

Урок 2

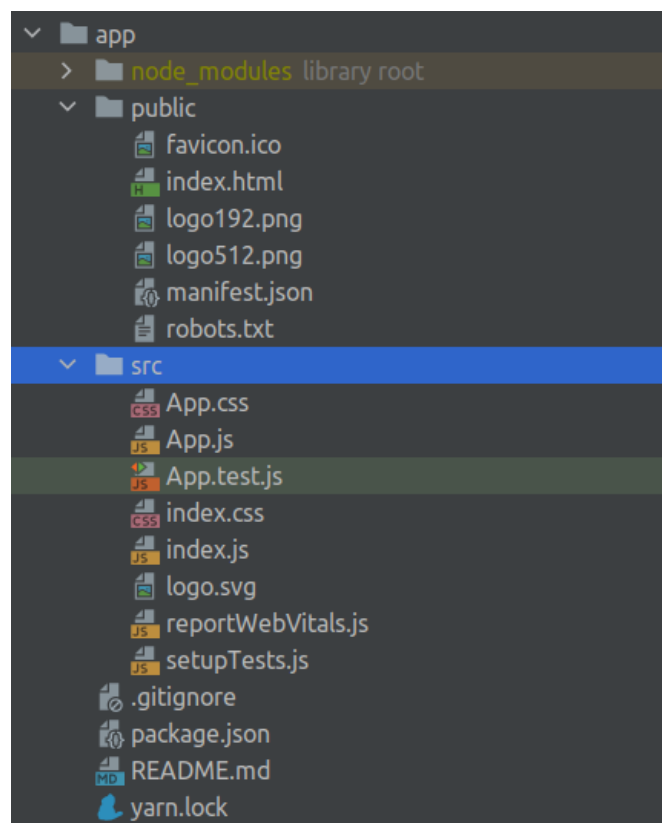
Первые компоненты на React

План урока

1. Структура проекта.
2. Компоненты.
3. JSX.
4. Props.

Структура проекта

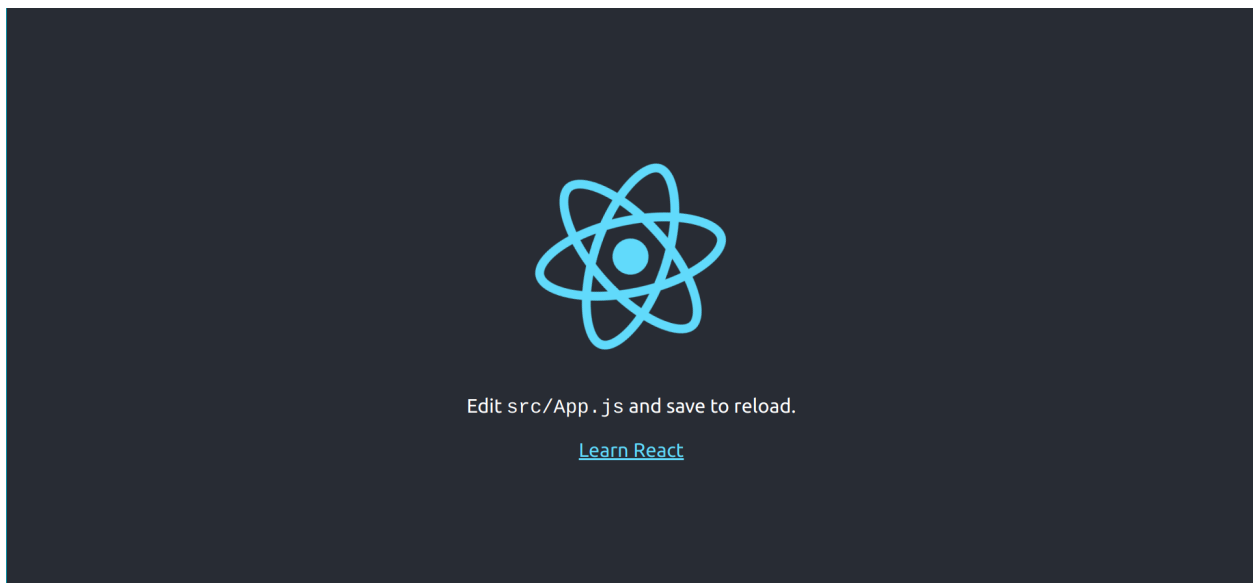
После инициализации шаблона проекта с помощью create-react-app, приложение будет иметь следующую структуру (у вас могут быть незначительные отличия):



Чтобы все было понятно, разберем, за что отвечают эти файлы:

- **node_modules/**: в этой папке помещаются все пакеты, которые вы устанавливаете с помощью `npm install <имя пакета>`. Так как `create-react-app` устанавливает зависимости в процессе создания приложения, в этой папке уже будет какое-то количество установленных пакетов.
- **.gitignore**: указывает, какие файлы и директории не должны попасть в git-репозиторий. На будущее стоит запомнить, что **всегда** нужно помещать в такой файл директорию `node_modules`. В нашем случае `create-react-app` все настроил за нас.
- **package.json**: в этом файле помещается базовая конфигурация приложения: название, версия, команды для запуска или тестирования, а также очень важный блок: список зависимостей проекта. С помощью этого списка другой разработчик, склониравший ваш проект, может установить все нужные пакеты командой `npm install`.
- **public/**: эта директория содержит некоторые статические ресурсы: корневой HTML-файл, `favicon.ico` и несколько картинок.
- **src/**: исходный код приложения, здесь мы располагаем все `js` и `css` файлы.

Результатом выполнения команды `npm start` является запущенное приложение, доступное по адресу <http://localhost:3000>. Если все настроено правильно, вы увидите следующее окно:



Компоненты

Посмотрим на точку входа в наш сайт для браузера — файл `index.html`. В нем можно найти любопытные строчки:

```
<body>
  <noscript>You need to enable JavaScript to run this app.</noscript>
  <div id="root"></div>
  <!--
    This HTML file is a template.
    If you open it directly in the browser, you will see an empty page.

    You can add webfonts, meta tags, or analytics to this file.
    The build step will place the bundled scripts into the <body> tag.

    To begin the development, run `npm start` or `yarn start`.
    To create a production bundle, use `npm run build` or `yarn build`.
  -->
</body>
```

Как же мы видим полностью работающий сайт, если тег **<body>** пуст? Вспомним, что мы говорили об SPA на прошлом уроке — его генерирует JavaScript в момент запуска приложения! Собственно, об этом и говорит строка

“The build step will place the bundled scripts into the `<body>` tag.”

Данный процесс можно проследить в файле `index.js`:



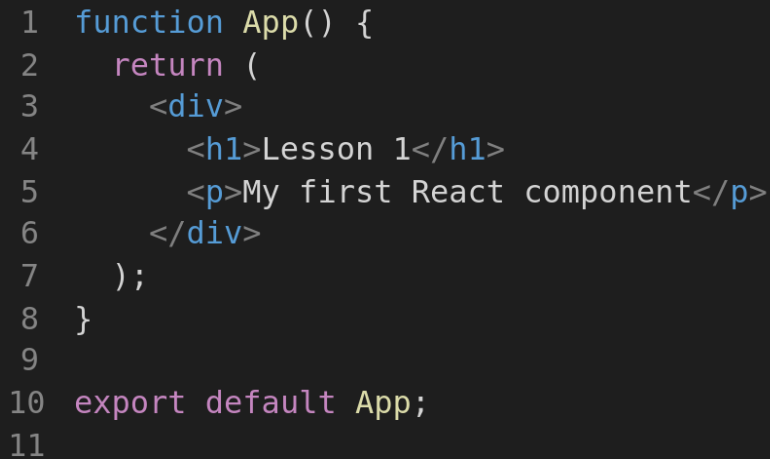
```
1 ReactDOM.render(  
2   <React.StrictMode>  
3     <App />  
4   </React.StrictMode>,  
5   document.getElementById('root')  
6 );
```

Вызывается функция `render` для двух аргументов — что надо отобразить и в каком узле DOM-дерева. В качестве узла используется единственный `<div>` в теге `<body>` из нашего HTML-файла. А вот первый аргумент может показаться очень непривычным на вид. Он не похож на валидный JS-код и чем-то напоминает HTML. Мы вернемся к этому в следующем же разделе.

Напомним, что в React используется компонентный подход: приложение представляет собой композицию компонентов. В сгенерированной заготовке есть только один компонент — `<App />`. Каждый компонент помещается в одноименный js файл для корректной декомпозиции проекта. Посмотрим на содержание файла `App.js`:

```
1 import logo from './logo.svg';
2 import './App.css';
3
4 function App() {
5   return (
6     <div className="App">
7       <header className="App-header">
8         <img src={logo} className="App-logo" alt="logo" />
9         <p>
10           Edit <code>src/App.js</code> and save to reload.
11         </p>
12         <a
13           className="App-link"
14           href="https://reactjs.org"
15           target="_blank"
16           rel="noopener noreferrer"
17         >
18           Learn React
19         </a>
20       </header>
21     </div>
22   );
23 }
24
25 export default App;
26
```

Снова видим файл странного формата, в которой есть единственная функция App(), которая возвращает верстку компонента. Сократим код до минимума, оставив только каркас функции:



```
1 function App() {  
2   return (  
3     <div>  
4       <h1>Lesson 1</h1>  
5       <p>My first React component</p>  
6     </div>  
7   );  
8 }  
9  
10 export default App;  
11
```

Сделаем так, чтобы функция возвращала обычную привычную нам HTML-верстку из нескольких элементов. Так это будет выглядеть на странице:

Lesson 1

My first React component

Компонент App — всего лишь обычная JavaScript функция, которая вызывается каждый раз, когда React решает, что этот компонент нужно отобразить (отрендерить). Компоненты, определенные в виде функций, называются **функциональными**. Компонент можно также создать в виде класса — такие компоненты называются **классовыми**. Их мы рассмотрим позже. Однако необходимо отметить, что с версии React 16.8 (2019 год) стандартом считается именно функциональные компоненты. Причины этого мы тоже затронем чуть позднее.

JSX

Вернемся к структуре файла App.js, а именно к вопросу о верстке в js файле. Такой формат файла — специальная концепция React, она называется **JSX** (JavaScript XML). Документация описывает JSX так: “JSX напоминает язык шаблонов, наделённый силой JavaScript”. Почему же команда React решила “смешать” разметку и логику, если до этого нам говорили помещать HTML, CSS и JS строго в отдельные файлы? Ответ такой: “Вместо того, чтобы искусственно разделить технологии, помещая разметку и логику в разные файлы, React разделяет ответственность с помощью слабо связанных единиц, называемых «компоненты», которые содержат и разметку, и логику”. Если вам хочется лучше прочувствовать логику такого решения, можно посмотреть следующий [доклад](#).

JSX позволяет встраивать в разметку любые корректные JavaScript-выражения, помещая их в фигурные скобки. Более того, после компиляции каждое JSX-выражение становится обычным вызовом JavaScript-функции, результат которого — объект JavaScript. Это значит, что JSX выражение можно присвоить переменной и использовать в дальнейшем.

```
1  const title = 'Lesson 1';
2  const paragraph = <p>My first React component</p>
3
4  function App() {
5    return (
6      <div>
7        <h1>{ title }</h1>
8        { paragraph }
9        <p className="redText">{ '3' + '1' }</p>
10     </div>
11   );
12 }
13
14 export default App;
15
```

Обратите внимание на небольшую, но очень важную деталь: из-за того, что слово **class** является встроенным для языка JavaScript, чтобы назначить класс элементу разметки, необходимо использовать слово **className**. При этом в конечной верстке все будет отображаться как положено:

```
▼ <div id="root">
  ▼ <div>
    <h1>Lesson 1</h1>
    <p>My first React component</p>
    <p class="redText">3-1</p> == $0
  </div>
</div>
```

В одном из предыдущих абзацев мы употребили слово “компиляция”, но что это значит для JS, который является интерпретируемым языком? Дело в том, что браузер понимает только обычный JavaScript, а многие браузеры даже обычный JS понимают только до версии ES6. Поэтому разработчики придумали **Babel** — специальный инструмент для транспиляции (можно сказать, конвертирования) новых синтаксических конструкций в старые, понятные всем браузерам. Тем самым программисты могут писать современный код, не задумываясь над вопросами совместимости.

Babel компилирует JSX в вызовы функций `React.createElement()`. Поэтому “под капотом” первый пример преобразуется во второй:

```
1  const title = <h1 className="app-title">Lesson 2</h1>
2
3  const title = React.createElement(
4    'h1',
5    { className: 'app-title' },
6    'Lesson 2'
7  );
```

Таким образом, JSX является удобной оберткой для создания React элементов.

Props

Компоненты могут принимать произвольные входные параметры (properties, сокращенно props, на русском употребляют “пропсы”), от которых зависят детали отображения. Благодаря пропсам компоненты можно делать универсальными и переиспользуемыми.

Создадим страницу со списком уроков данного курса. Заходим в файл App.js и пишем что-то вроде:

```
1 function App() {
2   return (
3     <div className="container">
4       <h1>Lessons list</h1>
5       <div className="lesson-container">
6         <h2 className="lesson-header">Lesson 0</h2>
7         <h4 className="lesson-title">Prerequisites</h4>
8         <p className="lesson-date">05.10.2021</p>
9       </div>
10      <div className="lesson-container">
11        <h2 className="lesson-header">Lesson 1</h2>
12        <h4 className="lesson-title">Intro to React</h4>
13        <p className="lesson-date">06.10.2021</p>
14      </div>
15      <div className="lesson-container">
16        <h2 className="lesson-header">Lesson 2</h2>
17        <h4 className="lesson-title">First React components</h4>
18        <p className="lesson-date">06.10.2021</p>
19      </div>
20      <div className="lesson-container">
21        <h2 className="lesson-header">Lesson 3</h2>
22        <h4 className="lesson-title">Hooks, events</h4>
23        <p className="lesson-date">13.10.2021</p>
24      </div>
25    </div>
26  );
27 }
28
29 export default App;
30
```

Однако это не очень гибко (вдруг на курсе будет 100 уроков), поэтому выделим урок в отдельный компонент. Создадим файл Lesson.js:

```
1 function Lesson(props) {
2   return (
3     <div className="lesson-container">
4       <h2 className="lesson-header">{ props.title }</h2>
5       <h4 className="lesson-title">{ props.name }</h4>
6       <p className="lesson-date">{ props.date }</p>
7     </div>
8   );
9 }
10
11 export default Lesson;
12
```

Функциональный компонент принимает параметр-объект props и “вытаскивает” из него поля title, name и date. Используя механизм **деструктуризации**, упомянутый в нулевом уроке, компонент можно переписать:

```
1 function Lesson({ title, name, date }) {
2   return (
3     <div className="lesson-container">
4       <h2 className="lesson-header">{ title }</h2>
5       <h4 className="lesson-title">{ name }</h4>
6       <p className="lesson-date">{ date }</p>
7     </div>
8   );
9 }
10
11 export default Lesson;
12
```

Теперь все, что нам нужно, чтобы создать множество компонентов Lesson, — это передать нужные свойства в каждый экземпляр. React соберет все переданные свойства в объект props и передаст его как параметр в компонент.

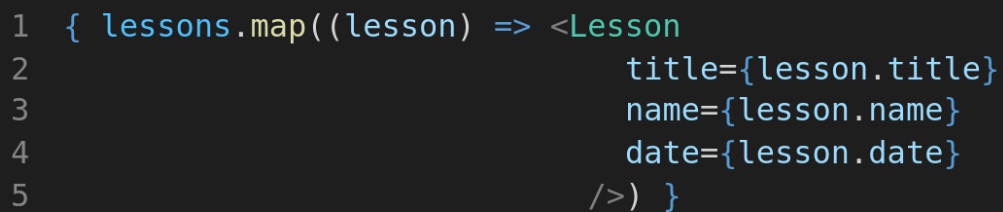
```
1 import Lesson from "./Lesson";
2
3 function App() {
4   return (
5     <div className="container">
6       <h1>Lessons list</h1>
7       <Lesson title="Lesson 0" name="Prerequisites" date="05.10.2021" />
8       <Lesson title="Lesson 1" name="Intro to React" date="06.10.2021" />
9       <Lesson title="Lesson 2" name="First React components" date="06.10.2021" />
10      <Lesson title="Lesson 3" name="Hooks, events" date="13.10.2021" />
11    </div>
12  );
13 }
14
15 export default App;
16
```

Но мы еще не полностью избавились от однообразного кода: если на курсе будет 100 уроков, нам все равно придется писать сто строк для ста компонентов. К тому же, обычно подобную информацию мы получаем откуда-то из бэкенда или внешнего API, а не хардкодим на клиенте. Создадим массив уроков и отрисуем все уроки в цикле:



```
1 import Lesson from "./Lesson";
2
3 const lessons = [
4   {
5     title: "Lesson 0",
6     name: "Prerequisites",
7     date: "05.10.2021"
8   },
9   {
10    title: "Lesson 1",
11    name: "Intro to React",
12    date: "06.10.2021"
13  },
14  {
15    title: "Lesson 2",
16    name: "First React components",
17    date: "06.10.2021"
18  },
19  {
20    title: "Lesson 3",
21    name: "Hooks, events",
22    date: "13.10.2021"
23  },
24 ];
25
26 function App() {
27   return (
28     <div className="container">
29       <h1>Lessons list</h1>
30       { lessons.map((lesson) => <Lesson {...lesson} />) }
31     </div>
32   );
33 }
34
35 export default App;
36
```

Так как в скобки можно поместить любое JavaScript-выражение, нет причин, чтобы не использовать там `map()`, который для каждого элемента списка вернет компонент `Lesson`. При передаче пропсов используем оператор `rest`, чтобы не писать отдельно в более многословном виде:



```
1 { lessons.map((lesson) => <Lesson
2                               title={lesson.title}
3                               name={lesson.name}
4                               date={lesson.date}
5                               />) }
```