GeekBrains

ReactJS. Базовый курс

Знакомство с ReactJS. Первые компоненты

[React 17.0.1]

На этом уроке

1. Познакомимся с современными подходами к созданию веб-приложений, кратко рассмотрим основные фреймворки и библиотеки, используемые в разработке.
2. Рассмотрим основные принципы, на которых работает React, и узнаем о JSX.
3. Познакомимся с инструментом create-react-app и создадим с его помощью первое приложение на React.
4. Познакомимся с инструментами отладки для приложений на React.
5. Узнаем о работе сборщиков и транспайлеров (webpack & babel).

Оглавление

Теория урока

Современный фронтенд, фреймворки и библиотеки, использующиеся для создания веб-приложений, React и создание первого приложения с помощью create-react-app.

Современные веб-приложения

Мир фронтенда - один из самых быстро развивающихся сегментов разработки. Стандарты и подходы, а также инструменты разработки могут меняться и обновляться буквально каждый год. Сайты сегодня могут представлять собой не просто статический набор информации, а достаточно сложные приложения. Достаточно большое количество логики в них может находиться на фронтенде. Для облегчения и ускорения создания таких приложений, как правило, используются фреймворки и библиотеки, и основными из них являются:

1. AngularJS - “Java в мире фронтенда” - относительно сложный, тяжелый и многословный, но весьма мощный фреймворк, предоставляющий многие инструменты “из коробки”.
2. ReactJS - самый популярный на сегодня инструмент. Относительно прост в изучении, достаточно быстр, однако для многих задач требует установки дополнительных модулей (нехватки в которых, впрочем, нет, т.к. имеет одно из самых активных и обширных сообществ).
3. VueJS - относительно молодой фреймворк, создававшийся как “объединяющий лучшие качества React & Angular”.

Следует отметить, что официальная документация определяет React не как **фреймворк**, а как **библиотеку** для построения пользовательских интерфейсов (с этим связано, в частности, то, что для многих задач требуется установить дополнительные модули). Для многих разработчиков этот момент, впрочем, является спорным (подробнее об этом можно почитать здесь и здесь).

ReactJS. Основные принципы и возможности. Знакомство с JSX.

Основные принципы React

Одной из ключевых концепций приложений на React является декларативность. Сама библиотека построена таким образом, что разработчику удобно использовать декларативный подход - то есть мы указываем, **какое состояние** должен иметь интерфейс, а React сам позаботится о том, как этого состояния достичь. Этот подход противопоставляется императивному программированию, в котором разработчику требуется указать, **какие действия** необходимо совершить, чтобы достичь нужного состояния (подробнее о декларативности и императивности можно почитать здесь).

Кроме того, важной особенностью React является возможность переиспользования кода, что достигается за счет использования компонентного подхода. Разработчик может создать компонент

React (подробнее об этом в следующем разделе), использовать его в нескольких местах, объединять с другими, кастомизировать его и т.д.

Также важно отметить, что, хотя сам React и обладает меньшим количеством встроенных возможностей по сравнению с другими фреймворками, это позволяет разработчику практически полностью кастомизировать стек, на котором разрабатывается приложение, с помощью установки любого из множества сторонних модулей, подходящих для конкретной задачи.

Первые компоненты. Знакомство с JSX

Как было указано выше, приложение на React строится на компонентах. Компонент - по сути, просто функция или класс javascript (класс должен наследовать от React.Component и реализовывать метод render). Соответственно, компоненты делятся на функциональные и классовые.

Взгляните на следующий пример

```
import React from 'react';

function App() {
  return <h1>Hello World</h1>;
}

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

Здесь создается функциональный компонент, который будет добавлять в DOM (“рендерить”) div с надписью “Hello world!”. Добавление корневого компонента в DOM осуществляется с помощью метода ReactDOM.render - этот метод принимает компонент, который необходимо добавить, и элемент DOM, в который следует добавить компонент. Как правило, в приложении вызов этого метода происходит лишь однажды - для корневого компонента, а он, в свою очередь, рендерит остальные компоненты. Важно понимать, что, за исключением единственного вызова этого метода, манипулирования DOM напрямую при использовании React происходить не должно.

Обратите внимание, как происходит создание элементов в компоненте HelloMessage - синтаксис напоминает HTML, хотя написан в .js-файле. С помощью такой же записи компонент HelloMessage передается в ReactDOM.render. Такой синтаксис называется JSX - он был создан разработчиками React для упрощения написания декларативного, лаконичного кода. JSX позволяет использовать не только элементы DOM (как div в примере), но и другие компоненты (HelloMessage в примере). Элементам DOM в JSX можно ставить атрибуты аналогично тому, как это делается в HTML (за исключением атрибута class - так как это слово является в js ключевым, вместо него указывается атрибут className).

Компоненты всегда должны называться с большой буквы, например, HelloMessage - корректное название, а helloMessage - некорректное. Названия с маленькой буквы допустимы только для элементов DOM.

Внимание!

Строго говоря, элемент , описанный в JSX, не является тем же самым, что элемент , описанный в HTML. div в JSX - это специальный объект, который с помощью ReactDOM.render рендерится в элемент DOM (подробнее об этом в следующих уроках).

Взглянем на следующий пример:

```
import React from 'react';

function App(props) {
  return <h1>Hello, {props.name}</h1>;
}

ReactDOM.render(<App name=”Alexander” />, document.getElementById('root'));
```

Атрибуты можно передавать не только элементам DOM, но и компонентам. В этом случае они будут доступны внутри функции-компонента в специальном аргументе props - объекте, содержащем значения этих атрибутов (а в классовом компоненте в свойстве this.props). Такие атрибуты называются пропсами (сокращение от properties). С помощью пропсов мы можем передавать данные от родительских компонентов дочерним.

Обратите внимание, внутри JSX обращение к пропсам (а также любым другим переменным) осуществляется с помощью фигурных скобок: `<span>{variable}</span>`.

```
import React from 'react';

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

ReactDOM.render(<App name=”Alexander” />, document.getElementById('root'));
```

В данном примере аналогичный функционал реализован на классовых компонентах.

Внимание!

До недавнего времени функциональные компоненты обладали ограниченной функциональностью по сравнению с классовыми, поэтому на сегодняшний день можно встретить большое количество

кода, написанного на классах. Сейчас большая часть приложений разрабатывается на функциональных компонентах, и в данном курсе мы будем в основном рассматривать именно их.

Чистые функции

Одним из важных принципов построения приложения на React является разделение компонентов на презентационные компоненты (также называемые глупыми, dummy) и контейнеры. Контейнеры - компоненты, которые содержат некоторую логику, могут, например, отправлять запросы или обрабатывать переданные им данные. Презентационные компоненты служат только для отображения переданных им данных, как правило, являются функциональными компонентами и чистыми функциями.

Чистая функция - функция, не имеющая побочных эффектов (т.е., не изменяющая переменные во внешней области видимости, не отправляющая запросы и т.п.), а также всегда зависящая только от переданных аргументов. Иными словами, для одних и тех же переданных аргументов такая функция всегда будет возвращать один и тот же результат.

Взгляните на следующий пример:

```
function isBefore(timestamp) {
 return timestamp < Date.now();
}
```

Данная функция не является чистой, так как результат ее вызова зависит от текущего момента времени. Однако, можно преобразовать ее следующим образом:

```
function isBefore(timestamp, date) {
 return timestamp < new Date(date).getTime();
}
```

Теперь эта функция зависит только от переданных ей аргументов.

Код, написанный с помощью чистых функций, легче тестировать и оптимизировать. При создании приложений на React, как правило, стараются разделить презентационную и логическую части и сделать презентационные компоненты - чистыми функциями.

В данном уроке рассмотрены только презентационные компоненты.

Внимание!

В отличие от классовых и функциональных компонентов, презентационные компоненты и контейнеры - скорее архитектурный паттерн, чем особенность фреймворка. Подобное

разделение встречается и в других фреймворках, и существует в первую очередь для облегчения поддержки, оптимизации и тестирования кода.

Create-react-app. Первое приложение

Самым простым способом начать разработку на React является использование create-react-app. Этот инструмент был создан разработчиками React специально для облегчения начала работы. Для того чтобы запустить его, необходимо:

1. Установить (или убедиться, что установлены) node и npm
2. Запустить в терминале команду npx create-react-app myapp, где myapp - название вашего приложения

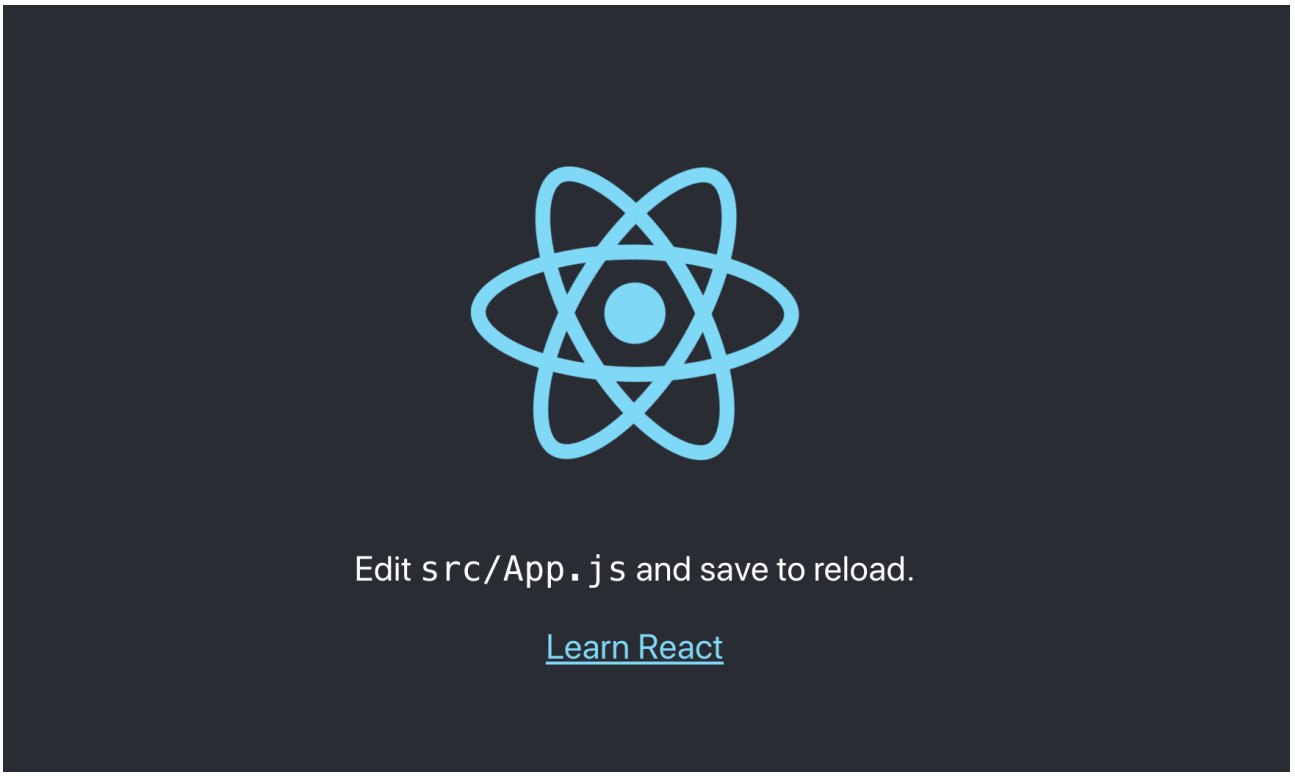
После этого скрипт создаст папку с названием, которое вы указали, а также выполнит установку и настройку необходимых для работы React модулей. Кроме того, будет автоматически создан первый компонент в файле App.js. Для того чтобы увидеть результат работы скрипта, перейдите в папку myapp:

```
cd ./myapp
```

И запустите в терминале следующую команду:

```
npm start
```

В браузере откроется следующая вкладка:



Давайте разберем, что здесь происходит. В папке public находится файл index.html. Он является точкой входа в наше приложение. В его body находится только один div с id “root”. В файле index.js с помощью ReactDOM.render в этот div рендерится компонент App, находящийся в файле App.js. Внутри этого компонент рендерится несколько элементов DOM, отображение которых мы и можем увидеть на странице в браузере.

Внимание!

В index.js также используется компонент React.StrictMode - аналогично директиве use strict для js, он устанавливает более строгие правила для запрета использования устаревших (deprecated) методов и подходов. Также там подключается скрипт reportWebVitals, предназначенный для сбора статистики.

Удалим существующий код компонента и создадим свои с использованием примеров из предыдущего раздела:

App.js

```
import './App.css';

function App() {
 return (
   <div className="App">
     <header className="App-header">
       My First React App
       <h3>Hello world!</h3>
     </header>
```

```
    </div>
  );
}

export default App;
```

Добавим передачу данных пропсами от родительского компонента компоненту App:

index.js

```
const myName = 'Alexander';

ReactDOM.render(
  <React.StrictMode>
    <App name={myName} />
  </React.StrictMode>,
  document.getElementById("root")
);
```

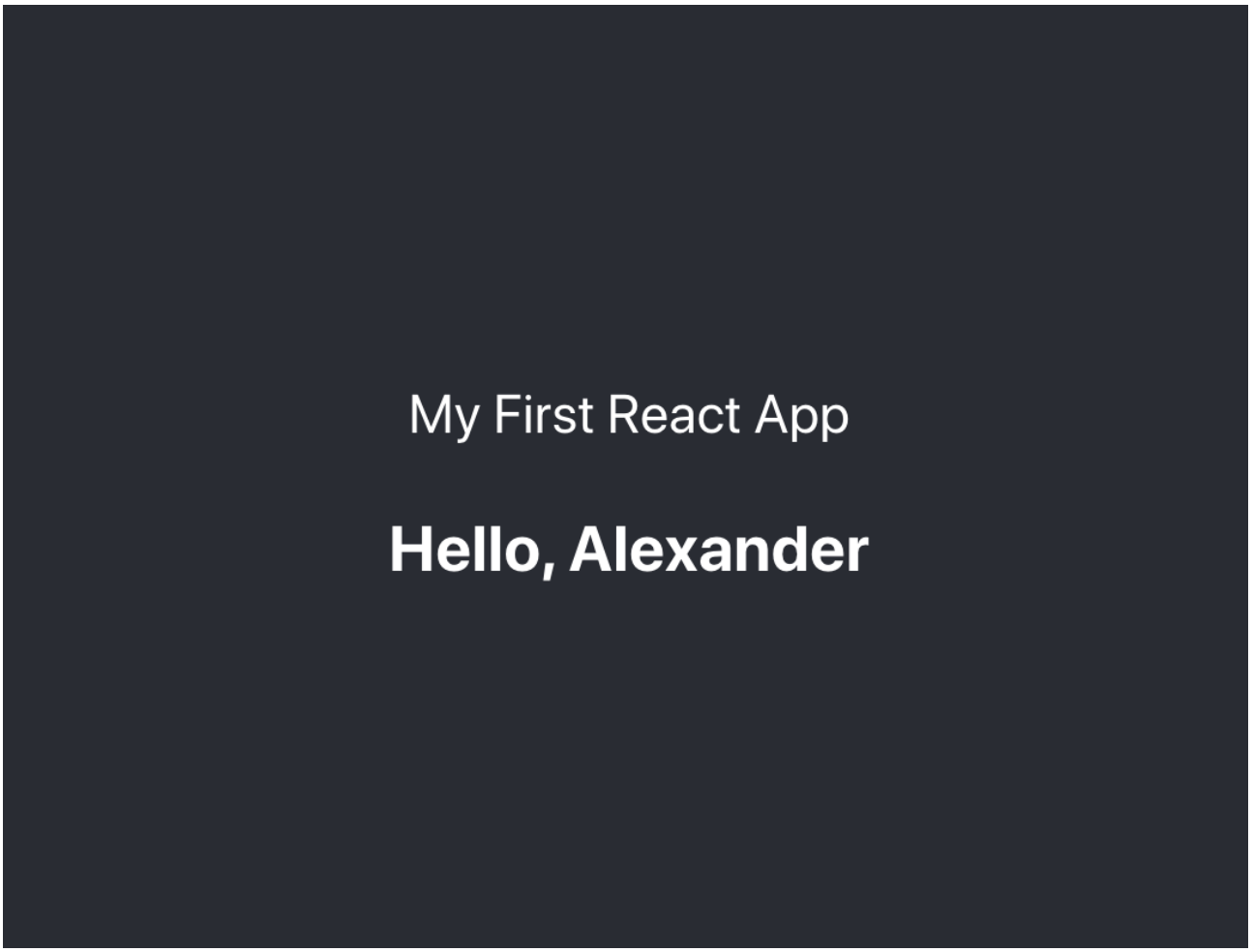
App.js

```
import './App.css';

function App(props) {
  return (
    <div className="App">
      <header className="App-header">
        My First React App
        <h3>Hello, {props.name}</h3>
      </header>
    </div>
  );
}

export default App;
```

Теперь в браузере (после сохранения файла) будет отображаться новый компонент.



Стилизация элементов

Элементам DOM можно устанавливать стили прямо в JSX:

```
function App(props) {
 return (
   <div style={{paddingTop: '25px'}}>
    Inline Styles
   </div>
 );
}

export default App;
```

Такой подход называется inline styles - обратите внимание, что здесь в свойство style передается объект. Его ключи похожи на значения свойств в css, но в них используется camelCase вместо kebab-case. Если значение свойства не является числом, его следует передать как строку.

```
/* В css */
{
  padding-top: 25px;
```

```
  opacity: 0.8;
  position: absolute;
  top: 40%;
}
```

```
// В JSX
{
  paddingTop: '25px',
  opacity: 0.8,
  position: 'absolute',
  top: '40%'
}
```

Можно также использовать css (а также less или sass) - для этого нужно создать и импортировать в файл компонента файл со стилями и установить элементу свойство className:

App.css

```
.App-header {
 background-color: #282c34;
 min-height: 100vh;
 display: flex;
 flex-direction: column;
 align-items: center;
 justify-content: center;
 font-size: calc(10px + 2vmin);
 color: white;
}
```

App.js

```
import './App.css'; // импортируем файл со стилями

function App(props) {
// и устанавливаем класс App-header элементу header
 return (
   <div>
     <header className="App-header">
       My First React App
       <h3>Hello, {props.name}</h3>
     </header>
   </div>
 );
}

export default App;
```

В большинстве случаев следует предпочитать className инлайновым стилям. React и браузер обрабатывают и отрисовывают такие элементы быстрее. Inline styles стоит использовать, если элементу устанавливается некоторое значение свойства стиля (к примеру, top), которое вычисляется непосредственно в компоненте. При этом, конечно, можно сочетать инлайновые стили и классы.

```
import './App.css';

function App(props) {
 const topPosition = '40px';
 return (
   <div>
     <header className="App-header" style={{top: topPosition}}>
       My First React App
       <h3>Hello, {props.name}</h3>
     </header>
   </div>
 );
}

export default App;
```

Как inline styles, так и classNames можно делать вычисляемыми свойствами:

```
import './App.css';

function App(props) {
 return (
   <div>
     <header
      className={`App-header ${props.showRed ? 'header-red' : 'header-blue'}`}
      style={{top: props.topPosition || '10px'}}
     >
       My First React App
       <h3>Hello, {props.name}</h3>
     </header>
   </div>
 );
}

export default App;
```

В данном примере элементу header будут установлены следующие стили:

1. Стили css-класса App-header
2. Если проп showRed передан, и его значение truthy, то будут установлены стили css-класса header-red, иначе - стили css-класса header-blue

3. Устанавливается значение свойства стиля top: если передан проп topPosition, то будет установлен он, в противном случае - значение 10px

Внимание!

При использовании CRA форматы css и less поддерживаются по умолчанию. Для использования sass необходимо установить node-sass:

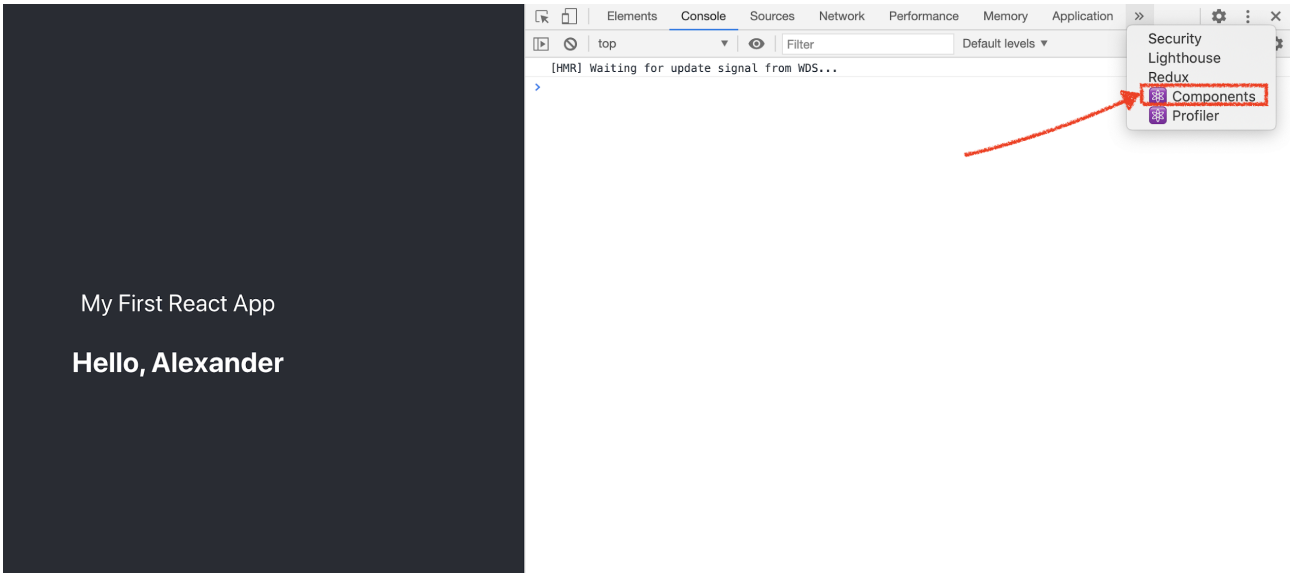
```
npm install node-sass --save
```

После этого можно использовать scss-формат.

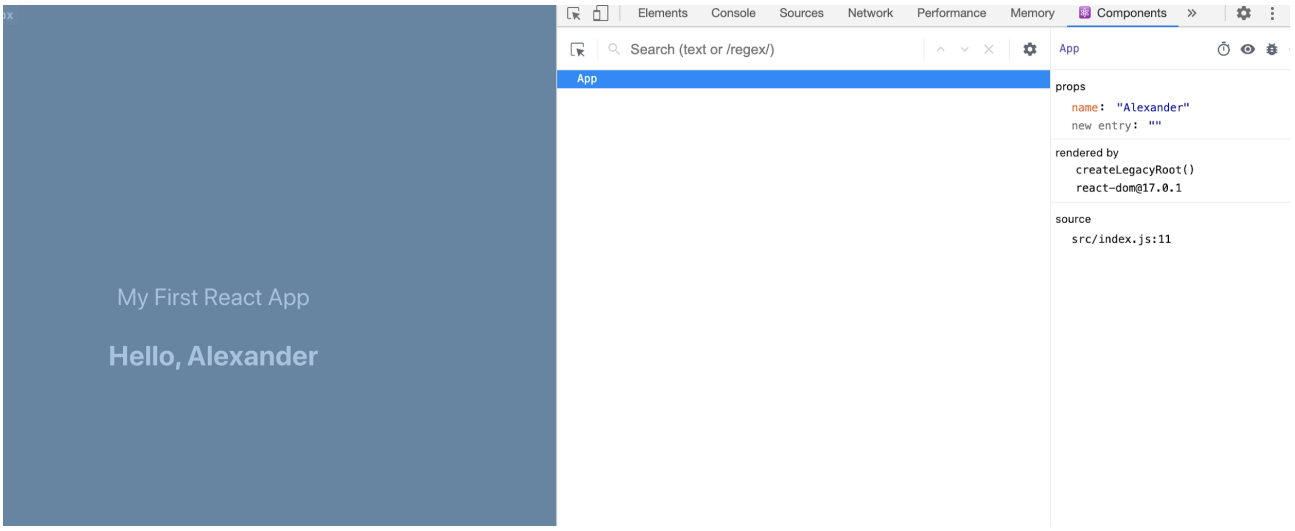
React Devtools

Для облегчения работы с приложением и его отладки существует специальное расширение для Chrome - React Devtools. Оно работает аналогично стандартным инструментам разработчика Chrome. Данное расширение устанавливается из магазина расширений Chrome и доступно для всех сайтов, на которых используется React.

После установки данного расширения запустите проект, в открывшейся вкладке откройте инструменты разработчика и перейдите на вкладку Components:



Здесь доступна для просмотра вся иерархия компонентов, созданных в вашем приложении, их названия и некоторые их свойства - например, props.



Webpack & Babel. Минусы create-react-app

Babel

Как уже было указано выше, стандарты фронт-энда и, в частности, стандарты языка JS меняются достаточно часто, иногда быстрее, чем браузеры успевают вносить изменения для соответствия стандартам языка. Кроме того, разработчикам хочется писать код с использованием современных стандартов, но при этом обеспечивать поддержку старых версий браузеров. Для решения этой проблемы создан инструмент Babel - т.н. транспайлер. Перед запуском JS в браузере он преобразует код (напр., с помощью полифиллов) в код, поддерживаемый всеми необходимыми версиями браузеров. Использование Babel позволяет писать современный код с применением, к примеру, optional chaining, async/await или генераторов.

Помимо поддержки старых браузеров, Babel используется в приложениях на React для обеспечения поддержки JSX - к примеру, данный код

```
import React from 'react';

function App() {
  return <h1>Hello World</h1>;
}
```

будет преобразован в такой:

```
import React from 'react';

function App() {
  return React.createElement('h1', null, 'Hello world');
}
```

Внимание!

До 17-й версии React в каждом файле, использующем JSX, было необходимо добавлять строку `'import React from "react";'` даже если мы не использовали напрямую объект React. Это было связано именно с тем, что при транспиляции Babel преобразовывает JSX в React.createElement. В 17-й версии механизм трансформации JSX претерпел некоторые изменения, благодаря чему теперь такой импорт необязателен.

Babel позволяет использование сторонних модулей для поддержки возможности транспиляции самых различных файлов и синтаксисов. Ниже приведен пример файла настроек Babel:

.babelrc

```
{
   "presets": ["@babel/env", "@babel/react"],
   "plugins": ["@babel/plugin-proposal-class-properties"]
}
```

Webpack

Как правило, при работе над js приложением, мы разбиваем наш код на модули, а затем подключаем необходимые модули в соответствующих местах с помощью import или require. Это удобно для разработки, но для запуска такого кода в браузере его следует собрать в один (или несколько) больших файлов (сборок). Для автоматизации этого процесса существуют специальные программы-сборщики, самой популярных из которых является webpack. Webpack - достаточно гибкий и мощный инструмент, имеющий множество настроек.

webpack.config.js

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

module.exports = {
   entry: {
       app: './index.jsx',
   },
   context: path.resolve(__dirname, 'static_src'),
   output: {
       path: path.resolve(__dirname, 'static', 'build'),
       filename: 'bundle.js',
       publicPath: '/static/build/',
   },
   resolve: {
       modules: [`${__dirname}/static_src`, 'node_modules'],
       extensions: ['.js', '.jsx'],
   },
```

```
    module: {
        rules: [
            {
            test: /\.(js|jsx)$/,
            include: path.resolve(__dirname, 'static_src'),
            loader: 'babel-loader',
            exclude: /node_modules/,
        },
        {
            test: /.(css|scss)$/,
            use: ["style-loader", "css-loader", "sass-loader"],
        },
    ],
    },
    devServer: {
        port: 8080,
        historyApiFallback: true,
    },
};
```

Подробно рассматривать работу сборщиков в данном курсе мы не будем.

Недостатки create-react-app

Create-react-app - простой в использовании инструмент, позволяющий через один скрипт выполнить полную настройку React-проекта (включая настройку линтера, тестов, поддержку для файлов стилей и т.д.). Однако, такая простота достигается за счет сокрытия большинства настроек инструментов, требующихся для корректной работы проекта (в частности, Webpack, Babel, Jest). Файлы настроек для этих программ скрыты от разработчика (в проекте, созданном с помощью ручной настройки, эти файлы, как правило, находятся в корне проекта и доступны для редактирования).

В связи с этим выполнение тонкой конфигурации вышеуказанных инструментов представляет собой достаточно трудоемкую задачу - для этого требуется предварительно выполнить команду npm run eject.

Внимание!

Eject выполняет необратимую операцию “разборки” проекта. Не выполняйте ее, если не уверены в том, что делаете. На начальном этапе для изучения работы с самим React достаточно автоматической настройки через create-react-app.

В качестве упражнения самостоятельную настройку проекта можно выполнить в соответствии с этой статьей.

Глоссарий

1. Транспиляция - процесс преобразования кода, как правило, подразумевается преобразование с целью обеспечить поддержку более старых браузеров, либо возможностей, не вошедших в официальный стандарт языка.
2. Полифилл - код, реализующий какую-либо функциональность, которая не поддерживается в некоторых версиях браузеров.
3. Сборка (bundle) - процесс создания одного или нескольких файлов с кодом из нескольких модулей, а также файл, получившийся в результате данного процесса.
4. JSX - расширение JS. Специальный синтаксис, позволяющий описывать необходимое состояние DOM. По сути, представляет собой “синтаксический сахар” для вызова React.createElement.
5. Элемент React - специальный объект, возвращаемый методом React.createElement, как правило, описывается с помощью JSX.
6. Компонент React - функция, возвращающая элемент React либо null, или класс, наследующий от React.Component и реализующий метод render, возвращающий элемент React либо null.
7. Чистая функция - функция, которая возвращает одинаковый результат, когда она вызывается с тем же набором аргументов, а также не имеет побочных эффектов.

Домашнее задание

1. Развернуть новый проект с использованием create-react-app.
2. Создать компонент Message, отображающий переданный ему пропсом текст.
3. Изменить компонент App так, чтобы тот рендерил Message и передавал ему пропсом текст (константу).
4. Стилизовать компоненты через css (при желании можно использовать less или sass, однако для sass нужно дополнительно установить node-sass: документация CRA).
5. Установить расширение React Devtools.

Дополнительные материалы

1. Официальная документация Webpack
2. Официальная документация Babel
3. Статья “Настройка Babel & Webpack для React-проекта”
4. Статья “Чистые функции в javascript”

Используемые источники

1. Официальная документация React
2. Официальная документация create-react-app