

JS Level 3



при поддержке



REACT ELEMENT

React

На прошлой лекции мы создали с вами наш первый компонент. В этой лекции пришло время разобраться, что из себя представляют компоненты и как всё устроено.

Начнём мы с `package.json` и посмотрим, что у нас подключается:

```
"dependencies": {  
  "@testing-library/jest-dom": "^4.2.4",  
  "@testing-library/react": "^9.3.2",  
  "@testing-library/user-event": "^7.1.2",  
  "react": "^16.14.0",  
  "react-dom": "^16.14.0",  
  "react-scripts": "3.4.3"  
},
```

библиотеки для тестирования

Сегодня нас будут интересовать последние три зависимости.

React

1. react - это общие возможности по созданию интерфейсов
2. react-dom - это интеграция с DOM
3. react-scripts - это удобный набор скриптов для использования с npm

Почему react и react-dom разнесены отдельно? Просто потому, что React можно использовать не только в веб, но и в мобильных приложениях (React Native), и там будет уже не react-dom, а react-native.

create-react-app

create-react-app (CRA), который мы использовали, позволяет за нас создать типовую структуру проекта и обеспечить всем необходимым для запуска, тестирования и сборки нашего приложения.

О чём идёт речь? CRA (через react-scripts) предоставляет уже готовые команды:

```

> Debug
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "eject": "react-scripts eject"
},

```

- start ([npm start](#)) - запускает сервер для разработки
- build ([npm run build](#)) - "собирает" приложение (оно готово для запуска)
- test ([npm test](#)) - запускает авто-тесты
- eject ([npm run eject](#)) - выгружает конфигурацию Webpack

create-react-app

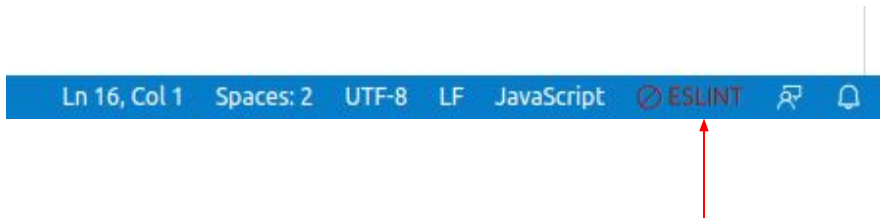
В шаблоне проекта уже настроен Webpack, Jest (инструмент авто-тестирования), ESLint (инструмент проверки стиля кодирования) и т.д.

Поэтому это очень удобный инструмент для старта.

ESLint

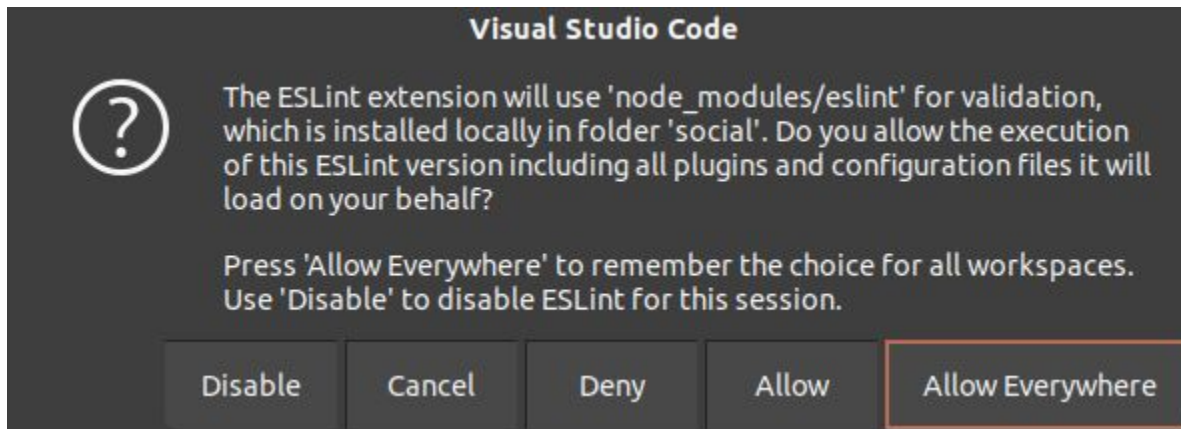
Чтобы писать более чистый код, необходимо использовать ESLint (в частности, бот будет проверять, что вы его используете).

Для включения автоподсказок, необходимо в нижней правой части редактора кликнуть на ESLint:



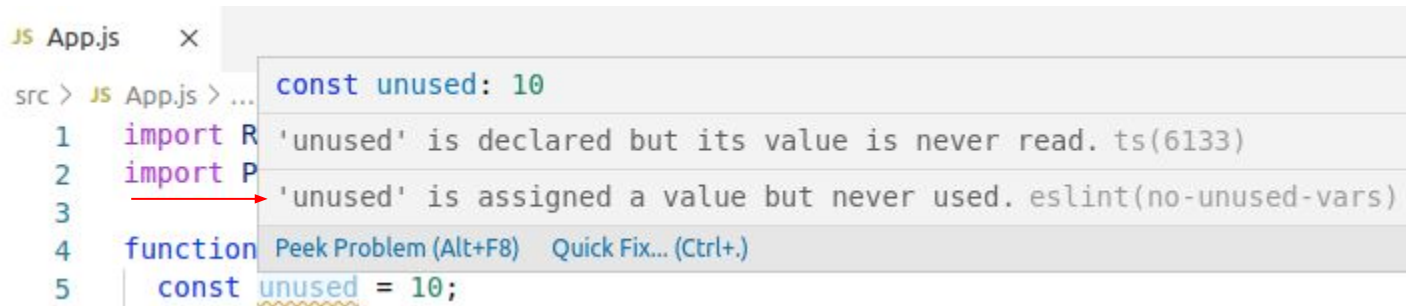
ESLint

После этого VS Code вас спросит, разрешать ли запуск ESLint:



ESLint

Теперь, если мы будем допускать какие-то стилистические ошибки (писать не очень чистый код), ESLint будет нам об этом подсказывать:

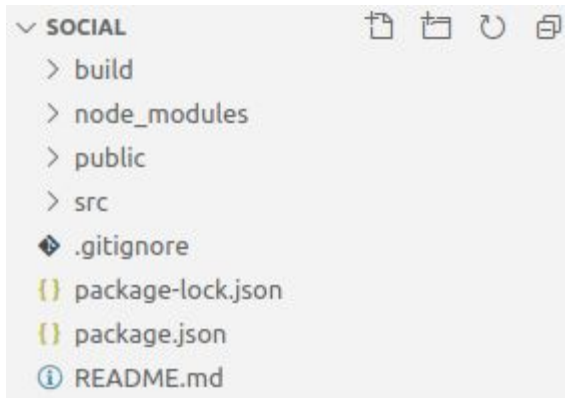


The screenshot shows a code editor with a file named 'App.js'. The code contains the following lines:

```
1 import R from 'react';
2 import P from 'prop-types';
3
4 function App() {
5   const unused = 10;
```

A red arrow points to line 3, where an ESLint error is displayed. The error message is: "'unused' is assigned a value but never used. eslint(no-unused-vars)". The error is highlighted in a light gray box. Below the error message, there are two links: "Peek Problem (Alt+F8)" and "Quick Fix... (Ctrl+.)".

Структура каталогов



- build - собранное приложение (появится, если вы в консоли запустите `npm run build`)
- node_modules - установленные зависимости
- public - статичные файлы
- src - исходники

npm

У вас может возникнуть вопрос: почему некоторые команды npm мы пишем просто `npm start`, `npm test`, а другие `npm run build`, `npm run eject`?

Дело в том, что в самом npm определён ряд типовых команд, которые можно запускать без слова `run`, например:

- `start`
- `test`

Полный их список приведён [на странице документации](#). В остальных же случаях, когда мы (либо другой инструмент) пишем названия скриптов не из этого списка, нужно писать `npm run`, например, `npm run build`.

public

В каталоге public располагаются статичные файлы вроде:

- иконка в формате ico
- index.html
- логотипы в формате png
- manifest.json - информация о том, как отображать иконку при установке на моб. устройство
- robots.txt - информация для поисковых роботов

При публикации приложения вам желательно пройтись по всем этим файлам и демо-значения вроде React App и т.д. заменить на собственные.

index.html head

Давайте посмотрим на index.html. head:

```
<head>
  <meta charset="utf-8" />
  <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <meta name="theme-color" content="#000000" />
  <meta
    name="description"
    content="Web site created using create-react-app"
  />
  <link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png" />
  <!--
    manifest.json provides metadata used when your web app is installed on a
    user's mobile device or desktop. See https://developers.google.com/web/fundamentals/web-app-manifest/
  -->
  <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
  <!--
    Notice the use of %PUBLIC_URL% in the tags above.
    It will be replaced with the URL of the `public` folder during the build.
    Only files inside the `public` folder can be referenced from the HTML.

    Unlike "/favicon.ico" or "favicon.ico", "%PUBLIC_URL%/favicon.ico" will
    work correctly both with client-side routing and a non-root public URL.
    Learn how to configure a non-root public URL by running `npm run build`.
  -->
  <title>React App</title>
</head>
```

index.html head

`%PUBLIC_URL%` в процессе сборки будет заменён на путь, который мы укажем (по умолчанию - пустая строка).

Детально о сборке вы можете почитать в Level 2, в разделе про Webpack.

Мы увидим это в действии, когда будем публиковать наше приложение на GitHub Pages.

index.html body

В body нашей страницы ещё меньше:

```
<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>
```

noscript будет обрабатываться тогда, когда у пользователя в браузере отключен JS, а div#root будет использоваться в качестве корневого элемента для построения DOM-дерева.

index.html

Как вы видите, никаких вставок JS в index.html нет. Просто потому, что вставку производит уже Webpack в процессе сборки.

index.html

Выполните в терминале команду `npm run build`:

```
$ npm run build
```

```
> social@0.1.0 build /projects/social  
> react-scripts build
```

```
Creating an optimized production build...  
Compiled successfully.
```

File sizes after gzip:

39.38 KB	build/static/js/2.c5aefca7.chunk.js
770 B	build/static/js/runtime-main.ef23f142.js
535 B (-52 B)	build/static/js/main.aebe0e14.chunk.js
278 B	build/static/css/main.5ecd60fb.chunk.css

The project was built assuming it is hosted at `/`.
You can control this with the `homepage` field in your `package.json`.

The `build` folder is ready to be deployed.
You may serve it with a static server:

```
npm install -g serve  
serve -s build
```

Find out more about deployment here:

bit.ly/CRA-deploy

Теперь если вы откроете `index.html` уже в каталоге `build`, то увидите, что там будет JS, который и подключил Webpack.

index.js

Завершая наш обзор, зайдём в каталог src файл index.js. index.js - это точка, начиная с которой собирается наше приложение. И именно здесь содержится код, который будет запускать наше приложение:

```
JS index.js ×
src > JS index.js
1  import React from 'react';
2  import ReactDOM from 'react-dom';
3  import './index.css';
4  import App from './App';
5  import * as serviceWorker from './serviceWorker';
6
7  ReactDOM.render(
8    <React.StrictMode>
9      <App />
10    </React.StrictMode>,
11    document.getElementById('root')
12  );
```

index.js

Давайте разберём подробнее вот эту часть:

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

[ReactDOM.render](#) фактически добавляет детей в `div#root` и следит за тем, чтобы вовремя обновлять их. Что значит обновлять?

Давайте копнём чуть глубже и попробуем разобраться, что же такое `<App />` например.

React Element

Заменим код на следующий (надеемся, что вы не забыли сделать npm start):

```
ReactDOM.render(  
  React.createElement(App),  
  document.getElementById('root')  
);
```

А потом снова на этот:

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

Разницы на странице и в DOM-дереве вы не увидите. Почему и что это такое?

JSX

React использует специальное расширение к синтаксису JS, которое называется JSX. Это расширение позволяет заменять вызовы `React.createElement(App)`, где `App` - это функция нашего компонента, на короткий и удобный `<App />`. Это преобразование прозрачно для нас выполняется специальным инструментом, который называется Babel (а именно плагином к нему).

Ключевое, что вам надо запомнить: любая запись вида `<App />` (например, `<Post />`) преобразуется в `React.createElement`, поэтому все ограничения JSX будут следовать из этого.

Кстати, именно поэтому мы обязаны в каждом файле компонента импортировать React, хоть нигде его явно не используем (он будет использоваться после конвертации JSX в JS).

JSX

Важно запомнить, что JSX - это не HTML! То, что разрешено в HTML не обязательно разрешено в JSX (JSX гораздо строже).

Элемент vs Компонент

До этого мы использовали термины и компонент, и элемент. Давайте разберёмся, чем они отличаются. Компонент - это такой строительный блок, который используется как шаблон для создания элемента.

Из одного компонента можно сделать сколько угодно элементов (например, компонент `Post` у нас был один, а вот элементов мы создали целых три).

Напоминаем, что элемент - это любой `React.createElement` или `<ElementName />`.

built-in компоненты

Есть компоненты, которые мы пишем с вами сами - они всегда пишутся с большой буквы, например App, Posts, а есть уже встроенные в React, например, div, span. Они пишутся с маленькой и их название заключается в кавычки: `React.createElement('div');`

DOM vs React

А теперь самое важное: у нас есть DOM элементы и React элементы. Давайте посмотрим, чем они отличаются:

```
console.dir(document.createElement('div'));  
console.dir(React.createElement('div'));
```

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

```
► div  
▼ Object ⓘ  
  $$typeof: Symbol(react.element)  
  key: null  
  ► props: {}  
    ref: null  
    type: "div"  
    _owner: null  
  ► _store: {validated: false}  
    _self: null  
    _source: null  
  ► __proto__: Object
```

Как вы видите, React Element достаточно небольшой, компактный объект. В то же время, если вы откроете div (DOM Element), то он будет просто огромным.

Virtual DOM

Дело в том, что React работает следующим образом: он описывает всё в виде дерева React Element'ов и после каждого изменения сравнивает, как дерево выглядело до изменения и как после (представьте, что он сравнивает то, что возвращают наши функции компонентов). Как только он находит различия, он эти самые различия применяет к реальному DOM-дереву. Такой подход называется Virtual DOM.

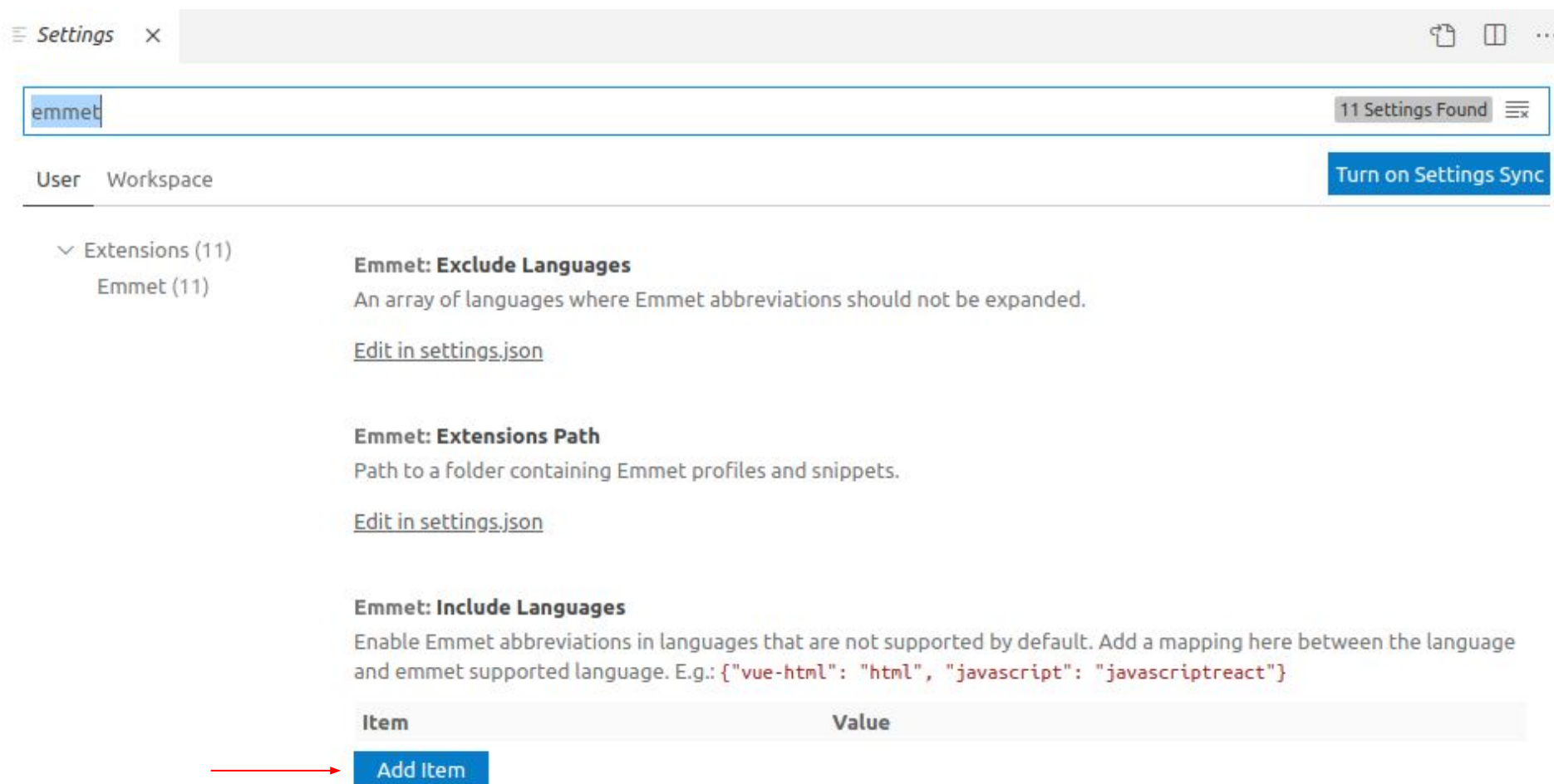
У этого подхода есть важная особенность: React видит только изменения своих элементов. Если вы напрямую что-то поменяете в DOM-дереве, то React этого может не заметить (обновит только тогда, когда в результате сравнения Virtual DOM примет решение обновить реальный DOM).

Emmet

Emmet - это специальный инструмент, позволяющий вам создавать разметку в разы быстрее. Посмотрите <https://docs.emmet.io/cheat-sheet/>, чтобы понять о чём речь.

Emmet

Для включения поддержки Emmet в JS файлах, необходимо нажать **Ctrl + ,** и в поисковой строке ввести `emmet`:



The screenshot shows the VS Code Settings interface. The search bar at the top contains the text "emmet", and a notification indicates "11 Settings Found". The left sidebar shows the "Settings" tab selected, with "Extensions (11)" and "Emmet (11)" listed. The main content area displays three Emmet-related settings:

- Emmet: Exclude Languages**
An array of languages where Emmet abbreviations should not be expanded.
[Edit in settings.json](#)
- Emmet: Extensions Path**
Path to a folder containing Emmet profiles and snippets.
[Edit in settings.json](#)
- Emmet: Include Languages**
Enable Emmet abbreviations in languages that are not supported by default. Add a mapping here between the language and emmet supported language. E.g.: `{"vue-html": "html", "javascript": "javascriptreact"}`

At the bottom, there is a table with two columns: "Item" and "Value". Below the table is a blue button labeled "Add Item", which is pointed to by a red arrow.

Item	Value
Add Item	

Emmet

После чего ввести javascript и javascriptreact:



Emmet: Include Languages

Enable Emmet abbreviations in languages that are not supported by default. Add a mapping here between the language and emmet supported language. E.g.: `{"vue-html": "html", "javascript": "javascriptreact"}`

Item	Value
javascript	javascriptreact

OK

Cancel



POSTS

Компонент

Давайте разбираться с JSX и постами. Первое, что мы сделаем - это научимся отделять данные от отображения. О чём идёт речь? Раньше мы передавали в наш компонент только строку.

Но ведь пост - это больше, чем строка. У него есть id, автор, дата публикации, контент, количество лайков, комментарии, теги и т.д.

А кроме того, постов у нас явно не один.

Компонент

Но давайте делать всё по-порядку. Научимся отображать один комплексный объект с помощью компонента Post.

Итак верните ваш код в index.js в следующий вид:

```
JS index.js  X
src > JS index.js
1  import React from 'react';
2  import ReactDOM from 'react-dom';
3  import './index.css';
4  import App from './App';
5  import * as serviceWorker from './serviceWorker';
6
7  ReactDOM.render(
8    <React.StrictMode>
9    |   <App />
10   </React.StrictMode>,
11   document.getElementById('root')
12 );
13
14 // If you want your app to work offline and load faster, you can change
15 // unregister() to register() below. Note this comes with some pitfalls.
16 // Learn more about service workers: https://bit.ly/CRA-PWA
17 serviceWorker.unregister();
```


Компонент

Теперь отредактируем App.js:

```
function App() {  
  const post = {  
    id: 1,  
    author: {  
      id: 1,  
      avatar: 'https://lms.openjs.io/logo_js.svg',  
      name: 'OpenJS',  
    },  
    content: null,  
    photo: 'https://lms.openjs.io/openjs.jpg',  
    hit: true,  
    likes: 100,  
    likedByMe: true,  
    created: 1603501200,  
  },  
  
  return (  
    <div className="App">  
      <Post post={post} />  
    </div>  
  );  
}
```

```
export default App;
```

Ключевые моменты:

1. Данные мы храним отдельно в переменной
2. Эти данные мы передаём в компонент

Обратите внимание, как мы их передаём (через {}).

{ vs ""

Но до этого мы передавали вот так:

```
function App() {  
  return (  
    <div className="App">  
      <Post content="Первый пост" />  
      <Post content="Второй пост" />  
      <Post content="Третий пост" />  
    </div>  
  );  
}
```

В чём же разница? Дело в том, что через "" можно передавать только строки. Т.е. если мы напишем `<Post post="post" />` то в props компонента `Post` будет передана строка `"post"`, а не объект, который хранится в переменной `post`.

`{}` же позволяют нам передавать объект. Правило достаточно простое - если вы хотите передать просто строку - используйте "", во всех остальных случаях - {}.

Компонент

Для начала попробуем отобразить аватарку автора:

```
JS Post.js  X
src > components > Post > JS Post.js > ...
1  import React from 'react'
2
3  function Post(props) {
4    return (
5      <article>
6        <header>
7          <img src={props.post.author.avatar} width="50" height="50" alt={props.post.author.name} />
8        </header>
9      </article>
10   )
11 }
12
13 export default Post
```

В данном случае - и `avatar`, и `name` - это строки. Но поскольку нам нужно отображать значение этих полей, то мы используем `{}`.

Компонент

Так работать с props достаточно неудобно (получается слишком длинная запись).

Поэтому сократим её с помощью деструктуризации:

```
function Post({post}) {  
  const {author} = post;  
  
  return (  
    <article>  
      <header>  
        <img src={author.avatar} width="50" height="50" alt={author.name}/>  
      </header>  
    </article>  
  )  
}
```

Эта запись с фигурными скобками (в аргументах и в переменной) говорит о том, что мы создаём имя, извлекая соответствующее поле. В случае `Post({post})` - `post` извлекается из props, а в случае `const {author} = post`, `author` извлекается из `post`.

Компонент

Давайте скруглим аватарку. Для этого у нас следующий вариант - прописать в CSS свойство `border-radius` для аватарки.

Но в какой CSS писать? У нас есть "глобальный" `index.css`, а также `App.css`. Давайте создадим свой `Post.css` и там разместим определение класса:



```
JS Post.js  # Post.css  X  
src > components > Post > # Post.css > ...  
1  .Post-avatar {  
2    |   border-radius: 50%;  
3  }
```

CSS

А затем подключим этот CSS к нашему компоненту:

```
JS Post.js  ×  # Post.css

src > components > Post > JS Post.js > ...
1  import React from 'react';
2  import './Post.css';
3
4  function Post({post}) {
5    const {author} = post;
6
7    return (
8      <article>
9        <header>
10         <img src={author.avatar} className="Post-avatar" width="50" height="50" alt={author.name}/>
11        </header>
12      </article>
13    )
14  }
15
16  export default Post
```



CSS

Как это работает? Почему мы импортируем CSS в JS? За это отвечает Webpack. Когда он видит подобную конструкцию (`import './Post.css'`) он загружает и подключает CSS к нашему проекту.

А затем уже через `className` мы используем объявленный CSS-класс.

Автор и время публикации

Мы пока не будем заниматься особо вёрсткой, поэтому просто выведем имя автора и время публикации (его тоже пока не будем конвертировать):

```
JS Post.js  X
src > components > Post > JS Post.js > ...
1  import React from 'react';
2  import './Post.css';
3
4  function Post({post}) {
5    const {author} = post;
6
7    return (
8      <article>
9        <header>
10         <img src={author.avatar} className="Post-avatar" width="50" height="50" alt={author.name}/>
11         <h5>{author.name}</h5>
12         <div>{post.created}</div>
13       </header>
14     </article>
15   )
16 }
17
18 export default Post
```


HIT

А теперь выведем бейджик **HIT**, если у поста установлено свойство hit в true. Как это сделать? Если мы попробуем поставить if, то получим ошибку:

```
return (  
  <article>  
    <header>  
      <img src={author.avatar} className="Post-avatar" width="50" height="50" alt={author.name}/>  
      <h5>{author.name}</h5>  
      <div>{post.created}</div>  
      <span>{if (post.hit) 'HIT'}</span>  
    </header>  
  </article>  
)
```

Error

Обратите внимание, об ошибке вам будут сообщать в консоли VS Code:

Failed to compile.

`./src/components/Post/Post.js`

Line 13:16: Parsing error: Unexpected token

```
11 |         <h5>{author.name}</h5>
12 |         <div>{post.created}</div>
> 13 |         <span>{if (post.hit) 'HIT'}</span>
    |                   ^
14 |     </header>
15 | </article>
16 | )
```

□

На самой странице:

Failed to compile

`./src/components/Post/Post.js`

Line 13:16: Parsing error: Unexpected token

```
11 |         <h5>{author.name}</h5>
12 |         <div>{post.created}</div>
> 13 |         <span>{if (post.hit) 'HIT'}</span>
    |                   ^
14 |     </header>
15 | </article>
16 | )
```

Error

При этом символами **> ^** будут отмечать точное местоположение ошибки:

Failed to compile.

`./src/components/Post/Post.js`

Line 13:16: Parsing error: Unexpected token

```
11 |         <h5>{author.name}</h5>
12 |         <div>{post.created}</div>
> 13 |         <span>{if (post.hit) 'HIT'}}</span>
    |                   ^
14 |     </header>
15 | </article>
16 | )
    |
```

Поэтому внимательно смотрите, что вы сделали не так.

if

Почему нельзя писать `if`? Потому что вы не можете написать `if` в вызове функции (вспоминайте предыдущие слайды):

```
React.createElement(if (post.hit) ...);
```

Это запрещено правилами JS.

if

Что же остаётся? Первое и самое простое - [React.createElement](#) возвращает объект, поэтому мы можем просто сделать вот так:

```
function Post({post}) {  
  const {author} = post;  
  
  let hit = '';  
  if (post.hit) {  
    hit = 'HIT';  
  }  
  
  return (  
    <article>  
      <header>  
        <img src={author.avatar} className="Post-avatar" width="50" height="50" alt={author.name}/>  
        <h5>{author.name}</h5>  
        <div>{post.created}</div>  
        <span>{hit}</span>  
      </header>  
    </article>  
  )  
}
```

if

Но это не совсем здорово, потому что зачем нам `span`, если у нас не будет бейджика **HIT**? Мы можем это сделать достаточно просто, если воспользуемся тернарным оператором: `<expr> ? <true> : <false>`. Т.е. проверяется условие, если оно `truthy` (см. лекции JS Level 1) возвращается `<true>`, если `falsy`, то `<false>`.

```
const hit = post.hit ? <span>HIT</span> : null;
```

```
return (  
  <article>  
    <header>  
      <img src={author.avatar} className="Post-avatar" width="50" height="50" alt={author.name}/>  
      <h5>{author.name}</h5>  
      <div>{post.created}</div>  
      {hit}  
    </header>  
  </article>  
)
```

В React, если вы вместо подставляете `null` в выражение в фигурных скобках (либо возвращаете из компонента), то компонент просто не отрисовывается.

if

А теперь попробуем это использовать сразу в выражении:

```
function Post({post}) {  
  const {author} = post;  
  
  return (  
    <article>  
      <header>  
        <img src={author.avatar} className="Post-avatar" width="50" height="50" alt={author.name}/>  
        <h5>{author.name}</h5>  
        <div>{post.created}</div>  
        {post.hit ? <span>HIT</span> : null}  
      </header>  
    </article>  
  )  
}
```

Уже лучше. Поскольку выражение достаточно простое, можно его сразу прописать в JSX ({} понимают любые JS выражения). Если бы оно было достаточно сложным, то стоило бы вынести его в отдельную переменную.

&&

Но есть и более профессиональный способ: оператор `&&`. Как он работает в выражении `a && b`:

- если `a` - falsy, он возвращает `a`
- если `a` - truthy, а `b` - false, он возвращает `b`
- если `a` и `b` - truthy, то он возвращает `b`

```
return (  
  <article>  
    <header>  
      <img src={author.avatar} className="Post-avatar" width="50" height="50" alt={author.name}/>  
      <h5>{author.name}</h5>  
      <div>{post.created}</div>  
      {post.hit && <span>HIT</span>}  
    </header>  
  </article>  
)
```

React смотрит на то значение, которое возвращается и если оно не falsy, то пытается отрисовать элемент.

ИТОГИ

ИТОГИ

В этой лекции мы обсудили достаточно много важных моментов и верхнеуровнево разобрали основы работы React и JSX.

Этих знаний вам будет достаточно, чтобы доделать карточку поста.

ДОМАШНЕЕ ЗАДАНИЕ

ДЗ: Post

Вам нужно отрисовать карточку поста в соответствии со следующими условиями:

```
const post = {  
  id: 1,  
  author: {  
    id: 1,  
    avatar: 'https://lms.openjs.io/logo_js.svg',  
    name: 'OpenJS',  
  },  
  content: null,  
  photo: 'https://lms.openjs.io/openjs.jpg',  
  hit: true,  
  likes: 100,  
  likedByMe: true,  
  created: 1603501200,  
}
```

Д3: Post

Отрисовывается вот в такое DOM-дерево:

```
▼<article>
  ▼<header>
    
    <h5>OpenJS</h5>
    <div>1603501200</div>
    <span>HIT</span>
  </header>
  ▼<div>
    <div class="Post-content"></div>
    
  </div>
  ▼<footer>
    ▼<span class="Post-likes">
      
      <span class="Post-likes-count">100</span>
    </span>
  </footer>
</article>
```

ДЗ: Post

Если данные вот такие:

```
const post = {  
  id: 9,  
  author: {  
    id: 99,  
    avatar: 'https://lms.openjs.io/logo_js.svg',  
    name: 'OpenJS',  
  },  
  content: 'Наше крутое лого',  
  photo: 'https://lms.openjs.io/openjs.jpg',  
  hit: false,  
  likes: 1000,  
  likedByMe: false,  
  created: 1603501200,  
}
```

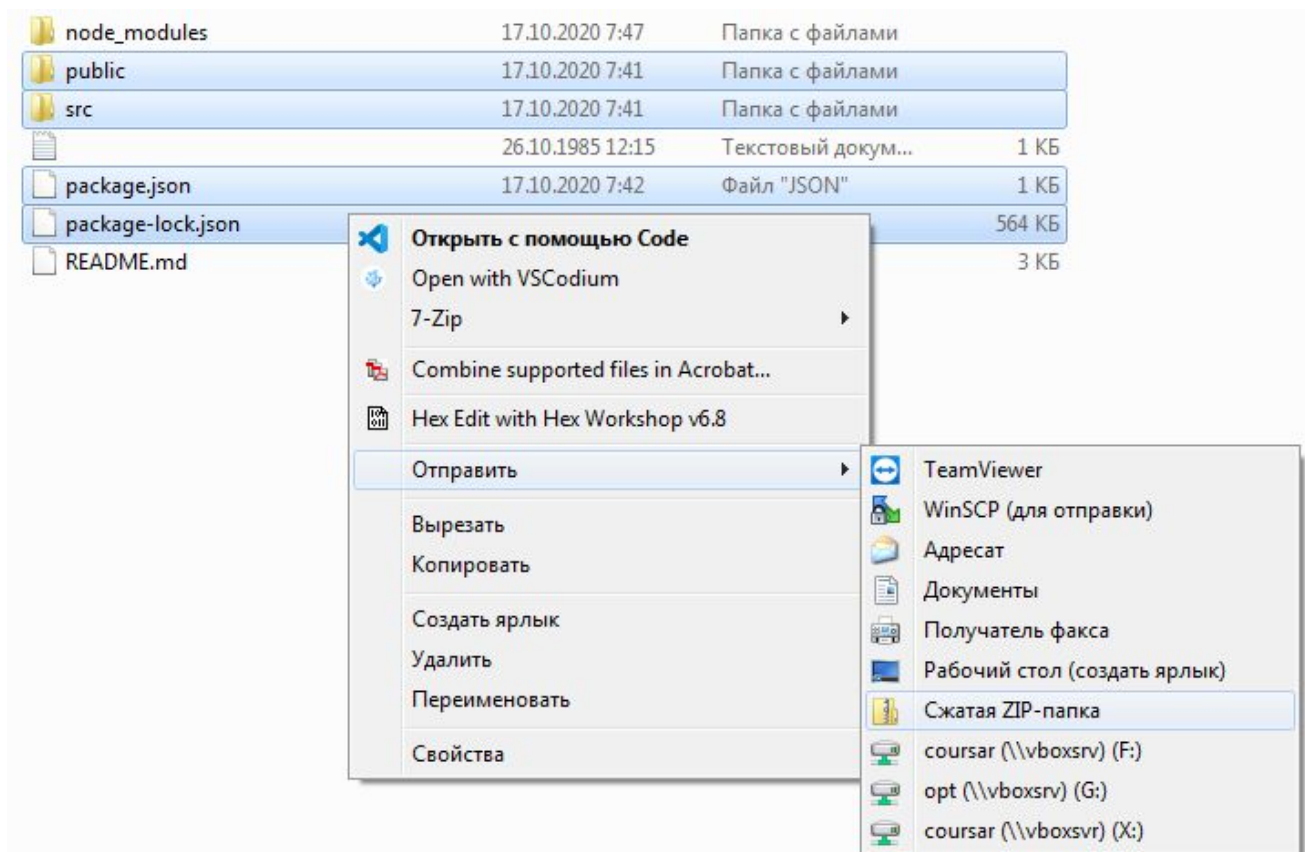
Д3: Post

То пост отрисовывается вот в такое DOM-дерево:

```
▼<article>
  ▼<header>
    
    <h5>OpenJS</h5>
    <div>1603501200</div>
  </header>
  ▼<div>
    <div class="Post-content">Наше крутое лого</div>
    
  </div>
  ▼<footer>
    ▼<span class="Post-likes">
      
      <span class="Post-likes-count">1000</span>
    </span>
  </footer>
</article>
```

Как сдавать ДЗ

Вам нужно запаковать в zip-архив ваш проект те файлы и каталоги, которые указаны на скриншоте ниже. Для этого выберите их, нажмите правую кнопку мыши и выберите Отправить → Сжатая ZIP-папка:



Как сдавать ДЗ

Полученный архив загружаете в личном кабинете пользователя.

Важно: учитывается только последняя отправленная попытка.

Спасибо за внимание

alif academy совместно с aims
2020г.