

БИБЛИОТЕКА REACT, КОМПОНЕНТЫ



АНТОН СТЕПАНОВ



АНТОН СТЕПАНОВ

Ведущий фронтэнд разработчик в StepIntegrator



[@anton_mesmer](https://www.instagram.com/anton_mesmer)



ПЛАН ЗАНЯТИЯ

1. [Библиотека React](#)
2. [Компонентный и декларативный подходы](#)
3. [Компоненты](#)
4. [Инструментарий](#)
5. [Установка](#)
6. [JSX](#)
7. [PropTypes](#)
8. [Class-based компоненты](#)



БИБЛИОТЕКА REACT



БИБЛИОТЕКА REACT

Библиотека **React** - инструмент для создания интерфейсов, основанный на компонентном подходе.


React часто называют фреймворком, ввязываясь в жаркие споры - фреймворк это уже или библиотека, мы же оставим этот вопрос в стороне, отметив лишь, что в рамках курса будем называть его библиотекой.

Официальная страница проекта: <https://reactjs.org>.



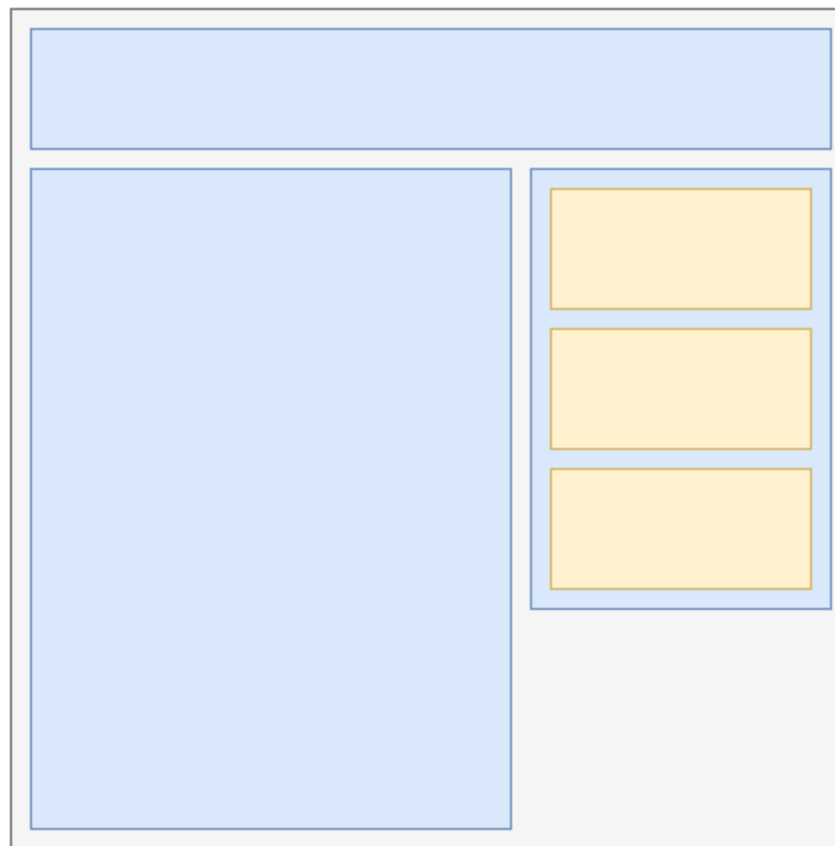
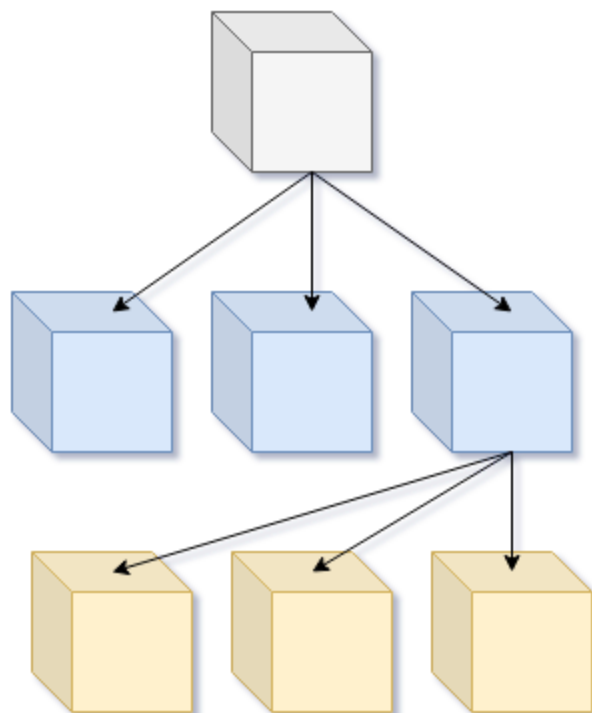
КЛЮЧЕВЫЕ МОМЕНТЫ

- React создан и поддерживается Facebook (выпущен 2013 года)
- используется для создания динамичных веб-интерфейсов
- использует декларативный и компонентный подходы
- является самым популярным инструментом в индустрии на 2019 год (обгоняя Angular и Vue)



КОМПОНЕНТНЫЙ И ДЕКЛАРАТИВНЫЙ ПОДХОДЫ

КОМПОНЕНТНЫЙ ПОДХОД





КОМПОНЕНТНЫЙ ПОДХОД

Основная идея компонентного подхода - создание переиспользуемых компонентов, которые могут выстраиваться в иерархию (согласно вложенности) и взаимодействовать друг с другом.

Напоминает идеи DOM, не правда ли?



КОМПОНЕНТНЫЙ ПОДХОД

Фактически, нам предоставляют инструмент, позволяющий создавать "кастомные элементы", которые могут:

- обладать собственным состоянием и поведением
- содержать дочерние элементы

Кроме того, эти элементы будут выглядеть как настоящие элементы (т.е. обладать собственными тегами) и их можно использовать в разметке*.

Примечание*: не в обычной HTML-разметке, а в некотором аналоге, но об этом чуть позже.

ДЕКЛАРАТИВНЫЙ ПОДХОД

При декларативном подходе мы описываем результат, а не способ его достижения, а инструмент (React) будет самостоятельно следить за тем, чтобы результат соответствовал текущему состоянию:

```
1 function Greeting({username}) {  
2   return <h1>Hello, {username}!</h1>  
3 }  
4 // где-то в разметке*  
5 <Greeting username="Vasya" />
```

Вместо:

```
1 function createGreeting(parentEl, tag, props) {  
2   const el = document.createElement(tag);  
3   el.textContent = `Hello, ${props.username}!`;  
4   parentEl.appendChild(el);  
5 }
```



ИНСТРУМЕНТАРИЙ

ИНСТРУМЕНТАРИЙ

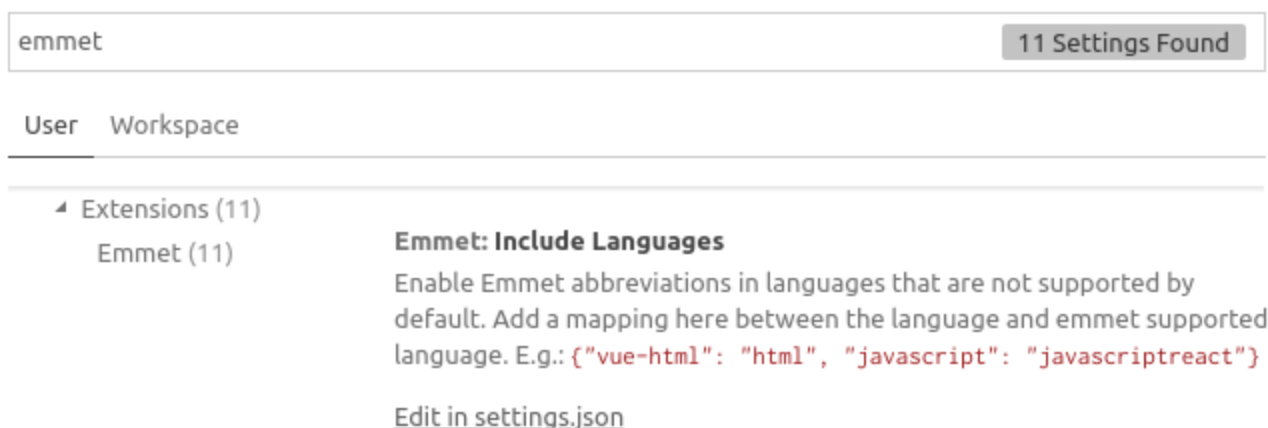
Для разработки нам понадобится следующий инструментарий (установка его описана в материалах к лекциям):

1. Node.js, npm или yarn
2. Git и GitHub
3. VSCode или VSCodium с плагином [ES7 React snippets](#)
4. Браузер Chrome/Chromium с плагинами [React DevTools](#), [Redux DevTools](#)

Есть ли у вас проблемы с установкой данных инструментов?

JSX И EMMET

Для поддержки сокращений Emmet в JSX (что это такое обсудим чуть позже), нужно в настройках VSCode включить поддержку:



В settings.json это будет выглядеть следующим образом:

```
1  "emmet.includeLanguages": {  
2    "javascript": "javascriptreact"  
3  }
```



УСТАНОВКА

УСТАНОВКА

Есть несколько способов установки React:

1. Через подключение тегов `script` с дистрибутивом библиотеки
2. Через создание полноценного проекта с использованием `npm` / `yarn`

ПОДКЛЮЧЕНИЕ СКРИПТОВ

Для подключения методом скриптов можно воспользоваться CDN:

```
1 <script
2   src="https://unpkg.com/react@16/umd/react.development.js" crossorigin>
3 </script>
4 <script
5   src="https://unpkg.com/react-dom@16/umd/react-dom.development.js" crossorigin>
6 </script>
```

После чего вам будет доступны объекты `React` и `ReactDOM`, через которые и можно получить доступ ко всей функциональности библиотеки.

Несмотря на то, что этот способ рабочий, в данном курсе он использоваться не будет и **домашние задания**, выполненные подобным образом, **приниматься не будут**.



REACT И REACTDOM

Q: Почему два скрипта два? В чём отличие?

A: React предоставляет механизмы для создания интерфейсов, ReactDOM позволяет использовать эти механизмы с DOM.

В качестве замечания стоит отметить, что есть проект React Native, который предоставляет возможность использовать React для создания мобильных приложений.

СОЗДАНИЕ ПРОЕКТА

Через `npx`:

```
npx create-react-app intro  
cd intro  
npm start
```

Через `yarn`:

```
yarn create react-app intro  
cd intro  
yarn start
```

Где `intro` - название вашего проекта.

ЗАЧЕМ НАМ `create-react-app` ?

Мы получаем уже настроенные:

- проект со всеми зависимостями и типовой структурой
- Webpack с плагинами и WebpackDevServer
- Babel, поддерживающий возможности, которых ещё нет в стандарте
- JSX (реализуется через плагин Babel)
- Jest
- ESLint с готовыми правилами
- ServiceWorker для создания Progressive Web App
- готовые скрипты запуска, сборки и тестирования

Официальная страница: <https://github.com/facebook/create-react-app>.

ТИПОВАЯ СТРУКТУРА

- `node_modules` - установленные зависимости
- `public` - публичный каталог (favicon, index.html)
- `src` - каталог для исходных кодов

INDEX.HTML

Стартовая страница нашего приложения, содержит подключаемые статичные ресурсы, `noscript` и элемент (`div#root`), в который мы будем отображать наше приложение:

```
1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="utf-8" />
5      <link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico" />
6      <meta name="viewport" content="width=device-width, initial-scale=1" />
7      <meta name="theme-color" content="#000000" />
8      <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
9      <title>React App</title>
10   </head>
11   <body>
12     <noscript>You need to enable JavaScript to run this app.</noscript>
13     <div id="root"></div>
14   </body>
15 </html>
```

INDEX.JS

Точка входа в наше приложение, в которой объект `ReactDOM` с помощью функции `render` отображает компонент `App` в элемент `#root`:

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

Обратите внимание на первый аргумент функции `render` - он очень "похож" на HTML.

ReactDOM.render

```
ReactDOM.render(element, container, [callback]);
```

1. `element` — React-элемент (важно - не DOM!)
2. `container` — узел DOM-дерева, в который будет отображаться наше приложение
3. `callback` — callback, который будет вызван после построения DOM

КОМПОНЕНТ APP

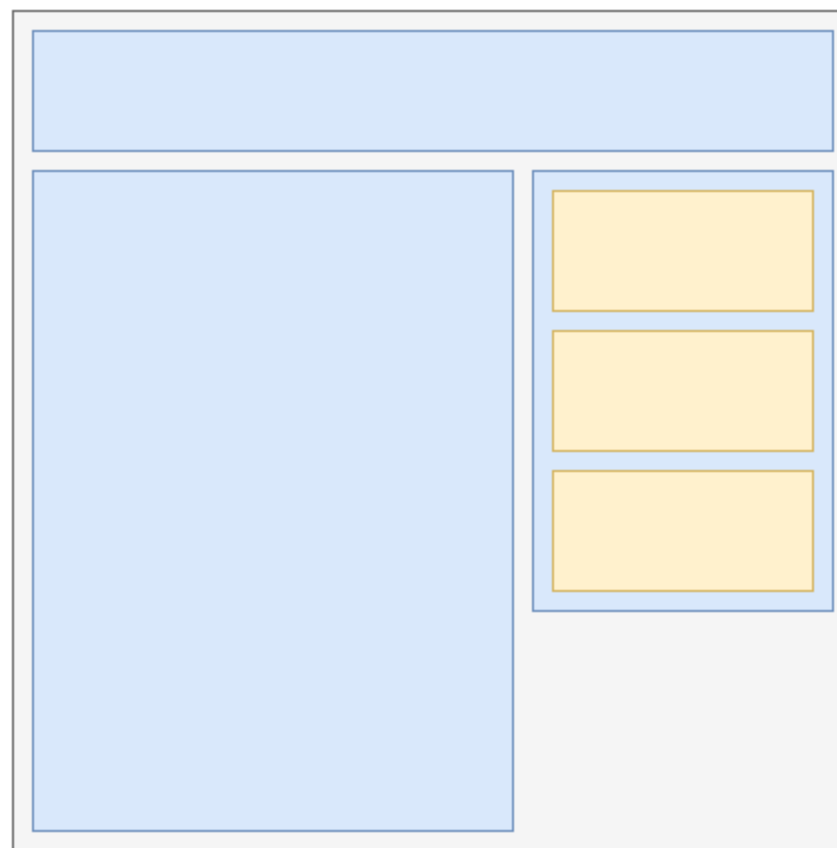
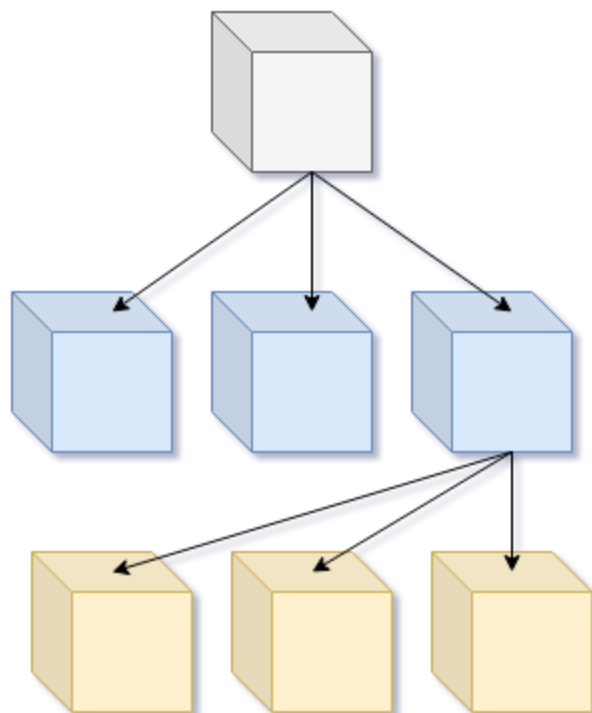
```
function App() {  
  return (  
    <div className="App">  
      <header className="App-header">  
        <img src={logo} className="App-logo" alt="logo" />  
        ... Код сокращён для удобства ...  
      </header>  
    </div>  
  );  
}  
  
export default App;
```

Т.е. компонент - это функция*, которая возвращает некую структуру (пока для простоты назовём это разметкой).



КОМПОНЕНТЫ

КОМПОНЕНТЫ





КОМПОНЕНТЫ

Вообще говоря, любой элемент интерфейса можно сделать компонентом (если у него есть своё поведение и вы собираетесь его переиспользовать).

Например, в библиотеке компонентов АльфаБанка, очень легко найти компонент, который представляет собой кнопку:

[https://design.alfabank.ru/components/button.](https://design.alfabank.ru/components/button)

ТИПЫ КОМПОНЕНТОВ

В React (по способу записи) можно выделить два типа компонентов:

1. Functional (основанные на функциях)
2. Class-based (основанные на классах)

Рассмотрим для начала Functional-компоненты, а затем посмотрим, как сделать то же самое с помощью Class-based компонентов.

Q: Почему Functional? Ведь в современные стандарты добавили классы.

A: Именно Functional компоненты являются на данный момент наиболее рекомендуемыми в сообществе React, поэтому мы в первую очередь будем рассматривать их.

ЗАДАЧА

Попробуем решить следующую задачу: отобразить профиль пользователя, информация о котором нам пришла в виде объекта:

```
{  
  name: 'Vasya',  
  status: 'React Developer',  
  online: true,  
  avatar: '/logo.svg',  
  interests: ['JavaScript', 'React', 'Frontend']  
}
```

КОМПОНЕНТ USERPROFILE

Создадим файл `src/components/UserProfile.js` - в котором разместим компонент для отображения профиля:

```
function UserProfile() {  
}  
  
export default UserProfile;
```

ИСПОЛЬЗУЕМ USERPROFILE

В файле `src/App.js`:

```
import UserProfile from './components/UserProfile'

function App() {
  const user = {
    name: 'Vasya',
    status: 'React Developer',
    online: true,
    avatar: '/logo.svg',
    interests: ['JavaScript', 'React', 'Frontend']
  }

  return <UserProfile>;
}

export default App;
```




ВОПРОСЫ

Здесь возникает несколько вопросов:

1. Что это за синтаксис такой - псевдо-HTML внутри JS?
2. Компонент-то мы создали, как туда передать данные? Ведь компонент - это переиспользуемый кусочек интерфейса.

Давайте разбираться по порядку.



JSX

JSX

JSX - синтаксическое расширение к JavaScript, которое позволяет внутри JS описывать структуры, похожие на HTML-разметку, например (в `index.js`):

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

JSX

Мы даже можем выносить эти структуры в переменные, передавать в качестве аргументов и т.д.

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

КАК ЭТО РАБОТАЕТ?

На самом деле, Babel содержит плагин [@babel/plugin-transform-react-jsx](https://babeljs.io/docs/en/babel-plugin-transform-react-jsx), который преобразует:

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

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

Пока бонусы от использования JSX не особо заметны, чуть позже мы сравним на реальном примере.

React.createElement

```
React.createElement(type, props, ...children);
```

1. `type` — тип создаваемого элемента
2. `props` — свойства элемента в формате ключ-значение
3. `children` — дочерние элементы

Знание этой конструкции даст вам понимание особенностей JSX, например, что нельзя в JSX поместить цикл, потому что помещение его в подобную цепочку вызовов приведёт к ошибке:

```
React.createElement(type, props, for (const child of children) {  
  <- так нельзя! ->  
});
```

REACT ELEMENT VS DOM ELEMENT

Обратите внимание - React Element'ы не являются DOM элементами. React использует обычные JS объекты для описания структуры, которые затем отображает с помощью React DOM уже в DOM-дерево.

Причём отображение должно быть достаточно эффективным, т.к. React выполняет сравнение различий между построенным деревом элементов и тем, что сейчас отображено на DOM и обновляет только различающиеся участки*.

Этот процесс называется Reconciliation.



VIRTUAL DOM

Virtual DOM - концепция, при которой виртуальное представление UI (состоящее из React Element'ов) хранится в памяти и синхронизируется с реальным DOM по мере необходимости.



ЗАЧЕМ НУЖЕН JSX?

JSX нужен для удобного декларативного описания интерфейсов.

JSX не является обязательным при использовании React, но является крайне рекомендуемым.

ВЕРНЁМСЯ К ЗАДАЧЕ

Вернёмся к нашей задаче и для простоты на время перенсём данные внутрь компонента `UserProfile`:

```
function UserProfile() {  
  const user = {  
    name: 'Vasya',  
    status: 'React Developer',  
    online: true,  
    avatar: '/logo.svg',  
    interests: ['JavaScript', 'React', 'Frontend']  
  }  
}  
  
export default UserProfile;
```

ОШИБКИ

В результате мы получим ошибку, которая в браузере будет выглядеть:

Failed to compile

```
./src/App.js
Line 8:  Parsing error: Unterminated JSX contents

   6 |   function App() {
   7 |     return (
>  8 |       <UserProfile>
      |                               ^
   9 |     );
  10 |   }
  11 |
```

This error occurred during the build time and cannot be dismissed.

В консоли (там где запущен npm/yarn):

JSX - НЕ HTML!

JSX следует более строгим правилам, чем HTML, поэтому теги мы обязаны закрывать, т.е. не `<UserProfile>`, а:

- `<UserProfile />`
- `<UserProfile></UserProfile>`

И самое важное - все пользовательские структуры должны быть написаны с большой буквы, чтобы React понимал, что на место этого элемента нужно создать и отобразить компонент.

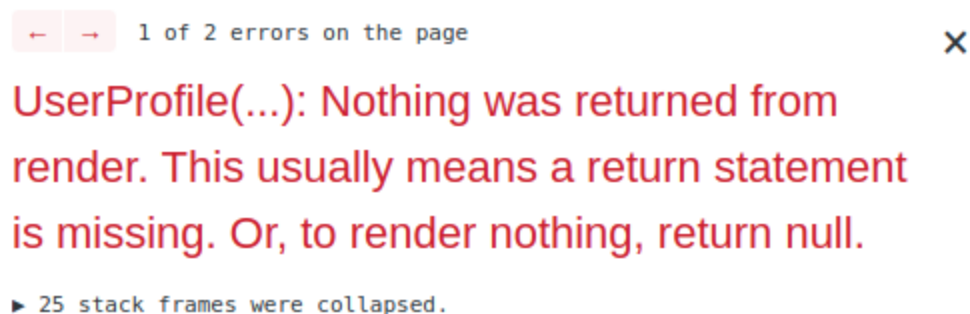


ОШИБКИ

Пожалуйста, внимательно смотрите на сообщения об ошибках: разработчики React очень постарались сделать их понятными и информативными, чтобы вы сразу нашли источник проблемы.

RENDER

Поправим и получим новую ошибку:



Компонент должен возвращать либо JSX, либо `null`.

Если возвращается JSX, то он обрабатывается согласно правилам, если `null` - то в DOM ничего не подставляется.

JSX

Определим вывод компонента `UserProfile`, для этого используем `{}` - которые позволят нам вставлять вывод выражения.

```
function UserProfile() {  
  const user = { ... };  
  return <h1>{user.name}</h1>  
}
```

Failed to compile

```
./src/components/UserProfile.js  
Line 10: 'React' must be in scope when using JSX  react/react-in-jsx-scope  
  
Search for the keywords to learn more about each error.
```

This error occurred during the build time and cannot be dismissed.

JSX автоматически преобразуется в `React.createElement`, но Babel при этом не импортирует имя `React`. Поэтому мы самостоятельно должны это делать каждый раз при использовании JSX:

```
import React from 'react'; // имр + TAB
```

JSX

Добавим вывод статуса

```
function UserProfile() {  
  const user = { ... };  
  return <h1>{user.name}</h1>  
    <p>{user.status}</p>  
}
```

Failed to compile

```
./src/components/UserProfile.js  
Line 13: Parsing error: Adjacent JSX elements must be wrapped in an enclosing tag.  
Did you want a JSX fragment <>...</>?
```

```
11 |  
12 |     return <h1>{user.name}</h1>  
> 13 |     <p>{user.status}</p>  
    |         ^  
14 |     ;  
15 | }  
16 |
```

JSX автоматически преобразуется в `React.createElement`, поэтому на верхнем уровне всегда может быть только один элемент в качестве родителя.

JSX

Кроме того, необходимы круглые скобки (если мы начинаем JSX с новой строки), чтобы многострочная конструкция и `return` воспринимались как одно выражение:

```
function UserProfile() {  
  const user = { ... };  
  return (  
    <div>  
      <h1>{user.name}</h1>  
      <p>{user.status}<p>  
    </div>  
  );  
}
```



Внутри фигурных скобок можно использовать любое валидное JS-выражение (но не циклы, if'ы и т.д.).

Например, вы легко можете вставить туда вызов функции, тернарный оператор и т.д.

Кроме того, внутри `{ }` можно использовать JSX, т.к. напомним, JSX превращается в `React.createElement(...)`.

УСЛОВНОЕ ОТОБРАЖЕНИЕ

Как уже говорилось выше, мы не можем в JSX использовать синтаксические конструкции вроде условий, циклов и т.д.

Но как же тогда выводить что-то в зависимости от условия? Например, online пользователь или offline?

Необходимо использовать тернарный оператор:

```
...  
return (  
  <div>  
    <h1>{user.name}</h1>  
    <p>{user.status}<p>  
    <p>{user.online ? <span>Сейчас на сайте</span> : null}<p>  
  </div>  
);
```

В примере специально показан сценарий, когда мы не выводим ничего, если `online = false`. Для этого просто возвращаем `null` из выражения.

УСЛОВНОЕ ОТОБРАЖЕНИЕ

Конструкцию тернарного оператора, возвращающего `null` при отрицательном результате можно заменить на `&&`:

```
...
return (
  <div>
    <h1>{user.name}</h1>
    <p>{user.status}<p>
    <p>{user.online && <span>Сейчас на сайте</span>}<p>
  </div>
);
```

В примере специально показан сценарий, когда мы не выводим ничего, если `online = false`. Для этого просто возвращаем `null` из выражения.



Вообще, `{}` достаточно универсальный инструмент, который позволяет вставлять:

- элемент (несколько элементов или массив элементов)
- текст, число
- null
- комментарии `{ /* текст комментария */ }`

Но при этом есть и ограничения, например, нельзя подставлять тег или имя атрибута:

```
<{tag}>{user.name}</{tag}>  
<h1 {attr}="value">{user.name}</h1>
```

АВАТАРКА

```
...  
return (  
  <div>  
    <h1>{user.name}</h1>  
    <p>{user.status}<p>  
    <p>{user.online ? <span>Сейчас на сайте</span> : null}<p>  
    <p><img src={user.avatar} width="50" height="50" alt="{user.name} avatar" /></p>  
  </div>  

```

Обратите внимание на вывод свойств:

- `src={user.avatar}` - подставится значение `user.avatar`
- `width="50"` - подставится 50 (не интерпретируется как выражение)
- `alt="{user.name} avatar"` - будет "{user.name} avatar" вместо "Vasya avatar"

Для свойства `alt` должно быть: `alt={user.name + ' avatar'}`

CLASSNAME

Давайте будем добавлять CSS-класс к имени, в зависимости от того, online пользователь или offline:

```
...  
return (  
  <div>  
    <h1 className={user.online ? 'online' : 'offline'}>{user.name}</h1>  
    <p>{user.status}<p>  
    <p>{user.online ? <span>Сейчас на сайте</span> : null}<p>  
    <p><img src={user.avatar} width="50" height="50" alt="..." /></p>  
  </div>  
);
```

Обратите внимание: свойство называется именно `className`, а не `class` (это зарезервированное слово в JS).

По мере того, как мы будем сталкиваться с подобными свойствами, будем запоминать их.

УВЛЕЧЕНИЯ

Осталось вывести увлечения - попробуем их вывести в виде набора тегов `<a>` (чтобы они в дальнейшем были кликабельны).

Возможности использовать циклы у нас нет, что же остаётся? Используем `map`:

```
...  
  return (  
    <div>  
      ...  
      <p>{user.interests.map(o => <a href="#">{o}</a>)}</p>  
    </div>  
  );
```

Отступы нужно будет расставить с помощью CSS, пока же вроде как всё работает, но в консоли:

Warning: Each child in a list should have a unique "key" prop.

KEY

`key` — атрибут, который помогает React определить, какой именно элемент был изменен, удален или добавлен. Нужен для увеличения производительности при отрисовке списков элементов.

Значение атрибута `key` должно быть уникальным в рамках списка (т.е. не для всей страницы)

В большинстве случаев в качестве `key` используют `id` элемента, но у нас их нет, зато сами элементы уникальны. Поэтому используем их.

Изменим код:

```
...  
return (  
  <div>  
    ...  
    <p>{user.interests.map(o => <a href="#" key={o}>{o}</a>)}</p>  
  </div>  
)
```

ПОЛНЫЙ КОД

```
function UserProfile() {
  const user = {
    name: 'Vasya',
    status: 'React Developer',
    online: true,
    avatar: '/img/logo.svg',
    interests: ['JavaScript', 'React', 'Frontend']
  };

  return (
    <div>
      <h1>{user.name}</h1>
      <p>{user.status}</p>
      <p>{user.online ? <span>Сейчас на сайте</span> : null}</p>
      <p><img src={user.avatar} width="50" height="50" alt={user.name + 'avatar'} /></p>
      <p>{user.interests.map(o => <a href="#" key={o}>{o}</a>)}</p>
    </div>
  );
}
```

АЛЬТЕРНАТИВА БЕЗ JSX

```
function UserProfile() {  
  ...  
  
  return React.createElement('div', null,  
    React.createElement('h1', null, user.name),  
    React.createElement('p', null, user.status),  
    React.createElement('p', null,  
      user.online ? React.createElement('span', null, 'Сейчас на сайте') : null  
    ),  
    React.createElement('img', {  
      src: user.avatar, width: '50', height: '50', alt: user.name + ' avatar'  
    }),  
    React.createElement('p', null, user.interests.map(  
      o => React.createElement('a', {href: '#', key: o}, o)  
    )),  
  );  
}
```

Читабельность упала в разы. Поэтому наш выбор - JSX.

STYLE

Несмотря на то, что стили лучше инкапсулировать в CSS-классы, посмотрим, как работать с inline-стилями в React.

Для этого нужно использовать свойство `style`, передавая в него объект, содержащий сами стили, при этом названия свойств пишутся в camelCase:

```
const styles = {
  backgroundColor: 'black',
  borderRadius: '5px',
  padding: '4px',
  color: '#fff',
  display: 'inline-block',
  marginRight: '5px',
  textDecoration: 'none',
};

return (
  <div>
    <p>{user.interests.map(
      o => <a href="#" key={o} style={styles}>{o}</a>
    )}</p>
  </div>
);
```

JSX: КЛЮЧЕВЫЕ МОМЕНТЫ

1. JSX - не HTML, все теги нужно закрывать
2. При использовании JSX обязательно импортировать `React`
3. JSX на верхнем уровне может содержать только один элемент
4. Нельзя использовать `if`'ы, циклы, можно - тернарный оператор и `map`
5. Для css-классов свойство `className`
6. Для inline-стилей свойство `style` с camelCase именами свойств



PROPS

PROPS

Итак, с отображением данных мы более-менее разобрались. Давайте посмотрим на то, как передавать данные в компонент.

Документация React говорит, что компоненты можно представлять как JS-функции: на вход получаем параметры (которые называются `props`), на выходе получаем представление компонента (в нашем случае в виде JSX).

ПЕРЕДАЧА PROPS

```
import UserProfile from './components/UserProfile'

function App() {
  const user = {
    name: 'Vasya',
    status: 'React Developer',
    online: true,
    avatar: '/logo.svg',
    interests: ['JavaScript', 'React', 'Frontend']
  }

  return <UserProfile user={user}>;
}

export default App;
```


ПОЛУЧЕНИЕ PROPS

```
function UserProfile(props) {  
  const {user} = props; // object destructuring  
  
  return (  
    <div>  
      <h1>{user.name}</h1>  
      <p>{user.status}</p>  
      <p>{user.online ? <span>Сейчас на сайте</span> : null}</p>  
      <p><img src={user.avatar} width="50" height="50" alt={user.name + 'avatar'}" /></p>  
      <p>{user.interests.map(o => <a href="#" key={o}>{o}</a>)}</p>  
    </div>  
  );  
}
```

PROPS

На самом деле любые свойства, передаваемые в JSX (или `React.createElement`), попадают в `props`. И передавать туда можно что угодно: примитивы, объекты, массивы, функции и т.д. Например, можно по отдельности передавать все поля:

```
function App() {  
  const user = {  
    ...  
  }  
  
  return <UserProfile name={user.name} status={user.status}>;  
}
```

```
function UserProfile(props) {  
  const {name, status} = props;  
  ...  
}
```

PROPS - READONLY

Самое важное замечание, которое следует сделать про `props` - это readonly аргументы, не нужно их модифицировать!



PROP-TYPES

PROP-TYPES

Поскольку JS - очень динамичный язык, в больших проектах используют инструменты позволяющие отслеживать типы передаваемых объектов. Библиотека [prop-types](#) позволяет описать типы `props`, для того, чтобы редакторы кода, а там же сам React проверял типы передаваемых аргументов.

Через `npm`:

```
npm install prop-types
```

Через `yarn`:

```
yarn add prop-types
```

MODEL

Вынесем описание объекта `UserModel` в отдельный класс (`src/models/UserModel.js`):

```
class UserModel {  
  constructor(id, name, status, online, avatar, interests) {  
    this.id = id;  
    this.name = name;  
    this.status = status;  
    this.online = online;  
    this.avatar = avatar;  
    this.interests = interests;  
  }  
}  
  
export default UserModel;
```

PROP-TYPES

```
import React from 'react';
import PropTypes from 'prop-types'; // impt + TAB
import UserModel from '../models/UserModel';

function UserProfile(props) {
  const { user } = props;
  return (
    <div>
      <h1>{user.name}</h1>
      <p>{user.status}</p>
      <p>{user.online ? <span>Сейчас на сайте</span> : null}</p>
      <p><img src={user.avatar} width="50" height="50" /></p>
      <p>{user.interests.map(o => <a href="#" key={o}>{o}</a>)}</p>
    </div>
  );
}

UserProfile.propTypes = {
  user: PropTypes.instanceOf(UserModel).isRequired
}

export default UserProfile;
```

PROP-TYPES

Теперь, при передаче объекта другого типа мы будем получать соответствующее сообщение в консоли:

Warning: Failed prop type: Invalid prop `user` of type `Object` supplied to `UserProfile`, expected instance of `UserModel`.

Мы ещё будем детально рассматривать пакет prop-types, пока же вы можете ознакомиться с документацией на [странице документации](#).

CLASS-BASED КОМПОНЕНТЫ

CLASS-BASED КОМПОНЕНТЫ

Class-based компоненты строятся на основе наследования от `React.Component` и определения метода `render`:

```
class UserProfile extends React.Component {
  // конструктор приведён лишь для полноты картины
  // в текущем виде он не является обязательным
  constructor(props) {
    super(props);
  }

  render() {
    const { user } = this.props;
    return (<div> ... </div>);
  }
}

UserProfile.propTypes = {
  user: PropTypes.instanceOf(UserModel).isRequired
}

export default UserProfile;
```



CLASS-BASED КОМПОНЕНТЫ

Это всё, что нужно было изменить, использование данного компонента в JSX ничем не будет отличаться от использования Functional-компонента.

CLASS-BASED И PROP-TYPES

prop-types для Class-based компонентов могут быть определены также с использованием синтаксиса, который будет реализован в будущих версиях ES:

```
class UserProfile extends React.Component {  
  static propTypes = {  
    user: PropTypes.instanceOf(UserModel).isRequired  
  }  
  
  render() {  
    const { user } = this.props;  
    return (<div> ... </div>);  
  }  
}  
  
export default UserProfile;
```



ИТОГИ



КЛЮЧЕВЫЕ ПОДХОДЫ

1. Декларативность и JSX
2. Компонентно-ориентированность

Важно: компонент - основной строительный блок React-приложения. Собирая иерархию компонентов (т.е. организовывая композицию) мы можем создавать сколь угодно сложные приложения.

Компоненты создаются для переиспользования и их можно использовать в качестве тегов JSX.

ES7 REACT SNIPPETS

Плагин ES7 React Snippets предлагает удобные сокращения для создания функциональных и class-based компонентов `rfcp + TAB` и `rcsp + TAB` соответственно.



Задавайте вопросы и напишите отзыв о лекции!

АНТОН СТЕПАНОВ



[@anton_mesmer](https://www.instagram.com/anton_mesmer)