



Урок 5

Взаимодействие в ReactJS

Организовываем взаимодействия внутри компонентов

[Как организуется взаимодействие компонентов](#)

[Отрисовка и события в ReactJS](#)

[Передача параметров в компоненты с помощью свойства props.](#)

[Связываем обработчики событий с методами](#)

[Как отрисовать несколько компонентов в рамках одного компонента \(дочерние компоненты\)](#)

[Компоненты-представления](#)

[Компоненты-контейнеры](#)

[Практика](#)

[Задача 1. Использование props](#)

[Задача 2. Пример синтаксиса компонентов-представлений](#)

[Задача 3. Пример синтаксиса компонентов-контейнеров](#)

[Домашнее задание](#)

[Дополнительные материалы](#)

[Используемая литература](#)

Как организуется взаимодействие компонентов

Каждый компонент отрисовывает себя единожды, основываясь на своих параметрах. Props неизменны — это значит, что они передаются от родителя, и он остается их владельцем. Для организации взаимодействия мы добавим изменяемое свойство в компонент. `this.state` является приватным для компонента и может быть изменён через вызов `this.setState()`. После обновления свойства компонент заново отрисует себя. `Render()` методы написаны декларативно, как функции `this.props` и `this.state`. React гарантирует соответствие данных на сервере и в интерфейсе пользователя.

Для взаимодействия компонентов будем использовать React Router - полная библиотека для маршрутизации. Он хранит пользовательский интерфейс, синхронизация с URL. Имеет простой API-интерфейс с мощными функциями, такие как: отложенная загрузка кода, динамическое согласование маршрута, и перехода в месте обработки встроенными.

Для установки прописываем в командной строке:

```
$ npm install --save react-router
```

Затем используем bundler-модуль, webpack, прописываем следующий шаблон:

```
// используем babel
import { Router, Route, Link } from 'react-router'
// без использования ES6
var Router = require('react-router').Router
var Route = require('react-router').Route
var Link = require('react-router').Link
```

Отрисовка и события в ReactJS

Данные могут передаваться от контейнера в представление, но возникают вопросы по поводу поведения данных. События попадают в категорию поведения и часто должны изменять данные. В React события навешиваются непосредственно в представлении. Это может создать проблему для разделения представления и данных.

React Router - это мощная библиотека маршрутизации построен на вершине реагировать, что позволяет добавлять новые экраны и потоки вашего приложения невероятно быстро, сохраняя URL-адрес в синхронизации с тем, что отображается на странице.

Чтобы понять проблему, добавим событие в компонент-представление (клик по элементу `<button>`-данный код будет рассматриваться в ES5):

```
// Компонент-представление
var UserList = React.createClass({
  render: function() {
    return (
      <ul className="user-list">
        {this.props.users.map(function(user) {
          return (
            <li key={user.id}>
              <Link to={"/users/" + user.id}>{user.name}</Link>
              <button onClick={this.toggleActive}>Toggle Active</button>
            </li>
          );
        })}
      </ul>
    );
  },
  toggleActive: function() {
    // Необходимо изменить состояние в компоненте-представлении
  }
});
```

Технически это будет работать, но будут возникать ошибки, наслаивающиеся друг на друга. Скорее всего, событие должно будет изменять данные, которые хранятся в состоянии, о котором компонент-представление не должен знать. Лучшим решением будет передать функцию из контейнера в представление в качестве свойства:

```
// Container Component
var UserListContainer = React.createClass({
  ...
  render: function() {
    return (<UserList users={this.state.users} toggleActive={this.toggleActive} />);
  },
  toggleActive: function() {
    // We should change state in container components :)
  }
});

// Presentational Component
var UserList = React.createClass({
  render: function() {
    return (
      <ul className="user-list">
        {this.props.users.map(function(user) {
          return (
            <li key={user.id}>
              <Link to={"/users/" + user.id}>{user.name}</Link>
              <button onClick={this.props.toggleActive}>Toggle Active</button>
            </li>
          );
        })}
      </ul>
    );
  }
});
```

Атрибут `onClick` задан в представлении, а функция вынесена в контейнер. Это более подходящее решение, поскольку контейнер отвечает за состояние.

В родительском компоненте происходит изменение состояния, при этом вызывается метод `render`, который обновляет свойства дочернего компонента, что, в свою очередь, приводит к перерисовке дочернего компонента. Это происходит в React автоматически.

Рассмотрим пример, в котором будут рассматриваться возможности менять состояние компонента-контейнера, что автоматически вызывает обновление компонента-представления. Обратите внимание, как этот пример работает с неизменяемыми данными и использует метод `bind()`.

Компонент-кнопка, самый простой пример, который только можно придумать. Тем не менее, на столь малом количестве кода можно проиллюстрировать многие особенности работы с React. Начнём с состояния:

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

Задавая свойство `state` для текущего класса, вы говорите реакту: “Это данные, за которыми стоит следить”. При изменении состояния React будет проводить свои магические манипуляции с виртуальным DOM и заново рендерить все изменившиеся элементы. При использовании свойства `state` необходимо придерживаться одного простого правила: состояние задаётся присваиванием всего один раз при инициализации компонента. Другими словами, не стоит присваивать значения напрямую, а вместо этого использовать функцию `setState`:

```
// Плохо
this.state.count=0;
// Хорошо
this.setState({
  count:0
});
```

Для изменения состояния чаще всего удобно создать отдельный метод, для кнопки подобный метод — `increment`, который и увеличивает счетчик на единицу:

```
increment(){
  this.setState({
    count:this.state.count+1
  });
}
```

С помощью свойства `this.state.count` получаем текущее значение, прибавляем к нему единицу и заново устанавливаем состояние.

Передача параметров в компоненты с помощью свойства props.

Неизменяемое состояние компонента передаётся ему в качестве аргумента при создании. Если бы было необходимо изменить HelloWorld так, чтобы он принимал в качестве аргумента начальное сообщение, мы бы сделали так:

```
var HelloWorld = React.createClass({
  ...
  getInitialState(): function() {
    return {message: this.props.welcomeMessage};
  }
  ...
});
React.renderComponent(
  HelloWorld({welcomeMessage: 'Hello World!'}),
  document.getElementById('app')
);
```

При создании компонента HelloWorld, в качестве первого аргумента передаём объект с props. В данном случае {welcomeMessage:'HelloWorld!'}. Эти свойства доступны в контексте компонента через this.props.

Пример использования props.

Создадим новый файл Header.js и скопируем в него App.js? изменим.

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

class Header extends Component {
  render() {
    return (
      <div>
        Header
      </div>
    );
  }
}
export default Header;
```

Импортируемого в App.js. За одно удалим Dropdown. И вставим Header вместо Dropdown.

```
import React, { Component } from 'react';
import Header from './Header';
class App extends Component {
  render() {
    return (
      <div>
        <Header />
      </div>
    );
  }
}
export default App;
```

В браузере вывелось слово header. Всё нормально работает и компонент отрендерился.

Хорошей практикой считается делать один “умный” компонент на страницу и вкладывать в него “глупые” компоненты. При условии, что уровень вложенности не очень глубокий. Умный компонент может получать данные, например, от бэкенда, а глупые просто получают данные от родителя. Чтобы разобраться, создадим меню, которое будет массив объектов.

```
const menu = [
  {
    link: '/articles',
    label: 'Articles'
  },
  {
    link: '/contacts',
    label: 'Contacts'
  },
  {
    link: '/posts',
    label: 'Posts'
  }
];
```

Пробросим их как параметр в Header. Для этого мы просто указываем атрибут items в значении menu.

```
<Header items={menu} />
```

Для того, чтобы работать с ними в Header нужно использовать this.props. Давайте напишем console.log в рендер методе:

```
console.log('items', this.props.items);
```

В консоли вывелся наш массив. И теперь мы можем их рендерить:

```
{this.props.items.map((item, index) =>
  <a href={item.link}>{item.label}</a>
)}
```

Если мы посмотрим в браузер, то увидим, что наш массив отрендерился, но у нас есть Warning.

```
warning.js:36 Warning: Each child in an array or iterator should have a unique "key" prop.  
Check the render method of `Header`.
```

Каждый элемент внутри массива должен иметь уникальный ключ. Поэтому давайте добавим атрибут key.

```
{this.props.items.map((item, index) =>  
<a href={item.link} key={index}>{item.label}</a>  
)}
```

Никаких ошибок нет.

Связываем обработчики событий с методами

Обработка событий с элементами React очень схожа с обработкой событий с элементами DOM. Существует несколько синтаксических различий: Названия событий React создаются с помощью camelCase, а не lowercase. С JSX вы передаете функцию как обработчик события, а не строку. Например, HTML:

```
<button onclick="activateLasers()">  
  Activate Lasers  
</button>
```

В React это выглядит по-другому:

```
<button onClick={activateLasers}>  
  Activate Lasers  
</button>
```

Ещё одно отличие заключается в том, что в React вы не можете вернуть false к предыдущему состоянию по умолчанию. Явно необходим алгоритм preventDefault. Например, с помощью обычного HTML, чтобы предотвратить по умолчанию открытие по ссылке новой страницы, вы можете написать:

```
<a href="#" onclick="console.log('The link was clicked.');" return false">  
  Clickme  
</a>
```

В React это будет выглядеть так:

```
functionActionLink() {
functionhandleClick(e) {
e.preventDefault();
console.log("The link was clicked.");
}
return (
<a href="#" onClick={handleClick}>
Clickme
</a>
); }
```

Здесь “e”- это синтетическое событие. React определяет такие синтетические события согласно [W3C spec](#), поэтому вам не стоит беспокоиться о кросс-браузерной совместимости.

Как правило, при использовании React, вам не нужно отсылать сигнал к `addEventListener`, чтобы добавить обработчики событий к DOM-элементу после его создания. Вместо этого, просто выполните обработчик, когда элемент впервые отображается.

Когда определяется компонент, используя класс ES6, общим образом для обработчика событий и будет алгоритмом класса. Например, компонент `Toggle` отображает кнопку, которая позволяет пользователям выбирать между состояниями «ON» и «OFF»:

```
class Toggle extends React.Component {
constructor(props) {
super(props);
this.state = {isToggleOn: true};
// This binding is necessary to make `this` work in the callback
this.handleClick = this.handleClick.bind(this);
}
handleClick() {
this.setState(prevState => ({
isToggleOn: !prevState.isToggleOn
}));
}
render() {
return (
<button onClick={this.handleClick}>
{this.state.isToggleOn ? 'ON' : 'OFF'}
</button>
);
}
}
ReactDOM.render(
<Toggle />,
document.getElementById('root')
);
```

Будьте осторожны с этими значениями в обратных сигналах JSX. Классовые алгоритмы не связаны по умолчанию. Если вы забыли связать их, выполните `this.handleClick` и передайте его на `onClick`. `This` станет `undefined`, когда функция получит сигнал.

Такое действие нетипично для React, оно относится к работе функций в JavaScript. В основном, если вы обращаетесь к алгоритму без `()` после него, например, как `onClick={this.handleClick}`, вы должны связать данный алгоритм.

Если вы не хотите отправлять сигналы в `bind`, то есть ещё два возможных действия. Если вы

используете экспериментальный плагин `property initializer syntax`, то вы можете использовать инициализаторы реализуемых свойств, чтобы правильно связать обратные вызовы:

```
class LoggingButton extends React.Component {
  // This syntax ensures `this` is bound within handleClick.
  // Warning: this is *experimental* syntax.
  handleClick = () => {
    console.log('this is:', this);
  }
  render() {
    return (
      <button onClick={this.handleClick}>
        Click me
      </button>
    );
  }
}
```

Данный синтаксис в [CreateReactApp](#) включен по умолчанию. Если вы не используете инициализаторы свойств, вы можете применить стрелочную функцию (arrow function) в обратных сигналах:

```
class LoggingButton extends React.Component {
  handleClick() {
    console.log('this is:', this);
  }
  render() {
    // This syntax ensures `this` is bound within handleClick
    return (
      <button onClick={(e) => this.handleClick(e)}>
        Click me
      </button>
    );
  }
}
```

Проблема с таким синтаксисом состоит в том, что другой обратный сигнал создаётся каждый раз, когда выводится `LoggingButton`. Чаще всего, это нормальное явление. Однако, если обратные сигналы передаются как свойства для нижних компонентов, то такие компоненты могут произвести повторное демонтирование. Чаще всего мы советуем производить связывание в конструкторе, чтобы избежать подобных проблем с производительностью. Как отрисовать несколько компонентов в рамках одного компонента (дочерние компоненты).

В разделе `Получение данных с Ajax` мы создали проблему. Компонент `UserList` работает, но он пытается делать слишком много вещей. Чтобы решить эту проблему, разделим `UserList` на два компонента, каждый из которых будет выполнять одну свою функцию. Эти два типа компонентов будем называть `компоненты-контейнеры` и `компоненты-представления`, или «умные» и «глупые» компоненты. Если кратко, `компоненты-контейнеры` отвечают за данные и операции с ними. Их состояние передается в виде свойств в `компоненты-представления` и отображается.

Как отрисовать несколько компонентов в рамках одного компонента (дочерние компоненты)

Компоненты-представления

Для описания компонентов-представлений рассмотрим, как выглядел компонент `UserList` до того, как получил собственное состояние:

```
var UserList=React.createClass({
  render:function(){
    return(
      <ul className="user-list">
        {this.props.users.map(function(user){
          return(
            <li key={user.id}>
              <Link to={"/users/" + user.id}>{user.name}</Link>
            </li>
          );
        })}
      </ul>
    );
  }
});
```

Это как раз и есть компонент-представление. Этот компонент выводит список элементов цикла, а данные получает в качестве свойства.

Компоненты-представления являются «глупыми» в том смысле, что они не представляют, откуда берутся данные. Они ничего не знают о состоянии.

Компоненты-представления не должны изменять данные. Фактически, любой компонент, получающий свойства от родителя, должен держать их неизменяемыми. В то же время, они могут как-либо форматировать данные (например, конвертируя `Unixtimestamp` в читаемую дату-время).

В React обработчики событий связываются непосредственно в представлении с помощью атрибутов, таких как `onClick`. Возникает вопрос, как работают обработчики событий в компонентах-представлениях, которые не могут изменять свойства. Об этом у нас есть целый раздел ниже.

Компоненты-контейнеры

Компоненты-контейнеры чаще всего являются родителями для компонентов-представлений и обеспечивают связь между представлениями и остальными частями приложения. Их также называют «умными» компонентами, поскольку они знают о приложении в целом.

Компонент-контейнер и компонент-представление должны иметь разные имена, чтобы избежать путаницы, назовем контейнер `UserListContainer`:

```

var React = require('react');
var axios = require('axios');
var UserList = require('../views/list-user');
var UserListContainer = React.createClass({
  getInitialState: function() {
    return {
      users: []
    }
  },
  componentDidMount: function() {
    var _this = this;
    axios.get('/path/to/user-api').then(function(response) {
      _this.setState({users: response.data})
    });
  },
  render: function() {
    return (<UserList users={this.state.users}/>);
  }
});
module.exports = UserListContainer;

```

Компонент-контейнер создаётся как любой другой компонент React. Он имеет метод `render`, но не создаёт элементов DOM, а только возвращает компонент-представление.

Практика

Задача 1. Использование props

Создадим новый файл `Header.js` и скопируем в него `App.js`, изменим.

```

import React, { Component } from 'react';

class Header extends Component {
  render() {
    return (
      <div>
        Header
      </div>
    );
  }
}
export default Header;

```

Импортируемого в `App.js`. За одно удалим `Dropdown`. И вставим `Header` вместо `Dropdown`.

```
import React, { Component } from 'react';
import Header from './Header';
class App extends Component {
  render() {
    return (
      <div>
        <Header />
      </div>
    );
  }
}
export default App;
```

В браузере вывелось слово header. Всё нормально работает и компонент отрендерился. Создадим меню, которое будет массив объектов.

```
const menu = [
  {
    link: '/articles',
    label: 'Articles'
  },
  {
    link: '/contacts',
    label: 'Contacts'
  },
  {
    link: '/posts',
    label: 'Posts'
  }
];
```

Пробросим их как параметр в Header. Для этого мы просто указываем атрибут items в значении menu.

```
<Header items={menu} />
```

Для того, чтобы работать с ними, в Header нужно использовать this.props. Давайте напишем console.log в рендер методе

```
console.log('items', this.props.items);
```

В консоли вывелся наш массив. И теперь мы можем их рендерить:

```
{this.props.items.map((item, index) =>
  <a href={item.link}>{item.label}</a>
)}
```

Если мы посмотрим в браузер, то увидим, что наш массив отрендерился, но у нас есть Warning:

```
warning.js:36 Warning: Each child in an array or iterator should have a unique "key" prop.
Check the render method of `Header`.
```

Каждый элемент внутри массива должен иметь уникальный ключ. Поэтому давайте добавим атрибут key.

```
{this.props.items.map((item, index) =>
<a href={item.link} key={index}>{item.label}</a>
)}
```

Никаких ошибок нет.

Задача 2. Пример синтаксиса компонентов-представлений

```
var UserList = React.createClass({
  render: function() {
    return (
      <ul className="user-list">
        {this.props.users.map(function(user) {
          return (
            <li key={user.id}>
              <Link to={"/users/" + user.id}>{user.name}</Link>
            </li>
          );
        })}
      </ul>
    );
  }
});
```

Задача 3. Пример синтаксиса компонентов-контейнеров

```
var React = require('react');
var axios = require('axios');
var UserList = require('../views/list-user');
var UserListContainer = React.createClass({
  getInitialState: function() {
    return {
      users: []
    }
  },
  componentDidMount: function() {
    var _this = this;
    axios.get('/path/to/user-api').then(function(response) {
      _this.setState({users: response.data});
    });
  },
  render: function() {
    return <UserList users={this.state.users}/>;
  }
});
module.exports = UserListContainer;
```

Домашнее задание

Для приложения необходимо выполнить следующее:

1. Необходимо реализовать страницы: Главная, Блог, Комментарии, Пользователи.
2. Для упрощения процесса разработки можете сделать также, как мы делали на занятии, реализовав Layout компонент (у нас он называется Default), где будет описан шаблон страниц.
3. *На всех страницах должно присутствовать навигационное меню, которое будет выделять текущее местоположение какойнибудь подсветкой элемента меню.

Для Главной страницы:

Можно написать либо текст-пустышку (lorem ipsum), либо общие сведения о проекте.

Для Блога:

1. Необходимо выводить каждый блог отдельным элементом (стилизационные вопросы оставляю на Вас)
2. *При клике на этом блоге должен происходить переход на страницу с деталями под блогу, где будут отображаться записи блога, а так же пользователь - владелец блога

Для страницы Комментарии:

1. Необходимо выводить список последних комментариев добавленных в наш блог.
2. Для каждого выведенного сообщения должна быть доступна ссылка на сам блог.

Для страницы Пользователи:

1. Необходимо выводить пользователей приложения.
2. *При клике на пользовате должна отображаться информация о блогах пользователя и его последних комментариях (это все может отсутствовать).

Все данные можно либо придумать и, например, разместить в json файлах, либо скачивать откуда-то из интернета. Для более грамотной разработки следует все эти вещи помещать в отдельную сущность, например, UserService, BlogService и т.д., которые будут заниматься своими данными, их отдачей, поиском и т.д.

Задачи со * повышенной сложности.

Дополнительные материалы

1. <https://github.com/ReactTraining/react-router>
2. <https://github.com/ReactTraining/react-router/blob/master/docs/Introduction.md>
3. <https://github.com/ReactTraining/react-router/blob/master/docs/API.md>
4. <http://knowbody.github.io/react-router-docs/>
5. <https://github.com/reactjs/react-router-tutorial/tree/master/lessons>
6. <https://github.com/ReactTraining/react-router/tree/master/docs/guides>

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. <https://github.com/ReactTraining/react-router>
2. <https://github.com/ReactTraining/react-router/blob/master/docs/Introduction.md>
3. <https://github.com/ReactTraining/react-router/blob/master/docs/API.md>
4. <http://knowbody.github.io/react-router-docs/>
5. <https://github.com/reactjs/react-router-tutorial/tree/master/lessons>
6. <https://github.com/ReactTraining/react-router/tree/master/docs/guides>