname, 'dist'),
    filename: 'bundle.js',
    publicPath: '/static/'
  },
  plugins: [
    new webpack.HotModuleReplacementPlugin()
  ],
  module: {
    loaders: [{
      test: /\.js$/,
      loaders: ['react-hot', 'babel'],
      include: path.join(__dirname, 'src')
    }]
  }
};
```

И, наконец, создайте файл `sever.js` чтобы запустить `npm start` со следующим содержимым:

```
var webpack = require('webpack');
var WebpackDevServer = require('webpack-dev-server');
var config = require('./webpack.config');

new WebpackDevServer(webpack(config), {
  publicPath: config.output.publicPath,
  hot: true,
  historyApiFallback: true
}).listen(3000, 'localhost', function (err, result) {
  if (err) {
    return console.log(err);
  }

  console.log('Serving your awesome project at http://localhost:3000/');
});
```

Создайте файл `src/app.js` чтобы увидеть, как проект React что-то делает.

```
import React, { Component } from 'react';

export default class App extends Component {
  render() {
    return (
      <h1>Hello, world.</h1>
    );
  }
}
```

Запустите `node server.js` или `npm start` в терминале, если вы определили, что означает `start` в вашем `package.json`

проект реагирования-стартера

Об этом проекте

Это простой шаблонный проект. Этот пост поможет вам настроить среду для ReactJs + Webpack + Bable.

Давайте начнем

нам понадобится менеджер пакетов узлов для запуска экспресс-сервера и управления зависимостями во всем проекте. если вы новичок в диспетчере пакетов узлов, вы можете проверить здесь . Примечание: здесь требуется установка диспетчера пакетов узлов.

Создайте папку с подходящим именем и перейдите в нее с терминала или с помощью GUI. Затем перейдите к терминалу и введите `npm init` это создаст файл package.json, ничего страшного, он задаст вам несколько вопросов, таких как название вашего проекта, описание, точка входа, репозиторий git, автор, лицензия и т. д. Здесь точка входа важна, потому что узел будет сначала искать ее при запуске проекта. В конце он попросит вас проверить предоставленную вами информацию. Вы можете ввести *да* или изменить его. Ну вот и все, наш файл *package.json* готов.

Экспресс-установка сервера запуска *npm install express @ 4 --save* . Это все зависимости, которые нам нужны для этого проекта. Сохранить флаг важен, без него файл *package.js* не будет обновляться. Основная задача *package.json* - хранить список зависимостей. Он добавит экспресс-версию 4. Ваш *package.json* будет выглядеть как `"dependencies": { "express": "^4.13.4", ..... },`

После полной загрузки вы можете увидеть папку *node_modules* и подпапку наших зависимостей. Теперь в корне проекта создайте новый файл *server.js* . Теперь мы устанавливаем экспресс-сервер. Я собираюсь пройти весь код и объяснить его позже.

```
var express = require('express');
// Create our app
var app = express();

app.use(express.static('public'));

app.listen(3000, function () {
  console.log('Express server is using port:3000');
});
```

var express = require ('express'); это даст вам доступ к полной экспресс-апи.

var app = express (); вызовет функцию экспресс-библиотеки как функцию. *app.use ();* добавьте функциональность в свое экспресс-приложение. *app.use (express.static (' общественного'));* будет указывать имя папки, которое будет отображаться на нашем веб-сервере. *app.listen (port, function () {})* будет здесь, наш порт будет *3000,* и функция, которую мы вызываем, проверит, что веб-сервер работает правильно. Именно это настроен сервер.

Теперь перейдите в наш проект и создайте новую папку общедоступным и создайте файл *index.html* . *index.html* - это файл по умолчанию для вашего приложения, и Express-сервер будет искать этот файл. *Index.html* - простой html-файл, который выглядит

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="UTF-8"/>
</head>

<body>
  <h1>hello World</h1>
</body>

</html>
```

И перейдите к пути проекта через терминал и введите *node server.js* . Затем вы увидите * console.log («Экспресс-сервер использует порт: 3000»); *.

Перейдите в браузер и введите *http: // localhost: 3000* в навигационной панели, где увидите *мир приветствия* .

Теперь зайдите в *общую* папку и создайте новый файл *app.jsx* . JSX - это шаг препроцессора, который добавляет синтаксис XML к вашему JavaScript. Вы можете определенно использовать React без JSX, но JSX делает React намного более элегантным. Вот пример кода для *app.jsx*

```
ReactDOM.render(
  <h1>Hello World!!!</h1>,
  document.getElementById('app')
);
```

Теперь перейдите в *index.html* и измените код, он должен выглядеть так:

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="UTF-8"/>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23
/browser.min.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.7/react.js">
</script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.7/react-dom.js">    </script>
</head>

<body>
  <div id="app"></div>

  <script type="text/babel" src="app.jsx"></script>
</body>

</html>
```

С этим на месте вы все закончили, надеюсь, вы найдете это простым.

Прочитайте Реагировать на котел [React + Babel + Webpack] онлайн: https://riptutorial.com/ru/reactjs/topic/5969/реагировать-на-котел--react-plus-babel-plus-webpack-

глава 21: Реагировать на формы

Examples

Контролируемые компоненты

Управляемый компонент привязан к значению, и его изменения обрабатываются кодом, используя обратные вызовы, основанные на событиях.

```
class CustomForm extends React.Component {
constructor() {
    super();
    this.state = {
        person: {
            firstName: '',
            lastName: ''
        }
    }
}

handleChange(event) {
    let person = this.state.person;
    person[event.target.name] = event.target.value;
    this.setState({person});
}

render() {
    return (
        <form>
            <input
            type="text"
            name="firstName"
            value={this.state.firstName}
            onChange={this.handleChange.bind(this)} />

            <input
            type="text"
            name="lastName"
            value={this.state.lastName}
            onChange={this.handleChange.bind(this)} />
        </form>
    )
}
```

}

В этом примере мы инициализируем состояние с пустым человеческим объектом. Затем мы связываем значения двух входов с отдельными ключами объекта person. Затем, когда пользователь вводит данные, мы `handleChange` каждое значение в функции `handleChange` . Поскольку значения компонентов привязаны к состоянию, мы можем повторно получить, как пользователь вводит, вызывая `setState()` .

ПРИМЕЧАНИЕ. Не вызов `setState()` при работе с контролируемыми компонентами приведет к тому, что пользователь напечатает, но не увидит вход, потому что React только изменяет, когда ему говорят об этом.

Также важно отметить, что имена входов такие же, как имена ключей в объекте person. Это позволяет нам фиксировать значение в форме словаря, как показано здесь.

```
handleChange(event) {
    let person = this.state.person;
    person[event.target.name] = event.target.value;
    this.setState({person});
}
```

`person[event.target.name]` - это тот же самый `person.firstName || person.lastName` . Конечно, это будет зависеть от того, какой вход вводится в настоящее время. Поскольку мы не знаем, где пользователь будет печатать, используя словарь и сопоставляя имена ввода с именами ключей, мы можем зафиксировать ввод пользователя по вопрос, из которого вызывается onChange.

Прочитайте Реагировать на формы онлайн: https://riptutorial.com/ru/reactjs/topic/8047/реагировать-на-формы

глава 22: Реагировать с Redux

Вступление

Redux стал статус-кво для управления состоянием на уровне приложений в интерфейсе в наши дни, а те, кто работает над «крупномасштабными приложениями», часто клянутся им. В этом разделе рассказывается, почему и как вы должны использовать библиотеку управления государством, Redux, в своих приложениях React.

замечания

В то время как архитектура, основанная на компонентах React, является фантастикой для разбивки приложения на модульные, инкапсулированные маленькие кусочки, это создает некоторые проблемы для управления состоянием приложения в целом. Время использования Redux - это когда вам нужно отображать одни и те же данные более чем на одном компоненте или странице (иначе это маршрут). В этот момент вы больше не сможете хранить данные в переменных, локальных для одного компонента или другого, и отправка сообщений между компонентами быстро становится беспорядочным. С Redux ваши компоненты все подписываются на одни и те же общие данные в магазине, и, таким образом, состояние может быть легко отражено последовательно во всем приложении.

Examples

Использование Connect

Создайте хранилище Redux с помощью *createStore* .

```
import { createStore } from 'redux'
import todoApp from './reducers'
let store = createStore(todoApp, { inistialStateVariable: "derp"})
```

Используйте *соединение* для подключения компонента к хранилищу Redux и вытащите реквизиты из хранилища в компонент.

```
import { connect } from 'react-redux'

const VisibleTodoList = connect(
  mapStateToProps,
  mapDispatchToProps
)(TodoList)

export default VisibleTodoList
```

Определите действия, которые позволяют вашим компонентам отправлять сообщения в

хранилище Redux.

```
/*
 * action types
 */

export const ADD_TODO = 'ADD_TODO'

export function addTodo(text) {
  return { type: ADD_TODO, text }
}
```

Обработайте эти сообщения и создайте новое состояние для хранилища в функциях редуктора.

```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    default:
      return state
  }
}
```

Прочитайте Реагировать с Redux онлайн: https://riptutorial.com/ru/reactjs/topic/10856/реагировать-с-redux

глава 23: Реагирующие инструменты

Examples

связи

Места для поиска компонентов и библиотек React;

- Каталог реагентов
- JS.coach

Прочитайте Реагирующие инструменты онлайн: https://riptutorial.com/ru/reactjs/topic/6595/реагирующие-инструменты

глава 24: Реакция маршрутизации

Examples

Пример файла Routes.js, за которым следует использование Router Link в компоненте

Поместите файл, как показано ниже, в каталог верхнего уровня. Он определяет, какие компоненты отображать для каких путей

```
import React from 'react';
import { Route, IndexRoute } from 'react-router';
import New from './containers/new-post';
import Show from './containers/show';

import Index from './containers/home';
import App from './components/app';

export default(
  <Route path="/" component={App}>
    <IndexRoute component={Index} />
    <Route path="posts/new" component={New} />
    <Route path="posts/:id" component={Show} />

  </Route>
);
```

Теперь в вашем верхнем уровне index.js, который является вашей точкой входа в приложение, вам нужно просто отобразить этот компонент Router следующим образом:

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Router, browserHistory } from 'react-router';
// import the routes component we created in routes.js
import routes from './routes';


// entry point
ReactDOM.render(
    <Router history={browserHistory} routes={routes} />
  , document.getElementById('main'));
```

Теперь это просто вопрос использования `Link` вместо тегов `<a>` в вашем приложении. Использование Link свяжется с React Router, чтобы изменить маршрут React Router на указанную ссылку, которая, в свою очередь, отобразит правильный компонент, определенный в route.js

```
import React from 'react';
```

```
import { Link } from 'react-router';

export default function PostButton(props) {
  return (
    <Link to={`posts/${props.postId}`}>
      <div className="post-button" >
        {props.title}
        <span>{props.tags}</span>
      </div>
    </Link>
  );
}
```

React Routing Async

```
import React from 'react';
import { Route, IndexRoute } from 'react-router';

import Index from './containers/home';
import App from './components/app';

//for single Component lazy load use this
const ContactComponent = () => {
  return {
      getComponent: (location, callback)=> {
        require.ensure([], require => {
          callback(null, require('./components/Contact')["default"]);
        }, 'Contact');
      }
    }
};

//for multiple componnets
 const groupedComponents = (pageName) => {
  return {
      getComponent: (location, callback)=> {
        require.ensure([], require => {
          switch(pageName){
            case 'about' :
                callback(null, require( "./components/about" )["default"]);
                    break ;
            case 'tos' :
                callback(null, require( "./components/tos" )["default"]);
                    break ;
          }
        }, "groupedComponents");
      }
    }
};
export default(
  <Route path="/" component={App}>
    <IndexRoute component={Index} />
    <Route path="/contact" {...ContactComponent()} />
    <Route path="/about" {...groupedComponents('about')} />
    <Route path="/tos" {...groupedComponents('tos')} />
  </Route>
);
```

Прочитайте Реакция маршрутизации онлайн: https://riptutorial.com/ru/reactjs/topic/6096/

реакция-маршрутизации

глава 25: Реквизит в действии

замечания

ПРИМЕЧАНИЕ. Что касается React 15.5, и компонент PropTypes живет в своем собственном пакете npm, а именно: «prop-types» и ему нужен собственный оператор импорта при использовании PropTypes. См. Официальную документацию о реакции на изменение: https://facebook.github.io/react/blog/2017/04/07/react-v15.5.0.html

Examples

Вступление

`props` используются для передачи данных и методов из родительского компонента в дочерний компонент.

Интересные вещи о `props`

1. Они неизменны.
2. Они позволяют нам создавать повторно используемые компоненты.

Основной пример

```
class Parent extends React.Component{
  doSomething(){
     console.log("Parent component");
  }
  render() {
    return <div>
         <Child
           text="This is the child number 1"
           title="Title 1"
           onClick={this.doSomething} />
         <Child
           text="This is the child number 2"
           title="Title 2"
           onClick={this.doSomething} />
      </div>
  }
}

class Child extends React.Component{
  render() {
    return <div>
       <h1>{this.props.title}</h1>
       <h2>{this.props.text}</h2>
      </div>
  }
```

```
}
```

Как вы можете видеть в этом примере, благодаря `props` мы можем создавать повторно используемые компоненты.

Репозитории по умолчанию

`defaultProps` позволяет установить по умолчанию, или запасной вариант, значения для компонента `props` . `defaultProps` полезны, когда вы вызываете компоненты из разных представлений с фиксированными реквизитами, но в некоторых представлениях вам нужно передать другое значение.

Синтаксис

ES5

```
var MyClass = React.createClass({
  getDefaultProps: function() {
    return {
      randomObject: {},
      ...
    };
  }
}
```

ES6

```
class MyClass extends React.Component {...}

MyClass.defaultProps = {
    randomObject: {},
    ...
}
```

ES7

```
class MyClass extends React.Component {
    static defaultProps = {
        randomObject: {},
        ...
    };
}
```

Результат `getDefaultProps()` или `defaultProps` будет кэшироваться и использоваться для обеспечения того, что `this.props.randomObject` будет иметь значение, если он не был указан родительским компонентом.

PropTypes

`propTypes` позволяет указать, какие `props` нужны вашему компоненту и какой тип они должны быть. Ваш компонент будет работать без установки `propTypes` , но это хорошая практика, чтобы определить их, поскольку это сделает ваш компонент более читабельным, станет документом для других разработчиков, которые читают ваш компонент, а во время разработки React предупредит вас, если вы попытаетесь установите опору, которая отличается от определения, которое вы установили для него.

Некоторые примитивные `propTypes` и обычно используемые `propTypes` -

```
  optionalArray: React.PropTypes.array,
  optionalBool: React.PropTypes.bool,
  optionalFunc: React.PropTypes.func,
  optionalNumber: React.PropTypes.number,
  optionalObject: React.PropTypes.object,
  optionalString: React.PropTypes.string,
  optionalSymbol: React.PropTypes.symbol
```

Если вы прикрепляете `isRequired` к любому `propType` тогда эта поддержка должна быть предоставлена при создании экземпляра этого компонента. Если вы не предоставите **требуемые** `propTypes` то экземпляр компонента не может быть создан.

Синтаксис

ES5

```
var MyClass = React.createClass({
  propTypes: {
    randomObject: React.PropTypes.object,
    callback: React.PropTypes.func.isRequired,
    ...
  }
}
```

ES6

```
class MyClass extends React.Component {...}

MyClass.propTypes = {
    randomObject: React.PropTypes.object,
    callback: React.PropTypes.func.isRequired,
    ...
};
```

ES7

```
class MyClass extends React.Component {
     static propTypes = {
        randomObject: React.PropTypes.object,
```

```
        callback: React.PropTypes.func.isRequired,
        ...
    };
}
```

Более сложная проверка реквизита

Точно так же `PropTypes` позволяет указать более сложную проверку

Проверка объекта

```
...
    randomObject: React.PropTypes.shape({
        id: React.PropTypes.number.isRequired,
        text: React.PropTypes.string,
    }).isRequired,
...
```

Проверка массива объектов

```
...
    arrayOfObjects: React.PropTypes.arrayOf(React.PropTypes.shape({
        id: React.PropTypes.number.isRequired,
        text: React.PropTypes.string,
    })).isRequired,
...
```

Передача реквизита с помощью оператора спредов

Вместо

```
var component = <Component foo={this.props.x} bar={this.props.y} />;
```

Если каждое свойство должно быть передано в качестве единственного значения поддержки, вы можете использовать оператор спреда `...` поддерживаемый для массивов в ES6 для передачи всех ваших значений. Теперь компонент будет выглядеть следующим образом.

```
var component = <Component {...props} />;
```

Помните, что свойства объекта, который вы передаете, копируются на реквизиты компонента.

Порядок важен. Более поздние атрибуты переопределяют предыдущие.

```
var props = { foo: 'default' };
var component = <Component {...props} foo={'override'} />;
console.log(component.props.foo); // 'override'
```

Другим случаем является то, что вы также можете использовать оператор спреда для передачи только частей реквизита для дочерних компонентов, тогда вы можете снова использовать синтаксис деструкции из реквизита.

Это очень полезно, когда компоненты для детей нуждаются в большом количестве реквизитов, но не хотят передавать их по одному.

```
const { foo, bar, other } = this.props // { foo: 'foo', bar: 'bar', other: 'other' };
var component = <Component {...{foo, bar}} />;
const { foo, bar } = component.props
console.log(foo, bar); // 'foo bar'
```

Состав реквизита и состав компонентов

« `props.children` » компоненты компонента доступны на специальной опоре, `props.children` . Эта пропозиция очень полезна для компонентов «Композиция» и может сделать разметку JSX более интуитивной или отражать предполагаемую окончательную структуру DOM:

```
var SomeComponent = function () {
    return (
        <article className="textBox">
            <header>{this.props.heading}</header>
            <div className="paragraphs">
                {this.props.children}
            </div>
        </article>
    );
}
```

Это позволяет нам включать произвольное количество подэлементов при использовании компонента позже:

```
var ParentComponent = function () {
    return (
        <SomeComponent heading="Amazing Article Box" >
            <p className="first"> Lots of content </p>
            <p> Or not </p>
        </SomeComponent>
    );
}
```

Удостоверения могут также управляться компонентом. Поскольку props.children может быть или не быть массивом , React предоставляет служебные функции для них как React.Children . Рассмотрим в предыдущем примере, если бы мы захотели обернуть каждый абзац в свой собственный элемент `<section>` :

```
var SomeComponent = function () {
    return (
        <article className="textBox">
            <header>{this.props.heading}</header>
            <div className="paragraphs">
```

```
                {React.Children.map(this.props.children, function (child) {
                    return (
                        <section className={child.props.className}>
                            React.cloneElement(child)
                        </section>
                    );
                })}
            </div>
        </article>
    );
}
```

Обратите внимание на использование React.cloneElement для удаления реквизита из дочернего `<p>` поскольку реквизиты неизменяемы, эти значения не могут быть изменены напрямую. Вместо этого нужно использовать клон без этих реквизитов.

Кроме того, при добавлении элементов в циклы, знайте, как React примиряет детей во время rerender , и настоятельно рекомендуем включить глобально уникальную подсказку `key` для дочерних элементов, добавленных в цикл.

Определение типа компонентов для детей

Иногда очень полезно знать тип дочернего компонента при повторении через них. Чтобы выполнить итерацию через дочерние компоненты, вы можете использовать функцию React `Children.map` util:

```
React.Children.map(this.props.children, (child) => {
  if (child.type === MyComponentType) {
    ...
  }
});
```

Детский объект предоставляет свойство `type` которое можно сравнить с конкретным компонентом.

Прочитайте Реквизит в действии онлайн: https://riptutorial.com/ru/reactjs/topic/2749/реквизит-в-действии

глава 26: Решения для пользовательского интерфейса

Вступление

Предположим, мы вдохновляемся некоторыми идеями из современных пользовательских интерфейсов, используемых в программах, и преобразуем их в компоненты React. Вот что такое тема « **Решения для пользовательских интерфейсов** ». Атрибуция описана.

Examples

Основная панель

```
import React from 'react';

class Pane extends React.Component {
    constructor(props) {
        super(props);
    }

    render() {
        return React.createElement(
            'section', this.props
        );
    }
}
```

панель

```
import React from 'react';

class Panel extends React.Component {
    constructor(props) {
        super(props);
    }

    render(...elements) {
        var props = Object.assign({
            className: this.props.active ? 'active' : '',
            tabIndex: -1
        }, this.props);

        var css = this.css();
        if (css != '') {
            elements.unshift(React.createElement(
                'style', null,
                css
            ));
        }
```

```
        return React.createElement(
            'div', props,
            ...elements
        );
    }

    static title() {
        return '';
    }
    static css() {
        return '';
    }
}
```

Основные отличия от простой панели:

- панель имеет фокус, например, когда она вызывается сценарием или нажата мышью;
- панель имеет статический метод `title` на компонент, поэтому он может быть расширен другим компонентом панели с переопределенным `title` (причина в том, что функция может быть снова вызвана при рендеринге для целей локализации, но в рамках этого примера `title` не имеет смысла) ;
- он может содержать индивидуальную таблицу стилей, объявленную в статическом методе `css` (вы можете предварительно загрузить содержимое файла из `PANEL.css`).

табуляция

```
import React from 'react';

class Tab extends React.Component {
    constructor(props) {
        super(props);
    }

    render() {
        var props = Object.assign({
            className: this.props.active ? 'active' : ''
        }, this.props);
        return React.createElement(
            'li', props,
            React.createElement(
                'span', props,
                props.panelClass.title()
            )
        );
    }
}
```

свойство `panelClass` экземпляра `Tab` должно содержать класс *панели,* используемый для описания.

PanelGroup

```
import React from 'react';
import Tab from './Tab.js';

class PanelGroup extends React.Component {
    constructor(props) {
        super(props);
        this.setState({
            panels: props.panels
        });
    }

    render() {
        this.tabSet = [];
        this.panelSet = [];
        for (let panelData of this.state.panels) {
            var tabIsActive = this.state.activeTab == panelData.name;
            this.tabSet.push(React.createElement(
                Tab, {
                    name: panelData.name,
                    active: tabIsActive,
                    panelClass: panelData.class,
                    onMouseDown: () => this.openTab(panelData.name)
                }
            ));
            this.panelSet.push(React.createElement(
                panelData.class, {
                    id: panelData.name,
                    active: tabIsActive,
                    ref: tabIsActive ? 'activePanel' : null
                }
            ));
        }
        return React.createElement(
            'div', { className: 'PanelGroup' },
            React.createElement(
                'nav', null,
                React.createElement(
                    'ul', null,
                    ...this.tabSet
                )
            ),
            ...this.panelSet
        );
    }

    openTab(name) {
        this.setState({ activeTab: name });
        this.findDOMNode(this.refs.activePanel).focus();
    }
}
```

`panels` свойство `PanelGroup` экземпляра должно содержать массив объектов. Каждый объект там объявляет важные данные о панелях:

- **`name`** - идентификатор панели, используемой скриптом контроллера;
- **`class`** - **`class`** панели.

Не забудьте указать свойство `activeTab` для имени нужной вкладки.

осветление

Когда вкладка не указана, необходимая панель получает имя класса `active` в элементе DOM (означает, что он будет видимым), и теперь он сфокусирован.

Примерный вид с панелью PanelGroup

```
import React from 'react';
import Pane from './components/Pane.js';
import Panel from './components/Panel.js';
import PanelGroup from './components/PanelGroup.js';

class MainView extends React.Component {
    constructor(props) {
        super(props);
    }

    render() {
        return React.createElement(
            'main', null,
            React.createElement(
                Pane, { id: 'common' },
                React.createElement(
                    PanelGroup, {
                        panels: [
                            {
                                name: 'console',
                                panelClass: ConsolePanel
                            },
                            {
                                name: 'figures',
                                panelClass: FiguresPanel
                            }
                        ],
                        activeTab: 'console'
                    }
                )
            ),
            React.createElement(
                Pane, { id: 'side' },
                React.createElement(
                    PanelGroup, {
                        panels: [
                            {
                                name: 'properties',
                                panelClass: PropertiesPanel
                            }
                        ],
                        activeTab: 'properties'
                    }
                )
            )
        );
    }
}

class ConsolePanel extends Panel {
```

```
    constructor(props) {
        super(props);
    }

    static title() {
        return 'Console';
    }
}

class FiguresPanel extends Panel {
    constructor(props) {
        super(props);
    }

    static title() {
        return 'Figures';
    }
}

class PropertiesPanel extends Panel {
    constructor(props) {
        super(props);
    }

    static title() {
        return 'Properties';
    }
}
```

Прочитайте Решения для пользовательского интерфейса онлайн:
https://riptutorial.com/ru/reactjs/topic/8112/решения-для-пользовательского-интерфейса

глава 27: Связь между компонентами

замечания

Всего существует 3 случая связи между компонентами React:

- Случай 1: общение от родителей к ребенку
- Случай 2: общение от ребенка к родителям
- Случай 3: Не связанные компоненты (любой компонент для любого компонента)

Examples

Компоненты от родителя к ребенку

Что самый простой случай на самом деле, очень естественный в мире React, и шансы - вы уже используете его.

Вы можете **передать реквизиты до дочерних компонентов** . В этом примере `message` является опорой, которую мы передаем дочернему компоненту, имя сообщения выбрано произвольно, вы можете называть его всем, что хотите.

```
import React from 'react';

class Parent extends React.Component {
    render() {
        const variable = 5;
        return (
            <div>
                <Child message="message for child" />
                <Child message={variable} />
            </div>
        );
    }
}

class Child extends React.Component {
    render() {
        return <h1>{this.props.message}</h1>
    }
}

export default Parent;
```

Здесь компонент `<Parent />` отображает два компонента `<Child />` , передавая `message for child` элемента внутри первого компонента и `5` внутри второго.

Итак, у вас есть компонент (родительский), который отображает еще один (дочерний) и передает ему некоторые реквизиты.

Ребенок к родительским компонентам

Отправка данных обратно родительскому, для этого мы просто **передаем функцию в качестве опоры из родительского компонента в дочерний компонент** , а **дочерний компонент вызывает эту функцию** .

В этом примере мы изменим состояние родителя, передав функцию дочернему компоненту и вызывая эту функцию внутри дочернего компонента.

```
import React from 'react';

class Parent extends React.Component {
    constructor(props) {
        super(props);
        this.state = { count: 0 };

        this.outputEvent = this.outputEvent.bind(this);
    }
    outputEvent(event) {
        // the event context comes from the Child
        this.setState({ count: this.state.count++ });
    }

    render() {
        const variable = 5;
        return (
            <div>
                Count: { this.state.count }
                <Child clickHandler={this.outputEvent} />
            </div>
        );
    }
}

class Child extends React.Component {
    render() {
        return (
            <button onClick={this.props.clickHandler}>
                Add One More
            </button>
        );
    }
}

export default Parent;
```

Обратите внимание, что метод родительского метода `outputEvent` (который изменяет состояние родителя) вызывается `outputEvent` кнопкой `onClick` .

Не связанные компоненты

Единственный способ, если ваши компоненты не имеют отношения родитель-ребенок (или связаны друг с другом, но слишком дальше, например, гранд-гроссмейстер), должен иметь какой-то сигнал, который один компонент подписывает, а другой записывает.

Это две основные операции любой системы событий: **подписаться** / **прослушать** событие, которое будет уведомляться, и **отправить** / **запустить** / **опубликовать** / **отправить** событие, чтобы уведомить тех, кто хочет.

Для этого есть как минимум 3 шаблона. Вы можете найти сравнение здесь .

Вот краткое резюме:

- Шаблон 1: **Излучатель событий** / **Целевой** / **Диспетчер** : слушатели должны ссылаться на источник для подписки.
    - подписаться: `otherObject.addEventListener('click', () => { alert('click!'); });`
    - для отправки: `this.dispatchEvent('click');`
- Шаблон 2: **Публикация** / **Подписка** : вам не нужна конкретная ссылка на источник, который запускает событие, есть глобальный объект, доступный везде, который обрабатывает все события.
    - подписаться: `globalBroadcaster.subscribe('click', () => { alert('click!'); });`
    - для отправки: `globalBroadcaster.publish('click');`
- Шаблон 3: **Сигналы** : похожие на Event Emmit / Target / Dispatcher, но здесь вы не используете никаких случайных строк. Каждый объект, который может генерировать события, должен иметь определенное свойство с этим именем. Таким образом, вы точно знаете, какие события могут излучать объекты.
    - подписаться: `otherObject.clicked.add( () => { alert('click'); });`
    - отправить: `this.clicked.dispatch();`

Прочитайте Связь между компонентами онлайн: https://riptutorial.com/ru/reactjs/topic/6567/связь-между-компонентами

глава 28: Состояние в действии

Examples

Основное состояние

Компоненты State in React необходимы для управления и передачи данных в вашем приложении. Он представлен как объект JavaScript и обладает областью *уровня компонента* , его можно рассматривать как частные данные вашего компонента.

В приведенном ниже примере мы определяем некоторое начальное состояние в `constructor` функции нашего компонента и используем его в функции `render` .

```
class ExampleComponent extends React.Component {
  constructor(props){
    super(props);

    // Set-up our initial state
    this.state = {
      greeting: 'Hiya Buddy!'
    };
  }

  render() {
    // We can access the greeting property through this.state
    return(
      <div>{this.state.greeting}</div>
    );
  }
}
```

SetState ()

Основной способ, которым вы делаете обновления пользовательского интерфейса для своих приложений React, - это вызов функции `setState()` . Эта функция будет выполнять *мелкое слияние* между новым состоянием, которое вы предоставляете, и предыдущим состоянием, и вызовет повторную визуализацию вашего компонента и всех его дециентов.

параметры

1. `updater` : Это может быть объект с несколькими парами ключ-значение, которые должны быть объединены в состояние или функцию, возвращающую такой объект.
2. `callback (optional)` : функция, которая будет выполнена после того, как `setState()` выполнена успешно. Из-за того, что `setState()` не гарантируется, что React является атомарным, иногда это может быть полезно, если вы хотите выполнить какое-либо действие после того, как вы `setState()` что `setState()` успешно выполнен.

Использование:

`setState` метод принимает программу `updater` аргумента , который может быть либо объектом с рядом ключом-значением-пар , которые должны быть объединены в состояние, или функцией , которая возвращает такой объект , вычисленный из `prevState` и `props` .

Использование `setState()` с объектом как средство `updater`

```
//
// An example ES6 style component, updating the state on a simple button click.
// Also demonstrates where the state can be set directly and where setState should be used.
//
class Greeting extends React.Component {
    constructor(props) {
        super(props);
        this.click = this.click.bind(this);
        // Set initial state (ONLY ALLOWED IN CONSTRUCTOR)
        this.state = {
            greeting: 'Hello!'
        };
    }
    click(e) {
        this.setState({
              greeting: 'Hello World!'
        });
    }
    render() {
        return(
            <div>
                <p>{this.state.greeting}</p>
                <button onClick={this.click}>Click me</button>
            </div>
        );
    }

}
```

Использование `setState()` с функцией в качестве средства `updater`

```
//
// This is most often used when you want to check or make use
// of previous state before updating any values.
//

this.setState(function(previousState, currentProps) {
  return {
    counter: previousState.counter + 1
  };
});
```

Это может быть безопаснее, чем использование аргумента объекта, в котором используются множественные вызовы для `setState()` , поскольку несколько вызовов могут быть собраны вместе с помощью React и выполняться сразу, и это предпочтительный подход при использовании текущих реквизитов для установки состояния.

```
this.setState({ counter: this.state.counter + 1 });
this.setState({ counter: this.state.counter + 1 });
this.setState({ counter: this.state.counter + 1 });
```

Эти вызовы могут быть объединены путем React с использованием `Object.assign()` , в результате чего счетчик увеличивается на 1, а не на 3.

Функциональный подход также может использоваться для перемещения логики настройки состояния вне компонентов. Это позволяет изолировать и повторно использовать логику состояния.

```
// Outside of component class, potentially in another file/module

function incrementCounter(previousState, currentProps) {
    return {
        counter: previousState.counter + 1
    };
}

// Within component

this.setState(incrementCounter);
```

Вызов функции `setState()` с объектом и функцией обратного вызова

```
//
// 'Hi There' will be logged to the console after setState completes
//

this.setState({ name: 'John Doe' }, console.log('Hi there'));
```

Общий антипаттерн

Вы не должны сохранять `props` в `state` . Он считается анти-шаблоном . Например:

```
export default class MyComponent extends React.Component {
    constructor() {
        super();

        this.state = {
            url: ''
        }
```

```
        this.onChange = this.onChange.bind(this);
    }

    onChange(e) {
        this.setState({
            url: this.props.url + '/days=?' + e.target.value
        });
    }

    componentWillMount() {
        this.setState({url: this.props.url});
    }

    render() {
        return (
            <div>
                <input defaultValue={2} onChange={this.onChange} />

                URL: {this.state.url}
            </div>
        )
    }
}
```

`url` prop сохраняется в `state` а затем изменен. Вместо этого выберите сохранение изменений в состоянии, а затем постройте полный путь, используя как `state` и `props` :

```
export default class MyComponent extends React.Component {
    constructor() {
        super();

        this.state = {
            days: ''
        }

        this.onChange = this.onChange.bind(this);
    }

    onChange(e) {
        this.setState({
            days: e.target.value
        });
    }

    render() {
        return (
            <div>
                <input defaultValue={2} onChange={this.onChange} />

                URL: {this.props.url + '/days?=' + this.state.days}
            </div>
        )
    }
}
```

Это связано с тем, что в приложении React мы хотим иметь единственный источник правды - т.е. все данные являются ответственностью одного единственного компонента и только

одного компонента. Ответственность этого компонента заключается в сохранении данных в его состоянии и распространении данных на другие компоненты через реквизиты.

В первом примере как класс MyComponent, так и его родительский элемент поддерживают «url» в своем состоянии. Если мы обновим state.url в MyComponent, эти изменения не будут отражены в родительском. Мы потеряли наш единственный источник правды, и становится все труднее отслеживать поток данных через наше приложение. Сравните это со вторым примером - url поддерживается только в состоянии родительского компонента и используется в качестве опоры в MyComponent - поэтому мы сохраняем единый источник правды.

Состояние, события и управляемые элементы управления

Вот пример компонента React с «управляемым» полем ввода. Всякий раз, когда изменяется значение поля ввода, вызывается обработчик событий, который обновляет состояние компонента с новым значением поля ввода. Вызов `setState` в обработчике событий вызовет вызов для `render` обновления компонента в dom.

```
import React from 'react';
import {render} from 'react-dom';


class ManagedControlDemo extends React.Component {

  constructor(props){
    super(props);
    this.state = {message: ""};
  }

  handleChange(e){
    this.setState({message: e.target.value});
  }

  render() {
    return (
      <div>
        <legend>Type something here</legend>
          <input
            onChange={this.handleChange.bind(this)}
            value={this.state.message}
            autoFocus />
        <h1>{this.state.message}</h1>
      </div>
   );
  }
}


render(<ManagedControlDemo/>, document.querySelector('#app'));
```

Очень важно отметить поведение во время работы. Каждый раз, когда пользователь меняет значение в поле ввода

- `handleChange` будет вызван и так
- `setState` будет вызываться и так
- `render` будет вызван

Pop quiz, после ввода символа в поле ввода, с помощью которого элементы DOM изменяются

1. все из них - верхний уровень div, легенда, вход, h1
2. только вход и h1
3. ничего такого
4. Что такое DOM?

Вы можете поэкспериментировать с этим больше здесь, чтобы найти ответ

Прочитайте Состояние в действии онлайн: https://riptutorial.com/ru/reactjs/topic/1816/состояние-в-действии

глава 29: Спектакль

Examples

Основы - HTML DOM vs Virtual DOM

HTML DOM дорогой

Каждая веб-страница представляется внутренне как дерево объектов. Это представление называется *Document Object Model* . Более того, это интерфейс, нейтральный для языков, который позволяет языкам программирования (например, JavaScript) обращаться к элементам HTML.

Другими словами

> HTML DOM является стандартом для получения, изменения, добавления или удаления элементов HTML.

Однако эти **операции DOM** чрезвычайно **дороги** .

Виртуальный DOM - это решение

Поэтому команда React разработала идею абстрагирования *HTML DOM* и создания собственной *виртуальной DOM* , чтобы вычислить минимальное количество операций, которые необходимо применить на *HTML DOM,* чтобы воспроизвести текущее состояние нашего приложения.

Виртуальный DOM экономит время от ненужных модификаций DOM.

Как точно?

В каждый момент времени React имеет состояние приложения, представленное как `Virtual DOM` . Когда изменяется состояние приложения, это шаги, которые выполняет React для оптимизации производительности

1. Создайте новый *виртуальный DOM,* который представляет новое состояние нашего приложения
2. Сравните старый виртуальный DOM (который представляет текущий HTML DOM) и новый виртуальный DOM
3. На основе 2. найдите минимальное количество операций для преобразования старой

виртуальной DOM (которая представляет текущий HTML DOM) в новую виртуальную DOM

- чтобы узнать больше об этом - прочитать алгоритм Diff Reiff

4. После обнаружения этих операций они отображаются в их эквивалентные операции *HTML DOM*

- помните, что *Virtual DOM* является лишь абстракцией *HTML DOM* и существует изоморфное соотношение между ними

5. Теперь минимальное количество операций, которые были найдены и перенесены в их эквивалентные операции *HTML DOM* , теперь применяются непосредственно к *HTML DOM приложения* , что экономит время от необходимости изменения *HTML DOM* .

Примечание. Операции, применяемые в виртуальной DOM, дешевы, потому что виртуальная DOM является объектом JavaScript.

Алгоритм сравнения React

Создание минимального количества операций для преобразования одного дерева в другое имеет сложность в порядке O (n ^ 3), где n - количество узлов в дереве. Реакция опирается на два предположения для решения этой задачи в линейном времени - O (n)

1. Два компонента одного и того же класса будут генерировать похожие деревья, а два компонента разных классов будут генерировать разные деревья.

2. Можно обеспечить уникальный ключ для элементов, которые стабильны в разных визуализаторах.

Чтобы решить, различаются ли два узла, React различает 3 случая

1. Два узла отличаются друг от друга, если они имеют разные типы.

- например, `<div>...</div>` отличается от `<span>...</span>`

2. Всякий раз, когда два узла имеют разные ключи

- например, `<div key="1">...</div>` отличается от `<div key="2">...</div>`

Более того, **следующее следует** иметь **решающее значение и чрезвычайно важно понять** , хотите ли вы оптимизировать производительность

> Если они [два узла] не одного типа, React не собирается даже пытаться сопоставить то, что они отображают. Он просто удалит первый из DOM и вставляет второй.

Вот почему

> Очень маловероятно, что элемент будет генерировать DOM, который будет выглядеть так, как будет генерироваться. Вместо того, чтобы тратить время на то, чтобы сопоставить эти две структуры, React просто восстанавливает дерево с нуля.

Советы и хитрости

Если два узла не одного типа, React не пытается их сопоставить - он просто удаляет первый узел из DOM и вставляет второй. Вот почему первый совет говорит

1. Если вы видите себя чередующимся между двумя классами компонентов с очень похожим выходом, вы можете сделать его одним и тем же классом.

---

2. Используйте shouldComponentUpdate, чтобы исключить компонент из rerender, если вы знаете, что он не изменится, например

```
shouldComponentUpdate: function(nextProps, nextState) {
  return nextProps.id !== this.props.id;
}
```

Измерение производительности с помощью ReactJS

Вы не можете улучшить то, что вы не можете измерить . Чтобы улучшить производительность компонентов React, вы должны быть способны их измерить. ReactJS обеспечивает *аддон* инструментов для измерения производительности. Импортируйте модуль `react-addons-perf` для измерения производительности

```
import Perf from 'react-addons-perf' // ES6
var Perf = require('react-addons-perf') // ES5 with npm
var Perf = React.addons.Perf; // ES5 with react-with-addons.js
```

Вы можете использовать ниже методы из импортированного модуля `Perf` :

- Perf.printInclusive ()
- Perf.printExclusive ()
- Perf.printWasted ()
- Perf.printOperations ()
- Perf.printDOM ()

Самый важный, который вам понадобится большую часть времени, - `Perf.printWasted()` который дает вам табличное представление о потерянном времени вашего отдельного компонента

(index)	Owner > component	Waste
0	"Todos > TodoItem"	102.7

Total time: 132.71 ms

Вы можете отметить столбец « **Wasted time»** в таблице и улучшить производительность компонента, используя раздел « **Советы и рекомендации** » выше

Обратитесь к официальному руководству React и отличной статье Benchling Engg. по реакционной эффективности

Прочитайте Спектакль онлайн: https://riptutorial.com/ru/reactjs/topic/6875/спектакль

глава 30: Формы и пользовательский ввод

Examples

Контролируемые компоненты

Контролируемые компоненты формы определяются с помощью свойства `value` . Значение управляемых входов управляется React, пользовательские входы не будут иметь прямого влияния на визуализированный вход. Вместо этого изменение свойства `value` должно отражать это изменение.

```
class Form extends React.Component {
  constructor(props) {
    super(props);

    this.onChange = this.onChange.bind(this);

    this.state = {
      name: ''
    };
  }

  onChange(e) {
    this.setState({
      name: e.target.value
    });
  }

  render() {
    return (
      <div>
        <label for='name-input'>Name: </label>
        <input
          id='name-input'
          onChange={this.onChange}
          value={this.state.name} />
      </div>
    )
  }
}
```

В приведенном выше примере показано, как свойство `value` определяет текущее значение ввода, а `onChange` события `onChange` обновляет состояние компонента с помощью ввода пользователем.

Входы формы должны быть определены как контролируемые компоненты, где это возможно. Это гарантирует, что состояние компонента и входное значение постоянно синхронизируются, даже если значение изменяется с помощью триггера, отличного от пользовательского ввода.

Неконтролируемые компоненты

Неконтролируемые компоненты являются входами, которые не имеют свойства `value` . В противоположность контролируемым компонентам, задача приложения заключается в сохранении состояния компонента и входного значения в синхронизации.

```
class Form extends React.Component {
  constructor(props) {
    super(props);

    this.onChange = this.onChange.bind(this);

    this.state = {
      name: 'John'
    };
  }

  onChange(e) {
    this.setState({
      name: e.target.value
    });
  }

  render() {
    return (
      <div>
        <label for='name-input'>Name: </label>
        <input
          id='name-input'
          onChange={this.onChange}
          defaultValue={this.state.name} />
      </div>
    )
  }
}
```

Здесь состояние компонента обновляется через `onChange` события `onChange` , как и для контролируемых компонентов. Однако вместо свойства `value` свойство `defaultValue` . Это определяет начальное значение ввода во время первого рендера. Любые последующие изменения состояния компонента автоматически не отражаются на входном значении; Если это необходимо, вместо этого следует использовать контролируемый компонент.

Прочитайте Формы и пользовательский ввод онлайн:
https://riptutorial.com/ru/reactjs/topic/2884/формы-и-пользовательский-ввод

глава 31: Функциональные компоненты без учета состояния

замечания

Безстоящие функциональные компоненты в Реактике - это чистые функции передаваемых в `props` . Эти компоненты не полагаются на состояние и отказываются от использования методов жизненного цикла компонентов. Тем не менее, вы можете определить `propTypes` и `defaultPropts` .

См. Https://facebook.github.io/react/docs/reusable-components.html#stateless-functions для получения дополнительной информации о функциональных компонентах без состояния.

Examples

Функциональный компонент без учета состояния

Компоненты позволяют разделить пользовательский интерфейс на *самостоятельные* , *многоразовые* элементы. Это красота Реакта; мы можем разделить страницу на многие мелкие повторно используемые **компоненты** .

До того, как React v14 мы могли создать компонент `React.Component` (в ES6), который может быть создан с `React.Component` состояния, или `React.createClass` (в ES5), независимо от того, требуется ли какое-либо состояние для управления данными или нет.

React v14 представил более простой способ определения компонентов, обычно называемых **функциональными компонентами без состояния** . Эти компоненты используют простые функции JavaScript.

Например:

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

Эта функция является действительной компонентой React, потому что она принимает один аргумент объекта `props` с данными и возвращает элемент React. **Функциональные** элементы называются **функциональными,** поскольку они являются буквально *функциями* JavaScript.

Функциональные компоненты без учета состояния обычно фокусируются на пользовательском интерфейсе; состояние должно управляться компонентами

«контейнера» более высокого уровня или через Flux / Redux и т. д. Функциональные компоненты без состояния не поддерживают методы состояния или жизненного цикла.

Выгоды:

1. Нет накладных расходов класса
2. Не нужно беспокоиться об `this` ключевом слове
3. Легко писать и легко понять
4. Не нужно беспокоиться об управлении государственными ценностями
5. Улучшение производительности

Резюме . Если вы пишете компонент React, который не требует состояния и хотел бы создать пользовательский интерфейс многократного использования, вместо создания стандартного компонента React вы можете записать его как **функциональный компонент без состояния** .

Возьмем простой пример:

Предположим, у нас есть страница, которая может регистрировать пользователя, искать зарегистрированных пользователей или отображать список всех зарегистрированных пользователей.

Это точка входа приложения, `index.js` :

```
import React from 'react';
import ReactDOM from 'react-dom';

import HomePage from './homepage'

ReactDOM.render(
    <HomePage/>,
    document.getElementById('app')
);
```

Компонент `HomePage` предоставляет пользовательский интерфейс для регистрации и поиска пользователей. Обратите внимание, что это типичный компонент React, включающий состояние, пользовательский интерфейс и поведенческий код. Данные для списка зарегистрированных пользователей хранятся в переменной `state` , но наш `List` многократного использования (показан ниже) инкапсулирует код пользовательского интерфейса для списка.

`homepage.js` :

```
import React from 'react'
import {Component} from 'react';

import List from './list';

export default class Temp extends Component{
```

```
    constructor(props) {
        super();
        this.state={users:[], showSearchResult: false, searchResult: []};
    }

    registerClick(){
        let users = this.state.users.slice();
        if(users.indexOf(this.refs.mail_id.value) == -1){
            users.push(this.refs.mail_id.value);
            this.refs.mail_id.value = '';
            this.setState({users});
        }else{
            alert('user already registered');
        }
    }

    searchClick(){
        let users = this.state.users;
        let index = users.indexOf(this.refs.search.value);
        if(index >= 0){
            this.setState({searchResult: users[index], showSearchResult: true});
        }else{
            alert('no user found with this mail id');
        }
    }

    hideSearchResult(){
        this.setState({showSearchResult: false});
    }

    render() {
        return (
            <div>
                <input placeholder='email-id' ref='mail_id'/>
                <input type='submit' value='Click here to register'
onClick={this.registerClick.bind(this)}/>
                <input style={{marginLeft: '100px'}} placeholder='search' ref='search'/>
                <input type='submit' value='Click here to register'
onClick={this.searchClick.bind(this)}/>
                {this.state.showSearchResult ?
                    <div>
                        Search Result:
                        <List users={[this.state.searchResult]}/>
                        <p onClick={this.hideSearchResult.bind(this)}>Close this</p>
                    </div>
                    :
                    <div>
                        Registered users:
                        <br/>
                        {this.state.users.length ?
                            <List users={this.state.users}/>
                            :
                            "no user is registered"
                        }
                    </div>
                }
            </div>
        );
    }
}
```

Наконец, наш `List` **функциональных компонентов** без учета состояния, который используется, отображает список зарегистрированных пользователей *и* результаты поиска, но не поддерживает само состояние.

`list.js` :

```
import React from 'react';
var colors = ['#6A1B9A', '#76FF03', '#4527A0'];

var List = (props) => {
    return(
        <div>
            {
                props.users.map((user, i)=>{
                    return(
                        <div key={i} style={{color: colors[i%3]}}>
                            {user}
                        </div>
                    );
                })
            }
        </div>
    );
}

export default List;
```

Ссылка: https://facebook.github.io/react/docs/components-and-props.html

Прочитайте Функциональные компоненты без учета состояния онлайн: https://riptutorial.com/ru/reactjs/topic/6588/функциональные-компоненты-без-учета-состояния

кредиты

S. No	Главы	Contributors
1	Начало работы с реакцией	Adam, Adrián Daraš, Alex, Alex Young, Anuj, Bart Riordan, Cassidy, Community, Daksh Gupta, Dave Kaye, diabolicfreak, DMan, Donald, Everettss, Gianluca Esposito, himanshuIIITian, hyde, Ilya Lyamkin, Inanc Gumus, ivarni, jengeb, jolyonruss, Jon Chan, JordanHendrix, juandemarco, Kaloyan Kosev, Konstantin Grushetsky, Maksim, Marty, MaxPRafferty, Md. Nahiduzzaman Rose, Md.Sifatul Islam, Ming Soon, MMachinegun, Nick Bartlett, orvi, paqash, Prakash, rossipedia, Shabin Hashim, Simplans, Sunny R Gupta, TheShadowbyte, Timo, Tushar Khanna, user2314737
2	JSX	Kaloyan Kosev, Ming Soon
3	React.createClass vs extends React.Component	Kaloyan Kosev, leonardoborges, Michael Peyper, pwolaq, Qianyue, sqzaman
4	Введение в серверную визуализацию	Adrián Daraš, MauroPorrasP
5	Использование ReactJS в потоке Flux	vintproykt
6	Использование ReactJS с jQuery	Kousha, Shuvo Habib
7	Использование ReactJS с машинописным текстом	Everettss, John Ruddell, kevgathuku, Leone, Rajab Shakirov
8	Использование реакции с потоком	JimmyLv, lifeiscontent, Rifat, Rory O'Kane
9	Как и почему использовать ключи в React	Sammy I.

10	Как настроить базовый веб-пакет, реагировать и загружать среду	Bart Riordan, Tien Do, Zac Braddy
11	Ключи реагирования	Dennis Stücken, thibmaek
12	Компоненты	akashrajkn, Anuj, Bart Riordan, Bond, Brandon Roberts, Denis Ivanov, Diego V, DMan, Evan Hammer, Everettss, goldbullet, GordyD, hmnzr, Ilya Lyamkin, ivarni, Jagadish Upadhyay, jbmartinez, John Ruddell, jolyonruss, Jon Chan, jonathangoodman, JordanHendrix, justabuzz, k170, Kousha, Kyle Richardson, m_callens, Maayan Glikser, Michael Peyper, Paul Graffam, philpee2, QoP, Radu Brehar, Sai Vikas, sjmarshy, Timo, Vlad Bezden, WooCaSh, Zakaria Ridouh, zurfyx
13	Компоненты более высокого порядка	Dennis Stücken
14	Монтаж	Rene R, Ruairi O'Brien
15	Настройка среды реагирования	ghostffcode, Tien Do
16	Общение между компонентами	Random User
17	Реагирование, установка Webpack & TypScript	Aron
18	Реагировать жизненный цикл компонентов	Alex Young, Alexg2195, Anuj, Ashari, Everettss, F. Kauder, irrigator, John Ruddell, QoP, Salman Saleem, Saravana, Siddharth, skav, Timo, ultrasamad, Vivian, WitVault
19	Реагировать на вызов AJAX	adamboro, Fabian Schultz, Jason Bourne, lifeiscontent, McGrady , Sunny R Gupta
20	Реагировать на котел [React + Babel + Webpack]	Mihir, parlad neupane, Tien Do
21	Реагировать на формы	promisified

22	Реагировать с Redux	Jim
23	Реагирующие инструменты	brillout
24	Реакция маршрутизации	abhirathore2006, Robeen
25	Реквизит в действии	Ahmad, Anuj, Danillo Corvalan, Everettss, Faktor 10, Fellow Stranger, hansn, Ilya Lyamkin, Jack7, Jagadish Upadhyay, JimmyLv, MaxPRafferty, QoP, Sergii Bishyr, vintproykt, WitVault, zbynour
26	Решения для пользовательского интерфейса	vintproykt
27	Связь между компонентами	David, Kaloyan Kosev
28	Состояние в действии	Alex Young, Alexander, Brad Colthurst, Everettss, Kousha, Kyle Richardson, QoP, skav, Timo
29	Спектакль	Aditya Singh, lustoykov, thibmaek
30	Формы и пользовательский ввод	Everettss, Henrik Karlsson, ivarni, Timo
31	Функциональные компоненты без учета состояния	Adam, Mark Lapierre, Mayank Shukla, Valter Júnior