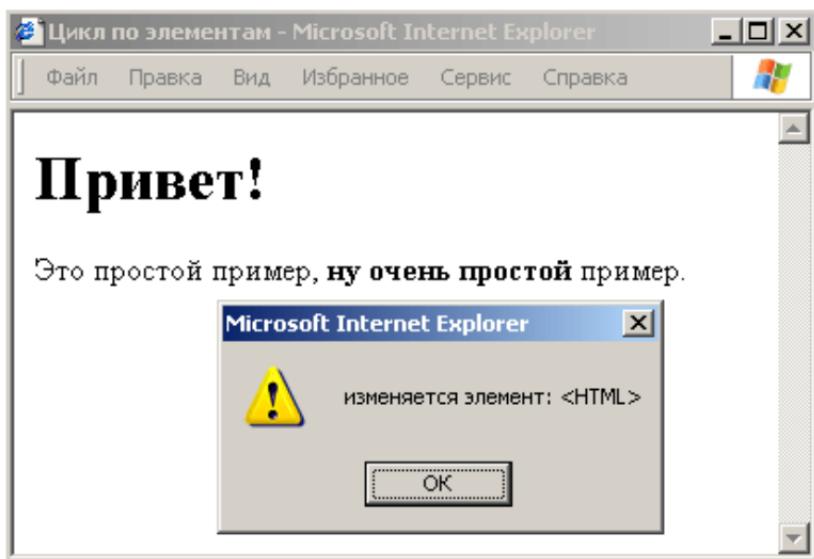


React

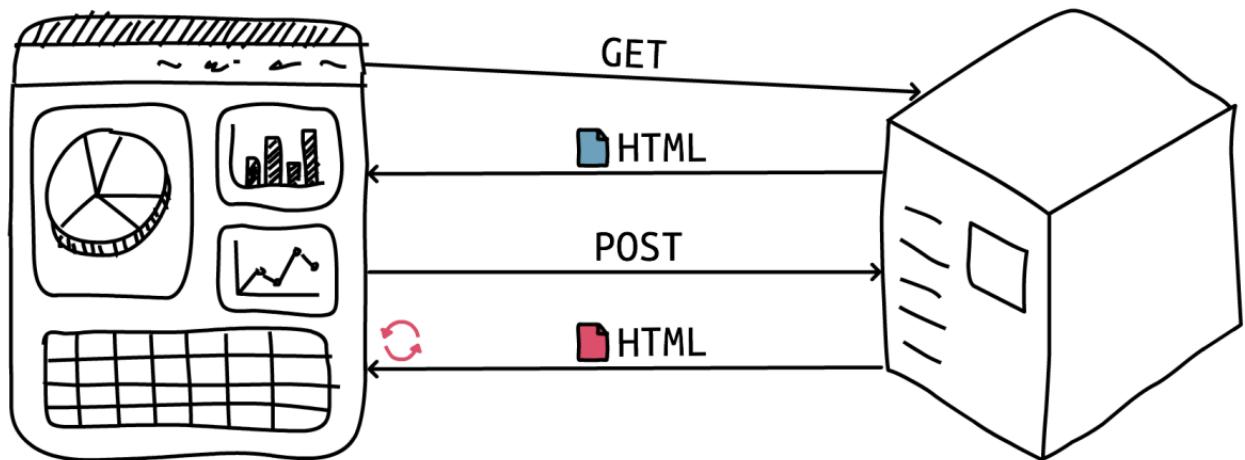
Немного истории

Эволюция задач перед JS

Простые задачи



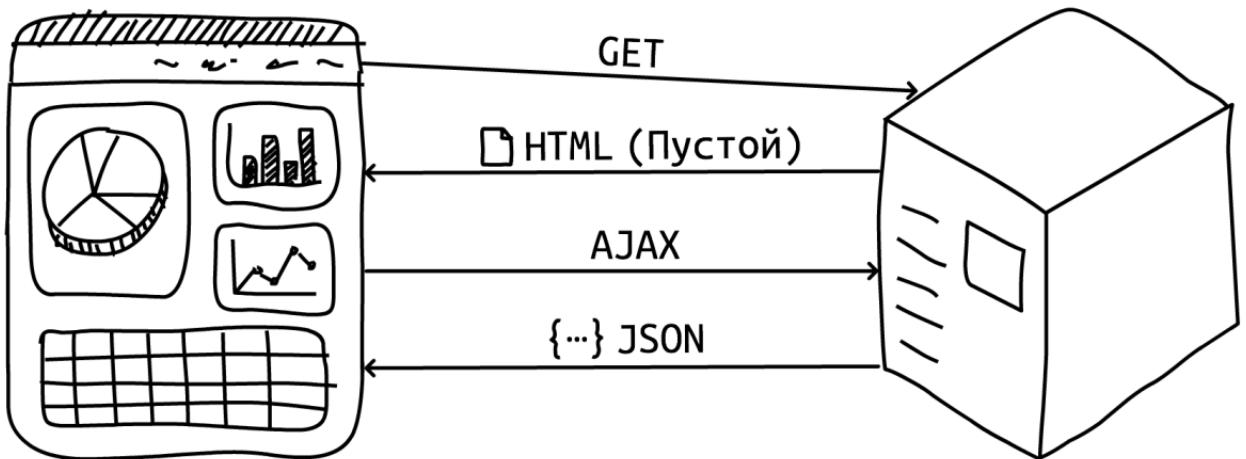
Любые изменения на странице чаще всего вызывали полную перезагрузку страницы

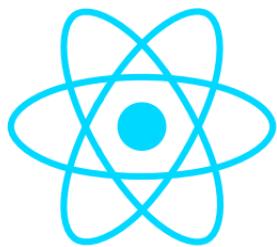


Single Page Application^(SPA)

Веб-приложение или сайт, который загружает только одну страницу и все последующие запросы обрабатываются без полной перезагрузки страницы

В SPA сервер отвечает только данными, а все изменения
верстки происходят из клиентского JavaScript





React

React — JavaScript-библиотека для создания пользовательских интерфейсов, которая позволяет довольно просто реализовывать и SPA в том числе

Концепции React

Компонентный подход

Эффективная абстракция над DOM

Реактивный рендеринг

Как пользоваться?

НОВАЯ ЗАМЕТКА

Название заметки

Текст заметки

Осталось символов: 100

Отменить Сохранить

МОИ ЗАМЕТКИ

 Books
Books to read

 Music
Music to listen

 Films
Films to watch

Как пользоваться?

```
// Импортируем необходимые библиотеки
import React from 'react';
import ReactDOM from 'react-dom';

// Выбираем элемент, внутри которого мы хотим нарисовать форму
const root = document.getElementById('root');

// Создаем элемент формы
const form = React.createElement(
  'div',
  { className: 'editor' },
  // Элемент состоит из трех частей: имя тега, атрибуты, дочерние элементы
  // HTML эквивалент: <input placeholder="Ключ заметки" />
  React.createElement('input', { placeholder: 'Ключ заметки' }),
  React.createElement('textarea', { placeholder: 'Текст заметки' }),
  React.createElement('button', null, 'Отменить'),
  // HTML эквивалент: <button>Сохранить</button>
  React.createElement('button', null, 'Сохранить')
);

// Отображаем форму на экране
ReactDOM.render(form, root);
```

Как пользоваться?

```
// Импортируем необходимые библиотеки
import React from 'react';
import ReactDOM from 'react-dom';

// Выбираем элемент, внутри которого мы хотим нарисовать форму
const root = document.getElementById('root');

// Создаем элемент формы
const form = React.createElement(
  'div',
  { className: 'editor' },
  // Элемент состоит из трех частей: имя тега, атрибуты, дочерние элементы
  // HTML эквивалент: <input placeholder="Ключ заметки" />
  React.createElement('input', { placeholder: 'Ключ заметки' }),
  React.createElement('textarea', { placeholder: 'Текст заметки' }),
  React.createElement('button', null, 'Отменить'),
  // HTML эквивалент: <button>Сохранить</button>
  React.createElement('button', null, 'Сохранить')
);

// Отображаем форму на экране
ReactDOM.render(form, root);
```

Как пользоваться?

```
// Импортируем необходимые библиотеки
import React from 'react';
import ReactDOM from 'react-dom';

// Выбираем элемент, внутри которого мы хотим нарисовать форму
const root = document.getElementById('root');

// Создаем элемент формы
const form = React.createElement(
  'div',
  { className: 'editor' },
  // Элемент состоит из трех частей: имя тега, атрибуты, дочерние элементы
  // HTML эквивалент: <input placeholder="Ключ заметки" />
  React.createElement('input', { placeholder: 'Ключ заметки' }),
  React.createElement('textarea', { placeholder: 'Текст заметки' }),
  React.createElement('button', null, 'Отменить'),
  // HTML эквивалент: <button>Сохранить</button>
  React.createElement('button', null, 'Сохранить')
);

// Отображаем форму на экране
ReactDOM.render(form, root);
```

Как пользоваться?

```
// Импортируем необходимые библиотеки
import React from 'react';
import ReactDOM from 'react-dom';

// Выбираем элемент, внутри которого мы хотим нарисовать форму
const root = document.getElementById('root');

// Создаем элемент формы целиком
const form = React.createElement(
  'div',
  { className: 'editor' },
  // Элемент состоит из трех частей: имя тега, атрибуты, дочерние элементы
  // HTML эквивалент: <input placeholder="Ключ заметки" />
  React.createElement('input', { placeholder: 'Ключ заметки' }),
  React.createElement('textarea', { placeholder: 'Текст заметки' }),
  React.createElement('button', null, 'Отменить'),
  // HTML эквивалент: <button>Сохранить</button>
  React.createElement('button', null, 'Сохранить')
);

// Отображаем форму на экране
ReactDOM.render(form, root);
```

Как пользоваться?

```
// Импортируем необходимые библиотеки
import React from 'react';
import ReactDOM from 'react-dom';

// Выбираем элемент, внутри которого мы хотим нарисовать форму
const root = document.getElementById('root');

// Создаем элемент формы
const form = React.createElement(
  'div',
  { className: 'editor' },
  // Элемент состоит из трех частей: имя тега, атрибуты, дочерние элементы
  // HTML эквивалент: <input placeholder="Ключ заметки" />
  React.createElement('input', { placeholder: 'Ключ заметки' }),
  React.createElement('textarea', { placeholder: 'Текст заметки' }),
  React.createElement('button', null, 'Отменить'),
  // HTML эквивалент: <button>Сохранить</button>
  React.createElement('button', null, 'Сохранить')
);

// Отображаем форму на экране
ReactDOM.render(form, root);
```

Как пользоваться?

```
// Импортируем необходимые библиотеки
import React from 'react';
import ReactDOM from 'react-dom';

// Выбираем элемент, внутри которого мы хотим нарисовать форму
const root = document.getElementById('root');

// Создаем элемент формы
const form = React.createElement(
  'div',
  { className: 'editor' },
  // Элемент состоит из трех частей: имя тега, атрибуты, дочерние элементы
  // HTML эквивалент: <input placeholder="Ключ заметки" />
  React.createElement('input', { placeholder: 'Ключ заметки' }),
  React.createElement('textarea', { placeholder: 'Текст заметки' }),
  React.createElement('button', null, 'Отменить'),
  // HTML эквивалент: <button>Сохранить</button>
  React.createElement('button', null, 'Сохранить')
);

// Отображаем форму на экране
ReactDOM.render(form, root);
```



```
const element = <h1>Что такое JSX?</h1>;
```

JSX — это расширение языка JavaScript,

которое поддерживается в TypeScript

JSX позволяет создавать React-элементы

более просто и наглядно

JSX

```
<select multiple>
  <option value="Пункт 1">Пункт 1</option>
  <option selected value="Пункт 2">Пункт 2</option>
</select>
```

JS

```
React.createElement(
  'select',
  { multiple: true },
  React.createElement(
    'option',
    { value: 'Пункт 1' },
    'Пункт 1'
  ),
  React.createElement(
    'option',
    { selected: true, value: 'Пункт 2' },
    'Пункт 2'
  )
);
```

JSX Live Editor

21

Отличия JSX от HTML

Все `html` атрибуты именуются в `camelCase` стиле

HTML

```
<input
  class="name"
  tabindex="2"
  onchange="console.log('changed!');"
/>
```

JSX

```
<input
  className="name"
  tabIndex="2"
  onChange={event => console.log('changed!')}
/>
```

Все элементы должны быть закрыты

HTML

```
<div>
  <span>Введите ваше имя:</span><br>
  <input type="text">
</div>
```

JSX

```
<div>
  <span>Введите ваше имя:</span><br />
  <input type="text" />
</div>
```

Имена пользовательских компонентов должны начинаться с заглавной буквы

```
<Article />    React.createElement(Article, null);
```

```
<article />    React.createElement("article", null);
```

Может быть только **один** корневой
элемент

```
<p>Первый абзац</p>
<p>Второй абзац</p>
```

SyntaxError: Adjacent JSX elements must be wrapped in an enclosing tag

Может быть только **один** корневой элемент

```
<div>
    <p>Первый абзац</p>
    <p>Второй абзац</p>
</div>
```

```
<React.Fragment>
    <p>Первый абзац</p>
    <p>Второй абзац</p>
</React.Fragment>
```

Так как JSX превратится в **валидный JavaScript**, то в нем можно использовать любые JavaScript выражения

JavaScript выражения должны быть заключены в {}

```
const user = { name: 'Студент' };
```

```
const element = <div>Привет, {user.name}</div>;
```

Всё содержимое JSX экранируется

```
const html = '<strong>Мир</strong>';
```

```
const element = <div>Привет, {html}</div>;
```

Привет, **Мир**

Однако, есть выход

```
const html = 'Привет, <strong>Мир</strong>!';  
  
const element = (  
  <div dangerouslySetInnerHTML={{ __html: html }} />  
);
```

Привет, Мир!

Компоненты

Components

Компоненты предоставляют механизм разбиения интерфейса на небольшие независимые части, которые реализуются по отдельности

Рассмотрим создание компонента на этом примере

```
import React from 'react';
import ReactDOM from 'react-dom';

const root = document.getElementById('root');

ReactDOM.render(
    // Весь код, отвечающий за форму, можно вынести в компонент
    <div className="editor">
        <input placeholder="Ключ заметки" />
        <textarea placeholder="Текст заметки" />
        <button>Отменить</button>
        <button>Сохранить</button>
    </div>,
    root
);
```

Рассмотрим создание компонента на этом примере

```
import React from 'react';
import ReactDOM from 'react-dom';

const root = document.getElementById('root');

ReactDOM.render(
  // Весь код, отвечающий за форму, можно вынести в компонент
  <div className="editor">
    <input placeholder="Ключ заметки" />
    <textarea placeholder="Текст заметки" />
    <button>Отменить</button>
    <button>Сохранить</button>
  </div>,
  root
);
```

В React компоненты бывают двух типов:
Функциональные и **Классовые**

1. Функциональный компонент (Function component)

```
import React from 'react';

function Editor() {
  return (
    <div className="editor">
      <input placeholder="Ключ заметки" />
      <textarea placeholder="Текст заметки" />
      <button>Отменить</button>
      <button>Сохранить</button>
    </div>
  );
}
```

2. Классовый компонент (Class component)

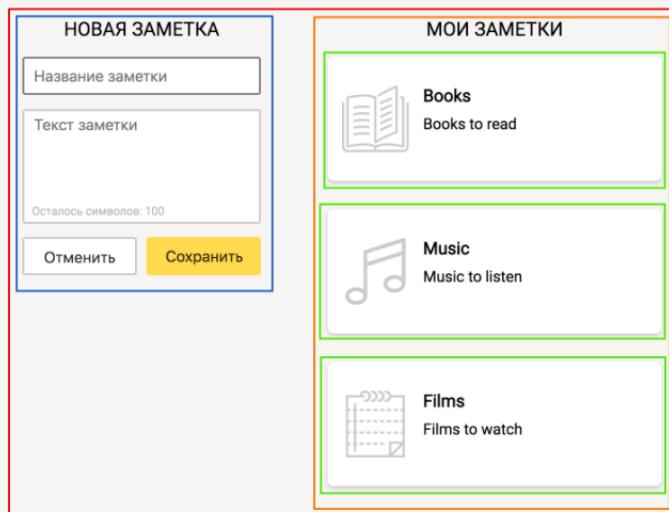
```
import React, { Component } from 'react';

class Editor extends Component {
  render() {
    return (
      <div className="editor">
        <input placeholder="Ключ заметки" />
        <textarea placeholder="Текст заметки" />
        <button>Отменить</button>
        <button>Сохранить</button>
      </div>
    );
  }
}
```

Почти всё в приложении, написанном на React, будет **компонентом**. Неважно что это — форма, компонент заметки или целая страница

Это становится возможным благодаря механизму **композиции или объединения компонентов**. Любой компонент в своей реализации может использовать любой другой компонент

Объединение компонентов



Объединение компонентов

```
import Editor from './Editor';
import Notes from './Notes';

function NotesApp() {
  return (
    <div className="notes-app">
      <Editor />
      <Notes />
    </div>
  );
}
```

Атрибуты Props

Атрибуты (Props)

Любой компонент может получать на вход **пропсы**, подобно тому как функции принимают аргументы, а html-элементы атрибуты

Именно пропсы позволяют делать компоненты универсальными

Атрибуты (Props)

```
function Notes() {  
  return (  
    <div className="notes">  
      <Note name="Books" text="Books to read" />  
      <Note name="Music" text="Music to listen" />  
      <Note name="Films" text="Films to watch" />  
    </div>  
  );  
}
```

Атрибуты (Props)

В функциональных компонентах пропсы доступны в **объекте**, который будет передан в **первый аргумент** функции-компонента

Атрибуты (Props)

```
interface NoteProps {  
    name: string;  
    text: string;  
}  
  
function Note(props: NoteProps) {  
    return (  
        <div className="note">  
            <h1>{props.name}</h1>  
            <p>{props.text}</p>  
        </div>  
    );  
}
```

Атрибуты (Props)

В классовых компонентах пропсы доступны
в любом методе класса через объект
this.props

Атрибуты (Props)

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

interface NoteProps {
    name: string;
    text: string;
}

class Note extends Component<NoteProps> {
    render() {
        return (
            <div className="note">
                <h1>{this.props.name}</h1>
                <p>{this.props.text}</p>
            </div>
        );
    }
}
```

ПОТОМКИ

Children

Потомки (Children)

children - это зарезервированное название
одного из пропсов

Этот механизм позволяет передавать
дочерние элементы более наглядно и явно,
подобно тому как это делается в HTML

ПОТОМКИ (Children)

```
function Notes() {  
    return (  
        <div className="notes">  
            <Note name="Books">  
                Books to read  
            </Note>  
  
            <Note name="Films">  
                <p>Films to read</p>  
                <button>Like</button>  
            </Note>  
  
            ...  
        </div>  
    );  
}
```

ПОТОМКИ (Children)

```
import React, { ReactDOM } from 'react';

interface NoteProps {
    // ReactDOM – тип, который описывает всё что может отрендерить React
    children: ReactDOM;
    name: string;
}

function Note(props: NoteProps) {
    return (
        <div className="note">
            <h1 className="note__title">
                {props.name}
            </h1>
            <div className="note__content">
                {props.children}
            </div>
        </div>
    );
}
```

ПОТОМКИ (Children)

```
import React, { ReactDOM } from 'react';

interface NoteProps {
    // ReactDOM – тип, который описывает всё что может отрендерить React
    children: ReactDOM;
    name: string;
}

function Note(props: NoteProps) {
    return (
        <div className="note">
            <h1 className="note__title">
                {props.name}
            </h1>
            <div className="note__content">
                {props.children}
            </div>
        </div>
    );
}
```

Условный рендеринг



Warning

Non labore cillum nulla consequat ea
proident sit dolore sint sunt ad
nostrud cupidatat laboris.



Books

Cillum est veniam culpa culpa sunt
sunt. Enim pariatur et irure sint nulla
quis non ea dolor Lorem in duis.
Commodo non laboris exercitation
consectetur adipisicing enim ad
occaecat dolor elit quis qui irure nisi.

Условный рендеринг

Так как JSX позволяет использовать произвольные JavaScript выражения, то проблемы необходимости рендеринга в зависимости от тех или иных условий решаются стандартными средствами языка

Условный рендеринг

```
function Note(props: NoteProps) {  
    return (  
  
        <div className="note">  
            <h2 className="note__title">  
                {props.type === 'warning'  
                    ? 'Warning'  
                    : props.name  
                }  
            </h2>  
            ...  
        </div>  
    );  
}
```

Условный рендеринг

```
function Note(props: NoteProps) {  
  return (  
    <div className="note">  
      <h2 className="note__title">  
        {props.type === 'warning'  
         ? 'Warning'  
         : props.name  
        }  
      </h2>  
      . . .  
    </div>  
  );  
}
```

Работа со списками



Warning

Non labore cillum nulla consequat ea
proident sit dolore sint sunt ad
nostrud cupidatat laboris.



Books

Cillum est veniam culpa culpa sunt
sunt. Enim pariatur et irure sint nulla
quis non ea dolor Lorem in duis.
Commodo non laboris exercitation
consectetur adipisicing enim ad
occaecat dolor elit quis qui irure nisi.

Работа со списками

Проблемы со списками однотипных элементов решаются аналогично

Работа со списками

```
function NotesList() {  
  return (  
    <div className="notes-list">  
      <Note name="Books" text="Books to read" />  
      <Note name="Films" text="Films to watch" />  
      <Note name="Music" text="Music to listen" />  
    </div>  
  );  
}
```

Работа со списками

```
function NotesList({ notes }: NotesListProps) {  
  return (  
    <div className="notes-list">  
      {notes.map(note => (  
        <Note name={note.name} text={note.text} />  
      ))}  
    </div>  
  );  
}
```

Однако, при рендеринге, нас ожидает ошибка в консоли браузера

Each child in an array or iterator should have a unique "key" prop.
Check the render method of **NotesList**. See <https://fb.me/react-warning-keys> for more information.

Работа со списками

```
function NotesList({ notes }: NotesListProps) {  
  return (  
    <div className="notes-list">  
      {notes.map(note => (  
        <Note  
          name={note.name}  
          text={note.text}  
          key={note.id} // Избавляемся от ошибки  
        />  
      ))}  
    </div>  
  );  
}
```

Чтобы её исправить достаточно добавить уникальный пропс `key` к каждому из элементов списка

Работа со списками

Ключи (keys) помогают React определять, какие элементы были изменены, добавлены или удалены. Их необходимо указывать, чтобы React мог сопоставлять элементы массива с течением времени

Работа со списками

Key должен однозначно определять элемент списка и быть стабильным

Повторяющиеся key в рамках одного списка недопустимы

Использовать индекс элемента в массиве лучше только тогда, когда другого выхода нет

Почему использовать индекс в качестве key это плохо

Состояние компонента (State)

Состояние компонента (State)

Состояние компонента — это механизм, который позволяет сделать его «живым»

Основное отличие состояния от пропсов в том, что состояние доступно только самому компоненту

Состояние компонента (State)

Любой компонент автоматически отреагирует на все изменения собственного состояния и будет перерисован

Состояние компонента (State)

Чтобы компонент узнал о том, что
состояние изменилось, делать это следует
только через **специальный метод**

Состояние компонента (State)

До недавнего времени использование состояния было возможно только в классовых компонентах

Однако, сейчас есть механизм и для функциональных компонентов — Хуки (Hooks)

Мы же сейчас рассмотрим только классовые компоненты

Состояние компонента (State)

Состояние компонента (State)

```
// Определяем интерфейс состояния компонента
interface NoteState {
  isReadMoreClicked: boolean;
}

class Note extends Component<NoteProps, NoteState> {
  // Определяем начальное состояние компонента
  state: NoteState = { isReadMoreClicked: false }

  // Создаем обработчик клика по кнопке «Read more»
  handleReadMoreClick = () => {
    // Изменяем состояние через метод this.setState()
    this.setState({ isReadMoreClicked: true });
  }
}

...
```

Состояние компонента (State)

```
// Определяем интерфейс состояния компонента
interface NoteState {
  isReadMoreClicked: boolean;
}

class Note extends Component<NoteProps, NoteState> {
  // Определяем начальное состояние компонента
  state: NoteState = { isReadMoreClicked: false }

  // Создаем обработчик клика по кнопке «Read more»
  handleReadMoreClick = () => {
    // Изменяем состояние через метод this.setState()
    this.setState({ isReadMoreClicked: true });
  }
}

...
```

Состояние компонента (State)

```
// Определяем интерфейс состояния компонента
interface NoteState {
  isReadMoreClicked: boolean;
}

class Note extends Component<NoteProps, NoteState> {
  // Определяем начальное состояние компонента
  state: NoteState = { isReadMoreClicked: false }

  // Создаем обработчик клика по кнопке «Read more»
  handleReadMoreClick = () => {
    // Изменяем состояние через метод this.setState()
    this.setState({ isReadMoreClicked: true });
  }
}

...
```

Состояние компонента (State)

```
// Определяем начальное состояние компонента
interface NoteState {
  isReadMoreClicked: boolean;
}

class Note extends Component<NoteProps, NoteState> {
  // Определяем начальное состояние компонента
  state: NoteState = { isReadMoreClicked: false }

  // Создаем обработчик клика по кнопке «Read more»
  handleReadMoreClick = () => {
    // Изменяем состояние через метод this.setState()
    this.setState({ isReadMoreClicked: true });
  }
}

...
```

Состояние компонента (State)

```
class Note extends Component<NoteProps, NoteState> {
    ...
    render() {
        return (
            <div className="note">
                <div className="note__name">Books</div>
                <div className="note__text">Books to read</div>
                // Доступ к текущему состоянию можно получить через this.state
                {this.state.isReadMoreClicked
                    ? <div className="note__additional-text">Additional text</div>
                    : <button onClick={this.handleReadMoreClick}>Read more</button>
                }
            </div>
        );
    }
}
```

Состояние компонента (State)

```
class Note extends Component<NoteProps, NoteState> {  
    ...  
  
    render() {  
        return (  
            <div className="note">  
                <div className="note__name">Books</div>  
                <div className="note__text">Books to read</div>  
                // Доступ к текущему состоянию можно получить через this.state  
                {this.state.isReadMoreClicked  
                    ? <div className="note__additional-text">Additional text</div>  
                    : <button onClick={this.handleReadMoreClick}>Read more</button>  
                }  
            </div>  
        );  
    }  
}
```

Правила работы с состоянием

1. Не изменяйте состояние напрямую

```
/**  
 * Неправильно, так как React не сможет узнать о том,  
 * что состояние было изменено  
 */  
this.state.isReadMoreClicked = true;  
  
// Правильно  
this.setState({ isReadMoreClicked: true });
```

2. Для повышения производительности React может группировать обновления состояния

```
/**  
 * Неправильно, так как обновления будут сгруппированы и каждое  
 * из них обратится к старому значению this.state.counter.  
 * Как результат значение counter увеличится на 1, вместо 2  
 */  
this.setState({ counter: this.state.counter + 1 });  
this.setState({ counter: this.state.counter + 1 });  
  
/**  
 * Правильно. Передавая функцию в качестве аргумента, мы делаем  
 * изменения атомарными. Каждое из них получит актуальное значение состояния  
 */  
this.setState(state => ({ counter: state.counter + 1 }));  
this.setState(state => ({ counter: state.counter + 1 }));
```

3. Можно обновлять не все состояние целиком

```
interface State {  
    notes: Note[];  
    isReadMoreClicked: boolean;  
}  
  
/**  
 * Если в вашем состоянии больше одного поля, то обновлять можно только  
 * то, что действительно изменилось. Все остальные значения React  
 * оставит без изменения  
 */  
this.setState({ isReadMoreClicked: true });
```

Жизненный цикл компонента

Component life-cycle

Жизненный цикл компонента (Component life-cycle)

React предоставляет нам возможность вызывать различные методы в разные моменты жизни компонента

Методы жизненного цикла доступны
ТОЛЬКО В КЛАССОВЫХ КОМПОНЕНТАХ

Этапы жизненного цикла

1. Монтирование компонента (Mounting)
2. Изменение пропсов или состояния (Updating)
3. Удаление компонента (Unmounting)

Монтирование (Mounting)

Монтированием называется первоначальный
рендеринг компонента в DOM

Монтирование (Mounting)

Порядок вызова методов

```
constructor
  ↓
(static) getDerivedStateFromProps
  ↓
render
  ↓
componentDidMount
```

Обновление компонента (Updating)

Обновление компонента это самая долгая стадия жизни компонента. На этом этапе могут происходить изменения пропсов или состояния компонента

Изменение пропсов (Updating)

Порядок вызова методов

(static) getDerivedStateFromProps

↓

shouldComponentUpdate

↓

render

↓

componentDidUpdate

Изменение состояния (Updating)

Порядок вызова методов

shouldComponentUpdate

↓

render

↓

componentDidUpdate

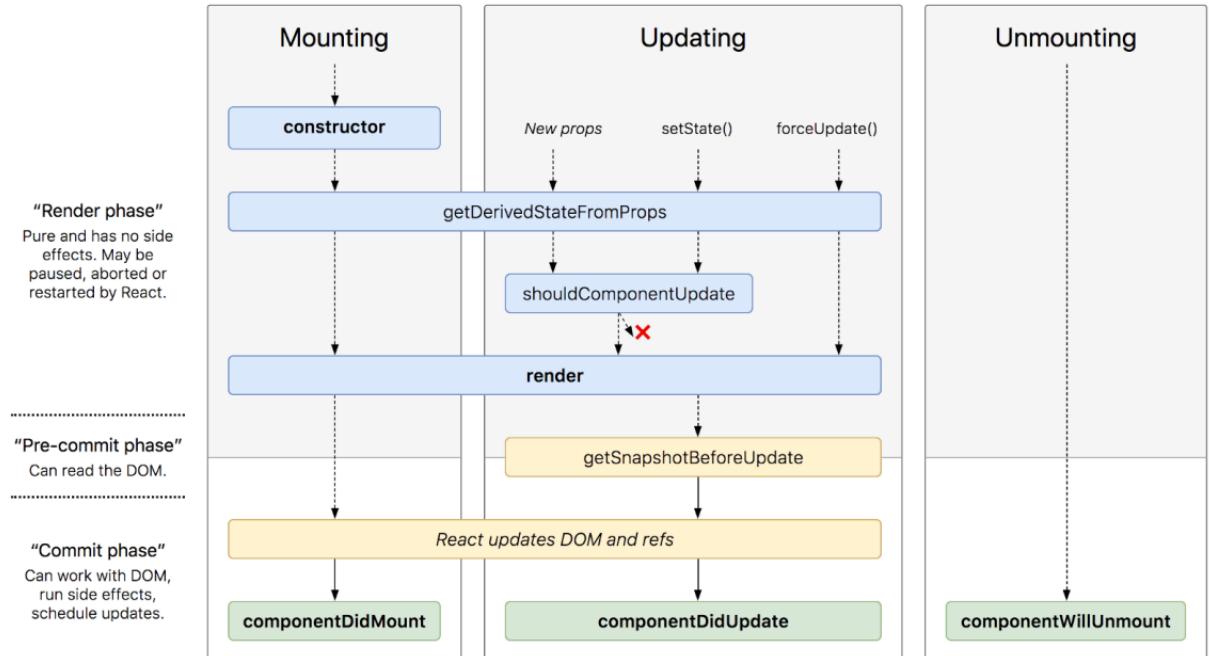
Удаление компонента (Unmounting)

Удалением или «размонтированием» компонента называется удаление соответствующего ему узла в DOM дереве

Удаление компонента (Unmounting)

Порядок вызова методов

`componentWillUnmount`



Рассмотрим использование методов жизненного цикла на примере

Реализуем кнопку «Вернуться наверх страницы», которая появляется если пользователь прокрутил страницу более чем на 300 пикселей

Жизненный цикл компонента (Component life-cycle)

```
class Notes extends Component<NotesProps, NotesState> {
    // В состоянии компонента храним признак, говорящий о том нужно ли показывать кнопку
    state: NotesState = { backToTop: false }

    // После монтирования компонента мы можем работать с DOM
    // Поэтому здесь будет безопасно подписаться на событие scroll
    componentDidMount() {
        window.addEventListener('scroll', this.handleScroll);
    }

    // Перед тем как компонент будет удален, нам необходимо отписаться
    // Это позволит нам подчистить за собой и избежать утечек памяти
    componentWillUnmount() {
        window.removeEventListener('scroll', this.handleScroll);
    }

    handleScroll = () => {
        // Если кнопка еще не показана, и пользователь достаточно прокрутил страницу
        // Показываем кнопку «Наверх»
        if (!this.state.backToTop && document.documentElement.scrollTop > 300) {
            this.setState({ backToTop: true });
        }
    }

    ...
}

...
}
```

Жизненный цикл компонента (Component life-cycle)

```
class Notes extends Component<NotesProps, NotesState> {
    // В состоянии компонента храним признак, говорящий о том нужно ли показывать кнопку
    state: NotesState = { backToTop: false }

    // После монтирования компонента мы можем работать с DOM
    // Поэтому здесь будет безопасно подписаться на событие scroll
    componentDidMount() {
        window.addEventListener('scroll', this.handleScroll);
    }

    // Перед тем как компонент будет удален, нам необходимо отписаться
    // Это позволит нам подчистить за собой и избежать утечек памяти
    componentWillUnmount() {
        window.removeEventListener('scroll', this.handleScroll);
    }

    handleScroll = () => {
        // Если кнопка еще не показана, и пользователь достаточно прокрутил страницу
        // Показываем кнопку «Наверх»
        if (!this.state.backToTop && document.documentElement.scrollTop > 300) {
            this.setState({ backToTop: true });
        }
    }

    ...
}

...
}
```

Жизненный цикл компонента (Component life-cycle)

```
class Notes extends Component<NotesProps, NotesState> {
    // В состоянии компонента храним признак, говорящий о том нужно ли показывать кнопку
    state: NotesState = { backToTop: false }

    // После монтирования компонента мы можем работать с DOM
    // Поэтому здесь будет безопасно подписаться на событие scroll
    componentDidMount() {
        window.addEventListener('scroll', this.handleScroll);
    }

    // Перед тем как компонент будет удален, нам необходимо отписаться
    // Это позволит нам подчистить за собой и избежать утечек памяти
    componentWillUnmount() {
        window.removeEventListener('scroll', this.handleScroll);
    }

    handleScroll = () => {
        // Если кнопка еще не показана, и пользователь достаточно прокрутил страницу
        // Показываем кнопку «Наверх»
        if (!this.state.backToTop && document.documentElement.scrollTop > 300) {
            this.setState({ backToTop: true });
        }
    }

    ...
}

...
}
```

Работа с формами

Работа с формами

Как получить данные из формы?

Как обрабатывать события?

Компоненты в React можно разделить на два вида, по способу взаимодействия:

1. Неуправляемые (Uncontrolled)
2. Управляемые (Controlled)

1. Неуправляемые компоненты (Uncontrolled components)

DOM управляет текущим состоянием компонента

1. Неуправляемые компоненты

(Uncontrolled components)

Так как в неуправляемых компонентах текущим состоянием управляет **DOM**, а не React, то при отправке формы необходимо получить текущее значение из DOM

Помочь в этом могут **ссылки на DOM элементы** (Refs)

Ссылка на DOM элемент (Ref)

```
import React, { Component, createRef } from 'react';

class UncontrolledForm extends Component {
  // Создаем контейнер для будущей ссылки на DOM элемент <input />

  inputRef = createRef<HTMLInputElement>()

  handleSubmit = () => {
    if (this.inputRef.current) {
      // Отправляем текущее значение введенное в поле ввода
      this.makeSomeApiRequest(this.inputRef.current.value);
    }
  }

  render() {
    return (
      <div>
        // Назначаем ссылку нужному элементу. После того как элемент появится в DOM
        // this.inputRef.current будет ссылаться на этот элемент
        <input ref={this.inputRef} />
        <button onClick={this.handleSubmit}>Отправить</button>
      </div>
    );
  }
}
```

Ссылка на DOM элемент (Ref)

```
import React, { Component, createRef } from 'react';

class UncontrolledForm extends Component {
  // Создаем контейнер для будущей ссылки на DOM элемент <input />

  inputRef = createRef<HTMLInputElement>()

  handleSubmit = () => {
    if (this.inputRef.current) {
      // Отправляем текущее значение введенное в поле ввода
      this.makeSomeApiRequest(this.inputRef.current.value);
    }
  }

  render() {
    return (
      <div>
        // Назначаем ссылку нужному элементу. После того как элемент появится в DOM
        // this.inputRef.current будет ссылаться на этот элемент
        <input ref={this.inputRef} />
        <button onClick={this.handleSubmit}>Отправить</button>
      </div>
    );
  }
}
```

Ссылка на DOM элемент (Ref)

```
import React, { Component, createRef } from 'react';

class UncontrolledForm extends Component {
  // Создаем контейнер для будущей ссылки на DOM элемент <input />

  inputRef = createRef<HTMLInputElement>()

  handleSubmit = () => {
    if (this.inputRef.current) {
      // Отправляем текущее значение введенное в поле ввода
      this.makeSomeApiRequest(this.inputRef.current.value);
    }
  }

  render() {
    return (
      <div>
        // Назначаем ссылку нужному элементу. После того как элемент появится в DOM
        // this.inputRef.current будет ссылаться на этот элемент
        <input ref={this.inputRef} />
        <button onClick={this.handleSubmit}>Отправить</button>
      </div>
    );
  }
}
```

Ссылка на DOM элемент (Ref)

```
import React, { Component, createRef } from 'react';

class UncontrolledForm extends Component {
  // Создаем контейнер для будущей ссылки на DOM элемент <input />

  inputRef = createRef<HTMLInputElement>()

  handleSubmit = () => {
    if (this.inputRef.current) {
      // Отправляем текущее значение введенное в поле ввода
      this.makeSomeApiRequest(this.inputRef.current.value);
    }
  }

  render() {
    return (
      <div>
        // Назначаем ссылку нужному элементу. После того как элемент появится в DOM
        // this.inputRef.current будет ссылаться на этот элемент
        <input ref={this.inputRef} />
        <button onClick={this.handleSubmit}>Отправить</button>
      </div>
    );
  }
}
```

1. Неуправляемые компоненты

(Uncontrolled components)

- + Простота
- + Один обработчик на всю форму
- Недостаточный контроль из кода

2. Управляемые компоненты (Controlled components)

React управляет текущим состоянием компонента

2. Управляемые компоненты

```
import React, { Component, FormEvent } from 'react';

class ControlledForm extends Component<ControlledFormProps, ControlledFormState> {
    // Храним текущее состояние поля ввода в локальном состоянии компонента

    state: ControlledFormState = { value: '' }

    // Изменяем его вручную, не опираясь на DOM
    handleChange = (event: FormEvent<HTMLInputElement>) => {
        this.setState({ value: event.target.value });
    }

    // Отправляем текущее состояние, полученное из локального состояния компонента
    handleSubmit = () => this.makeSomeApiRequest(this.state.value)

    render() {
        return (
            <div>
                // Сам задаем текущее значение поля ввода, а так же обработчик события change
                <input value={this.state.value} onChange={this.handleChange} />
                <button onClick={this.handleSubmit}>Отправить</button>
            </div>
        );
    }
}
```

2. Управляемые компоненты

```
import React, { Component, FormEvent } from 'react';

class ControlledForm extends Component<ControlledFormProps, ControlledFormState> {
    // Храним текущее состояние поля ввода в локальном состоянии компонента

    state: ControlledFormState = { value: '' }

    // И меняем его вручную, не опираясь на DOM
    handleChange = (event: FormEvent<HTMLInputElement>) => {
        this.setState({ value: event.target.value });
    }

    // Отправляем текущее состояние, полученное из локального состояния компонента
    handleSubmit = () => this.makeSomeApiRequest(this.state.value)

    render() {
        return (
            <div>
                // Сам задаем текущее значение поля ввода, а так же обработчик события change
                <input value={this.state.value} onChange={this.handleChange} />
                <button onClick={this.handleSubmit}>Отправить</button>
            </div>
        );
    }
}
```

2. Управляемые компоненты

```
import React, { Component, FormEvent } from 'react';

class ControlledForm extends Component<ControlledFormProps, ControlledFormState> {
    // Храним текущее состояние поля ввода в локальном состоянии компонента

    state: ControlledFormState = { value: '' }

    // Изменяем его вручную, не опираясь на DOM
    handleChange = (event: FormEvent<HTMLInputElement>) => {
        this.setState({ value: event.target.value });
    }

    // Отправляем текущее состояние, полученное из локального состояния компонента
    handleSubmit = () => this.makeSomeApiRequest(this.state.value)

    render() {
        return (
            <div>
                // Сам задаем текущее значение поля ввода, а так же обработчик события change
                <input value={this.state.value} onChange={this.handleChange} />
                <button onClick={this.handleSubmit}>Отправить</button>
            </div>
        );
    }
}
```

2. Управляемые компоненты (Controlled components)

- + Состояние управляется через интерфейс
- + Полный контроль над изменениями
- Более сложное взаимодействие

Как начать работать с React ?

Способ 1 – create-react-app

```
# Устанавливаем утилиту для генерации проекта  
$ npm install -g create-react-app
```

```
# Создаем проект notes-app с использованием TypeScript  
$ create-react-app notes-app --typescript
```

```
# Переходим в папку с проектом  
$ cd notes-app
```

```
# Запускаем проект  
$ npm run start
```

Compiled successfully!

You can now view **notes-app** in the browser.
Local: <http://localhost:3000/>

Note that the development build is not optimized.
To create a production build, use `npm run build`.

create-react-app

- + Очень быстрый старт
- + Активное развитие и поддержка от разработчиков React
- + TypeScript «из коробки»
- Сложности при интеграции с собственным сервером

Способ 2 – next.js



next.js

```
# Создаем папку для проекта и переходим в неё
$ mkdir notes-app && cd notes-app

# Создаем package.json
{
  "scripts": {
    "dev": "next",
    "build": "next build",
    "start": "next start"
  }
}

# Устанавливаем необходимые зависимости
$ npm install --save next react react-dom

# Создаем файл ./pages/index.js, который будет отвечать за главную страницу
export default () => <div>Welcome to next.js!</div>

# Запускаем проект в режиме разработки
$ npm run dev

[DONE] Compiled successfully span 1773ms
> Ready on http://localhost:3000
```

next.js

- + Очень быстрый старт
- + Активное развитие и поддержка
- + Множество точек для расширения и кастомизации

next.js Свой сервер

```
// Импортируем нужные зависимости
const { parse } = require('url');

const next = require('next');

const server = require('express')();

// Создаем next.js приложение
const app = next({ dev: process.env.NODE_ENV !== 'production' });

// Чтобы отрисовать страницу вызываем метод next.js приложения - app.render()
const render = pageName => (req, res) => app.render(req, res, `/${pageName}`);
const handleRequest = (req, res) =>
  app.getRequestHandler()(req, res, parse(req.url, true));

// Запускаем next.js приложение
app.prepare().then(() => {
  // После того как next.js приложение запущено можно создать сервер
  server
    .get('/notes', render('index'))
    .get('/notes/:note', render('note'))
    .get('*', handleRequest)
    .listen(3000, () => console.log('Listening on http://localhost:3000'));
});
```