



React.js Course

Занятие 1. Введение и рендеринг

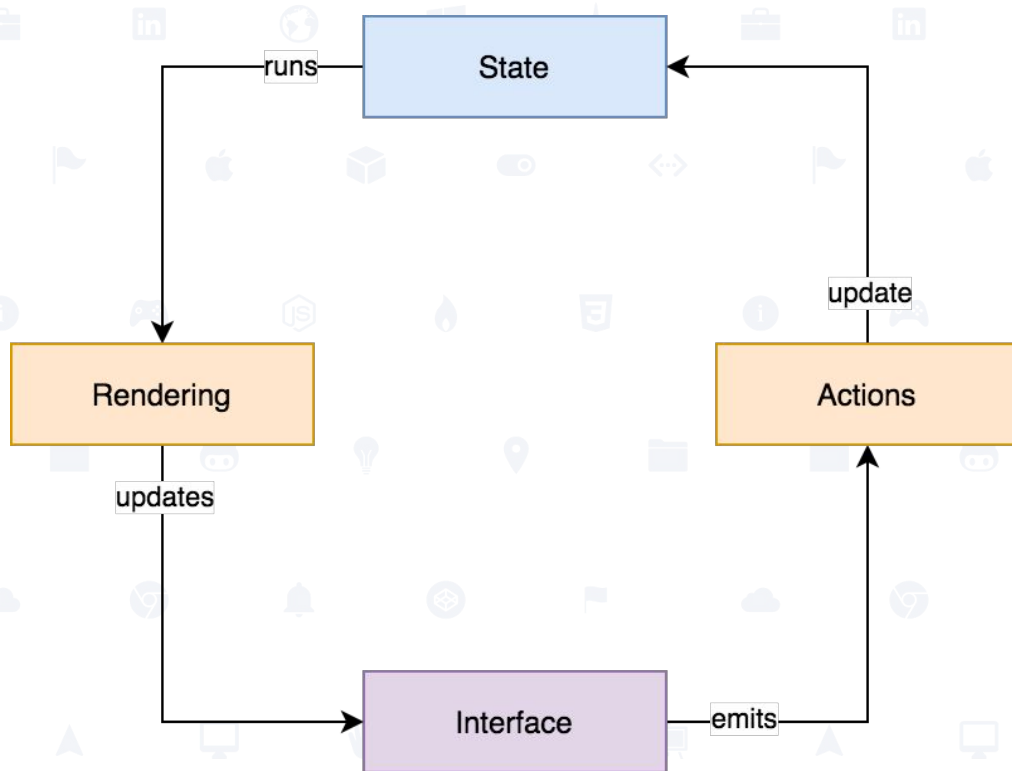
Автор программы и преподаватель - [Помазкин Илья](#)



Структура занятия

1. Концепция интерактивности
2. Развития frontend-разработки в web
3. Что такое React.js и React-stack и какие задачи они решают
4. Setup проекта и окружения
5. Структура приложения
6. Рендеринг и JSX

Концепция интерактивности



1. Пользователь видит интерфейс и делает в нем какие-то действия
2. Действия обновляют состояние
3. Изменение состояния запускает рендер
4. Рендер обновляет интерфейс



Примеры интерактивности

1. Меню, которое прячется “в бургер”
2. URL сайта и рендер страницы
3. Чекбокс
4. Радиокнопки
5. Текстовые поля
6. Ползунок громкости
7. Timeline с прогрессом проигрывания
8. Выпадающий список
9. Слайдер
10. Что угодно

Развитие frontend-разработки в web

1991 год - первый веб-сайт. Интерактивность на уровне URL - какую страницу запросил, такую и получил.

1993 год - опубликован стандарт разметки HTML 1. Интерактивность через формы и поля ввода.

1996 год - публикация JavaScript 1.0. Интерактивность через взаимодействие с DOM и BOM.

1996 год - принят стандарт CSS 1. Добавилась интерактивность через псевдоклассы: hover, focus, visited

1996-1998 годы - первое использование AJAX. Интерактивность через динамически загружаемый контент - iframe, img, script

2000 год - появление XMLHttpRequest. Интерактивность через асинхронные HTTP-запросы.

2002-2003 годы - появление концепции SPA. Первый сайт - slashdotslash.com.

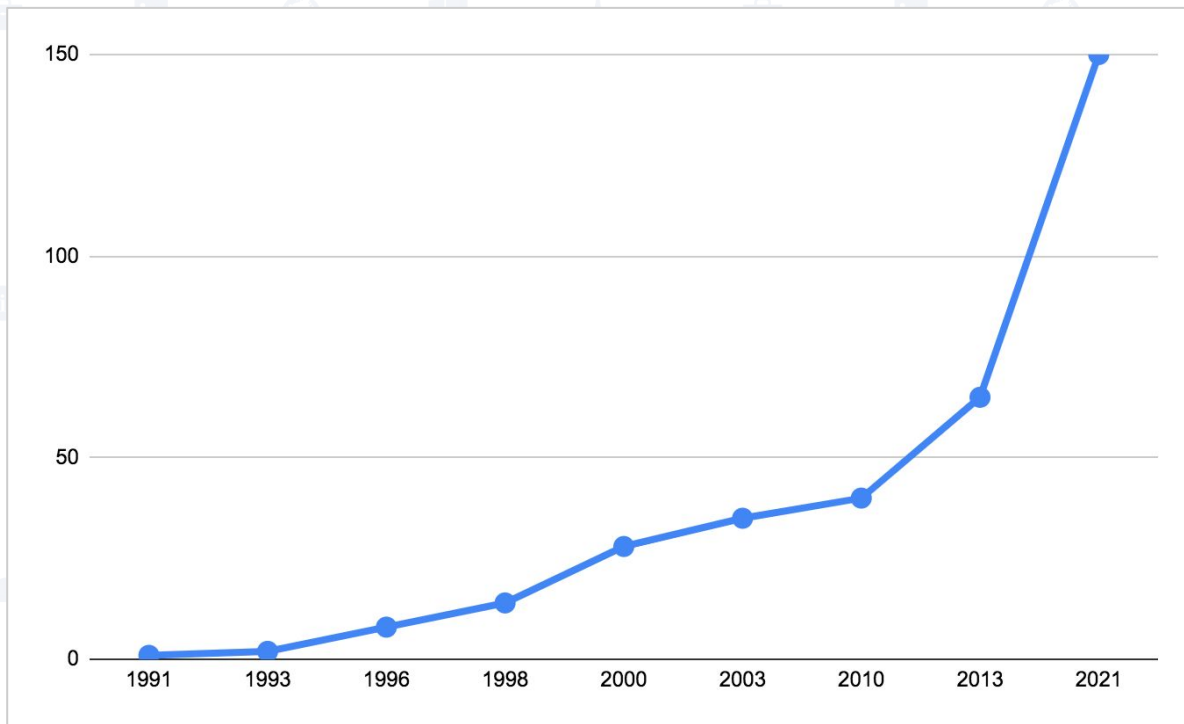
2004-2012 годы - появление первых фреймворков для реализации SPA. (Ext JS, AngularJS, Knockout, Ember)

2010 год - первая реализация WebSocket

2013-2014 годы - появление React, Vue

2016 год - появление Angular

Увеличение сложности приложений

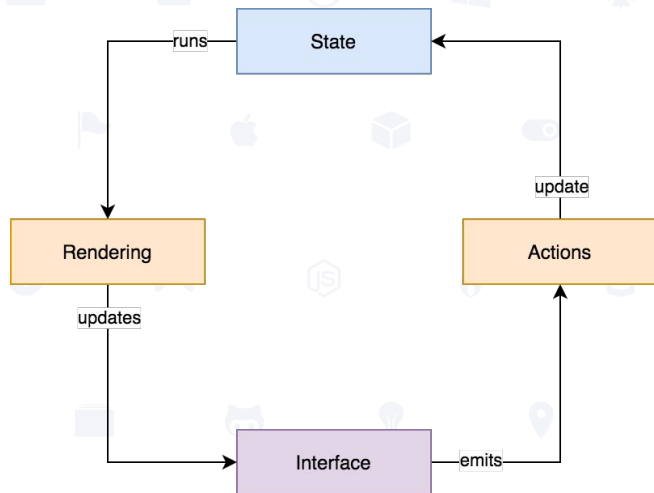


Со временем приложения стали сложнее: больше интерактивности, больше данных, выше скорость обновления.

Создавать и поддерживать такие приложения стало трудно. Для удобства, стандартизации, упрощения и ускорения стали создаваться фреймворки и библиотеки, такие как React, Vue, Angular и прочие.

** Значение сложности выставлено примерно, для наглядности. За основу взято примерное к-во доступных технологий и к-во разных типов интерактивности.*

Что такое React.js



React решает задачи на всех 4 этапах и делает это быстро, эффективно и удобно

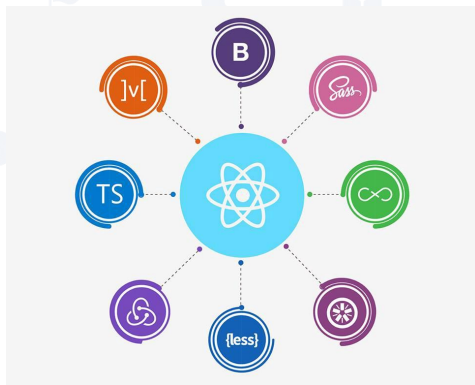
React.js - это библиотека, которая решает такие задачи:

- рендеринг HTML;
- обработка и добавление событий к DOM-элементам;
- хранение и управление состоянием;
- автоматический запуск рендера при обновлении состояния;
- переиспользование кода через использование компонентов

Проще говоря - делает разработку проекта удобной, а frontend-разработчика - счастливым :)



Что включает React-stack



Сам по себе React.js не решает все задачи, которые встречаются в SPA. Для них используются сторонние библиотеки. Вот несколько примеров:

- react-router - для маршрутизации в браузере
- redux & react-redux - для управления глобальным состоянием
- axios, superagent - для AJAX-запросов
- formik, react-hook-forms - для работы с формами, валидации и прочего
- react-transition-group - для контролируемых анимаций
- ... - что угодно: <https://www.npmjs.com/search?q=react>

Setup проекта и окружения

Обычно для сборки приложения используется [webpack](#). Для этого нужно написать конфиг, установить зависимости, настроить package.json.

Также нужно создать базовую структуру файлов и добавить в зависимости все необходимые библиотеки.

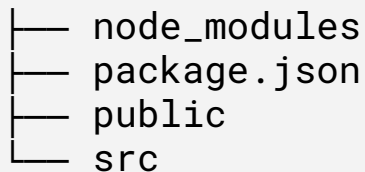
Для упрощения всего этого процесса есть [create-react-app](#). Это утилита командной строки, которая одной командой создает все необходимые файлы и устанавливает зависимости. Также она позволяет использовать webpack без его ручной настройки.

Для создания папки с приложением просто запустите эту команду:

```
npx create-react-app my-app
```

В текущей папке будет создана папка “my-app” с файлами проекта.

Структура файлов проекта



```
├── node_modules
├── package.json
├── public
└── src
```

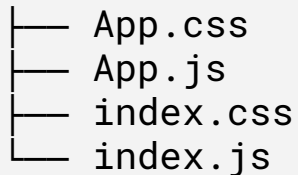
`node_modules` - тут хранятся зависимости проекта, добавленные `npm` или `yarn`

`package.json` - файл с описанием `npm`-пакета вашего приложения. Тут - список зависимостей, команды для автоматизации, конфигури определенных плагинов и библиотек

`public` - это папка, где лежат статичные файлы проекта и `index.html`

`src` - папка с исходным кодом проекта

Структура файлов проекта - src/



- App.css
- App.js
- index.css
- index.js

index.js - это корневой файл приложения, с него начинается сборка всего приложения

App.js - это файл корневого компонента приложения

index.css и App.css - файлы стилей

Структура проекта - src/index.js

```
import React from 'react';           // импорт основной части React
import ReactDOM from 'react-dom';    // импорт библиотеки, отвечающей за рендеринг в DOM

import './index.css';                // импорт файла стилей
import App from './App';             // импорт корневого компонента

// рендер компонента в DOM-элемент на странице
ReactDOM.render(
  <App/>,                             // передаем компонент как JSX-элемент
  document.getElementById('root')    // находим нужный DOM-элемент по ID
);
```

Структура проекта - src/App.js

```
import logo from './logo.svg'; // импорт SVG-файла с логотипом
import './App.css';             // импорт стилей

function App() {                 // объявление функционального компонента
  return (                       // рендерим контент компонента
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>Edit <code>src/App.js</code> and save to reload.</p>
        <a className="App-link" href="https://reactjs.org" target="_blank">
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App; // экспорт компонента через экспорт по умолчанию
```

Рендеринг и JSX

В файле index.js такой код:

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

Для рендера в DOM используется функция ReactDOM.render. Она имеет такое описание:

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

Первым параметром функции идет element. Это React-элемент, объект, который описывает React что и как нужно рендерить.

Второй параметр - container. Это DOM-элемент, внутри которого нужно рендерить переданный element.

Третий параметр необязателен - это callback, который будет вызываться при каждом рендере.

В index.js мы указываем, что нужно отрендерить React-элемент “App” в DOM-элементе, который мы выбрали по ID.

Что такое React-элемент

React-элемент - это объект, который описывает что должен рендерить React.js.

Для создания элементов используется функция `React.createElement`. Она имеет вот такое описание:

```
React.createElement(type, [props], [...children])
```

Первый параметр - `type`. Это тип элемента, который мы хотим отрендерить. Типом может быть: строка, содержащая имя тега (например, `'div'`), React-компонент (класс или функция) или React-фрагмент.

Второй параметр, необязательный - `props`. “Пропсы” - это входящие данные, которые получит элемент.

Третий и далее параметры, необязательные - `children`. Это элементы, которые мы хотим отрендерить внутри создаваемого элемента.

`React.createElement` создаст и вернет React-элемент определенного типа. И дальше этот элемент мы передадим либо в `ReactDOM.render` либо как `children` при создании другого React-элемента.

Композиция React-элементов

React.createElement может принимать дочерние элементы, для рендера их внутри создаваемого элемента. Так мы можем вкладывать одни элементы в другие и строить наше приложение по кирпичикам. Это очень похоже на то, как работают HTML-теги.

Как бы выглядел HTML:

```
<div class="hero">
  <h1>Hello!</h1>
  <p>Lorem ipsum.</p>
</div>
```

Но описывать разметку с помощью React.createElement не очень удобно - код неочевидный и однотипный.

Что бы решить эту проблему был создан JSX.

Реализация с помощью React.createElement:

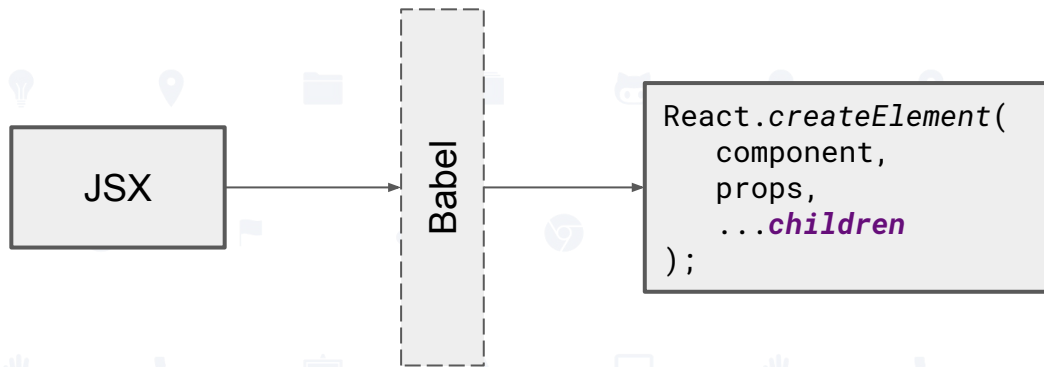
```
React.createElement(
  "div",
  { className: "hero" },
  React.createElement(
    "h1",
    null,
    "Hello!"
  ),
  React.createElement(
    "p",
    null,
    "Lorem ipsum."
  )
);
```


Что такое JSX

JSX - это синтаксический сахар для функции `React.createElement`. Его можно расценивать как шаблонизатор, в котором доступен обычный JS.

Он был создан для удобства работы разработчика - намного проще работать с чем-то, похожим на HTML-теги, чем с обычной JS-функцией.

В итоге каждый JSX-тег с помощью Babel превращается в вызов функции `React.createElement`.



Примеры трансформаций JSX

Посмотреть, как будет трансформироваться JSX можно вот в [этой онлайн-песочнице](#). Ниже - несколько примеров.

```
<div>Hello!</div>;
```

```
React.createElement("div", null, "Hello!");
```

```
<div className="text">Hello!</div>;
```

```
React.createElement("div", {  
  className: "text"  
}, "Hello!");
```

```
<div className="text">  
  <p>Here is a text</p>  
</div>
```

```
React.createElement(  
  "div",  
  { className: "text" },  
  React.createElement(  
    "p",  
    null,  
    "Here is a text"  
  )  
)
```

Возвращаясь к src/index.js

Теперь нам понятно, что происходит в этом куске кода:

```
ReactDOM.render(  
  <App/>, // передаем React-элемент с типом "App" для рендера приложения  
  document.getElementById('root')  
);
```

После трансформации JSX в JS это будет выглядеть вот так:

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



Рендеринг

React.createElement первым параметром принимает тип элемента. Типом может быть: строка, содержащая имя тега (например, 'div'), React-компонент (класс или функция) или React-фрагмент.

Тег рендерится сразу в DOM-element.

React-компонент рендерится:

- вызовом функции для функциональных компонентов
- вызовом метода render для классовых компонентов.

В обоих случаях функция должна вернуть React-элемент.

React-фрагмент просто рендерит дочерние элементы.

Рендеринг DOM-элементов

Что бы отрендерить DOM-элемент, используйте тег этого элемента, написанный с маленькой буквы. Вот так:

```
<h1>Hello!</h1>
```

Дочерние элементы указывайте между открывающим и закрывающим тегом, как в обычном HTML:

```
<h1>  
  <span>Hello </span><span>World!</span>  
</h1>
```

Дочерние элементы можно передавать как:

- строки и числа; числа будут приведены к строковому типу. При этом:
 - *JSX удаляет пустые строки и пробелы в начале и конце строки. Новые строки, примыкающие к тегу будут удалены. Новые строки между строковых литералов сжимаются в один пробел.*
- JSX-компоненты
- JS-выражения, обернутые в {}. Результат выражения будет приведен к строке.
- Boolean, null, undefined - React просто проигнорирует их

Если дочерних элементов нет, можно использовать самозакрывающийся тег:

```
<h1 />
```

Рендеринг DOM-элементов - атрибуты и пропсы

Задать атрибуты DOM-элемента можно, передав в компонент соответствующие пропсы. Вот так:

```
<h1 id="heading"></h1>
```

При этом все свойства и атрибуты DOM (включая обработчики событий) должны быть в стиле camelCase. Исключения: aria-* data-*.

Как бы это выглядело в HTML:

```
<h1
  class="heading"
  tabindex="0"
  data-custom=""
  aria-label="Heading"
  id="heading">Heading</h1>
```

Как это выглядит в JSX:

```
<h1
  className="heading" // camel case
  tabIndex="0"       // camel case
  data-custom=""      // как обычно
  aria-label="Heading" // как обычно
  id="heading">Heading</h1>
```

Подробнее в официальной документации: [ссылка](#).

Рендеринг DOM-элементов - атрибуты и пропсы

В пропсы можно передавать:

- строки и числа; числа будут приведены к строковому типу
- JS-выражения, обернутые в {}. Результат выражения будет приведен к строке.
- Boolean, null, undefined - React просто проигнорирует их
- ничего - тогда значение автоматически выставится в true

```
<h1 id="test">Hello</h1>           // это выражение
<h1 id={'test'}>Hello</h1>         // и это - работают одинаково
<input type="checkbox" checked />     // checked автоматически установится в true
```

Пропсы можно передавать с помощью spread operator:

```
const props = {
  className: "test",
  id: "heading",
};

<h1 {...props}>Hello!</h1>
```



Рендеринг React-компонентов

Что бы отрендерить React-компонент, используйте имя функции или класса, написанное с большой буквы. Вот так:

```
<App></App>
```

Дочерние элементы передаются так же как и для DOM-элементов:

```
<App>  
  <h1>Hello!</h1>  
</App>
```

В компоненты можно передавать любые пропсы:

```
<App some="hello" data={[1, 2, 3]} options={{ test: 1 }}>  
  <h1>Hello!</h1>  
</App>
```

Дочерние элементы запишутся в пропс "children", остальные пропсы передадутся как есть.

Краткое введение в React-компоненты

React-компоненты есть двух типов: классовые и функциональные.

Классовый компонент:

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

export class Button extends Component {
  render() {
    const { children, ...rest } = this.props;

    return (
      <button {...rest}>{children}</button>
    );
  }
}
```

Класс должен наследовать от `React.Component`.
За рендер отвечает метод `render` - он должен вернуть `React-элемент`.

Пропсы записываются в свойство объекта `"props"`.
К ним можно обратиться через `this.props` в методе `render`.

Дочерние элементы доступны в `props.children`.

Функциональный компонент:

```
import React from "react";

export function Button({ children, ...rest }) {
  return (
    <button {...rest}>{children}</button>
  );
}
```

За рендер отвечает сама функция - она должна вернуть `React-элемент`.

Пропсы передаются как первый аргумент функции. К ним можно обращаться напрямую.

Дочерние элементы доступны в `props.children`.

Пропсы можно только читать

Рендеринг списков

Что бы отрендерить список элементов, используйте массивы. Например вот так:

```
<h1>{[1, 2, 3]}</h1>
```

Массив можно создавать прямо во время рендера, методом `Array.map`:

```
const list = [
  { id: 0, title: 'Banana', },
  { id: 1, title: 'Apple', },
  { id: 2, title: 'Orange', },
];

<ul>
  {list.map(el => (
    <li key={el.id}>{el.title}</li>
  ))}
</ul>
```

При рендере списков React должен как-то отличать разные элементы. Для этого используется проп “key”. Он должен быть уникальным в этом конкретном списке. Если key не указать React использует индекс элемента в списке и выдаст ошибку в консоли.

Если не указать key, могут пострадать производительность и появится баги при рендере. Например, когда поменяется порядок элементов в списке.

Рендеринг с условием

Можно рендерить элементы и атрибуты в зависимости от определенных условий. Для этого есть 2 пути - обычные условия в функции отвечающей за рендер и JS-выражения прямо в JSX.

Условия в JS:

```
function Button({ children, isLink = false, ...rest }) {  
  const text = <span>{children}</span>;  
  if (isLink) return <a {...rest}>{text}</a>;  
  return <button {...rest}>{text}</button>;  
}
```

Условия в JSX. Оператор &&:

```
function Button({ children, isShowIcon = false, ...rest }) {  
  return (  
    <button {...rest}>  
      {children}  
      // false && <img /> вернет false и React ничего не отрендерит  
      // true && <img /> вернет <img /> и React отрендерит изображение  
      {isShowIcon && }  
    </button>  
  );  
}
```

Рендеринг с условием

Условия в JSX. Тернарный оператор:

```
function Button({ children, type = "text", text = null, icon = null, ...rest }) {  
  return (  
    <button {...rest}>  
      {type === 'text' ? text : <img src={icon} alt="" />} // обычный тернарный оператор  
    </button>  
  );  
}
```

Рендеринг нескольких элементов без обертки

Что бы отрендерить несколько элементов, не используя оборачивающий DOM-элемент, используйте `React.Fragment`. Например при рендере ячеек таблицы:

```
import React, { Fragment } from "react";
```

```
<Fragment>  
  <td>Hello!</td>  
  <td>Lorem ipsum.</td>  
</Fragment>
```

Можно использовать сокращенный синтаксис:

```
<>  
  <td>Hello!</td>  
  <td>Lorem ipsum.</td>  
</>
```

Рендеринг готовой HTML-строки

Бывают случаи, когда необходимо отрендерить готовую HTML-строку. Например, если пришли данные с сервера, и вам нужно отрендерить контент как есть. Если вы передадите такую строку в проп children - React просто отрендерит ее как строку.

Ваш код:

```
const someHTML = `

Here is some  
<span>HTML</span></p>`;  
  
return (  
  <div>{someHTML}</div>  
);


```

Что отрендерит React:

```
<div><p>Here is some <span>HTML</span></p></div>
```

Это потенциально опасная операция, поэтому React перестраховывается и рендерит HTML-строку как строку. Что бы исправить это используйте проп "dangerouslySetInnerHTML"

Ваш код:

```
const someHTML = `

Here is some <span>HTML</span></p>`;  
  
return (  
  <div dangerouslySetInnerHTML={{ __html: someHTML }}/>  
);


```

Что отрендерит React:

```
<div>  
  <p>  
    "Here is some "  
    <span>HTML</span>  
  </p>  
</div>
```

Рендеринг за пределами корневого DOM-элемента

Бывают случаи, когда необходимо отрендерить элемент за пределами корневого DOM-элемента. Например при рендере модальных окон. Для этого используются порталы:

```
import React from "react";
import { createPortal } from 'react-dom';

// в функции рендера
return createPortal((
  <h1>Hello!</h1>
), document.getElementById('modals'));
```

Первым параметром функции createPortal идет element. Это React-элемент.

Второй параметр - container. Это DOM-элемент, внутри которого нужно рендерить переданный element.

