

# Лекция 8

# React

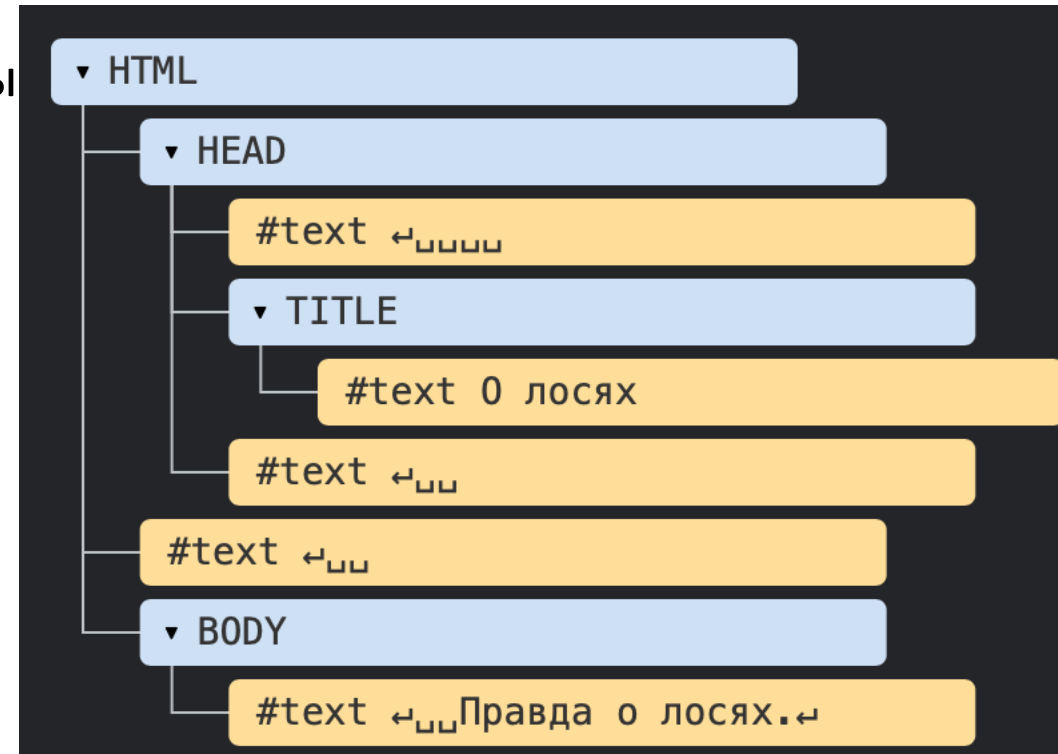
Разработка интернет приложений

Канев Антон Игоревич

# DOM

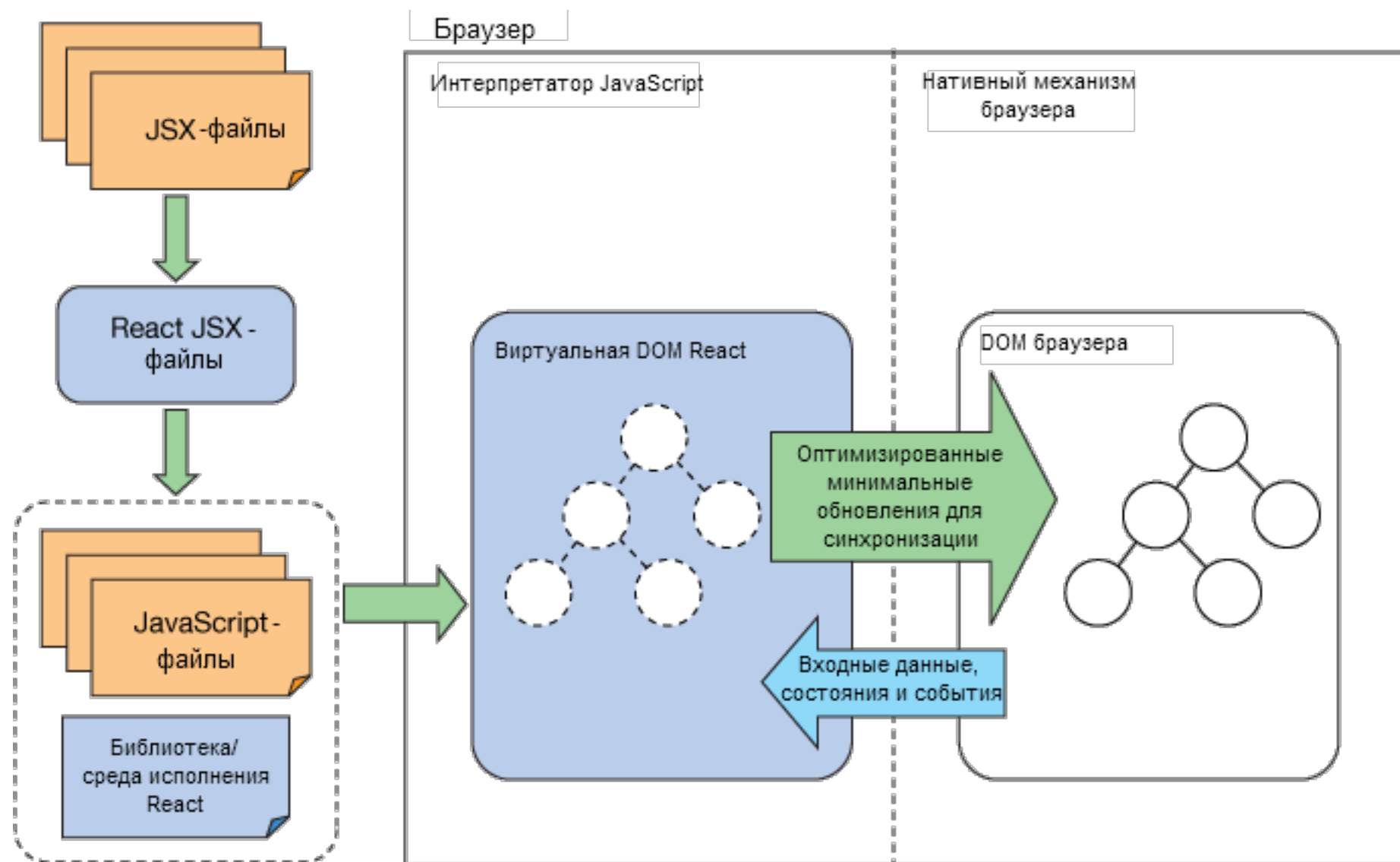
- Основой HTML-документа являются теги.
- В соответствии с объектной моделью документа («Document Object Model», коротко DOM), каждый HTML-тег является объектом. Вложенные теги являются «детьми» родительского элемента. Текст, который находится внутри тега, также является объектом.
- Все эти объекты доступны при помощи JS
- Можем использовать их для изменения страницы

```
<!DOCTYPE HTML>
<html>
<head>
  <title>0 лосях</title>
</head>
<body>
  Правда о лосях.
</body>
</html>
```

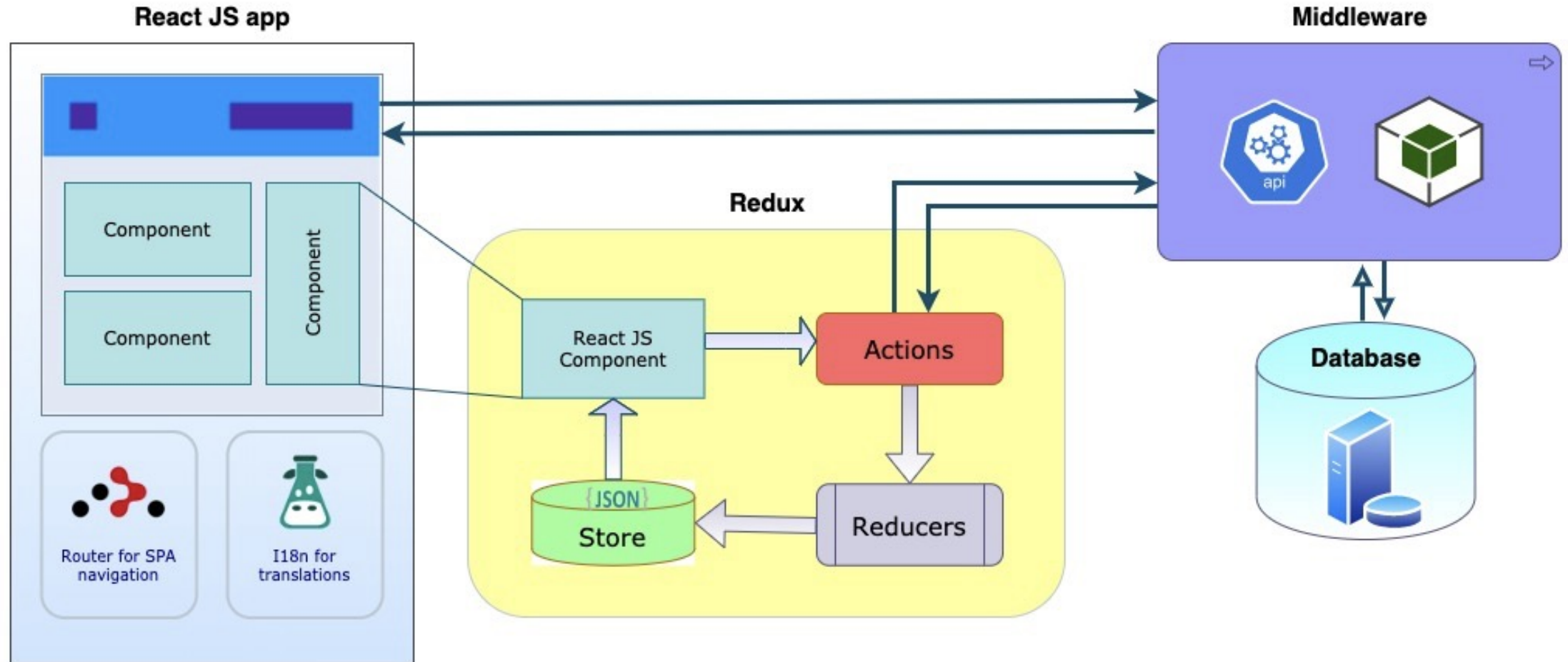


# React

- Библиотека для работы с виртуальным DOM
- Документация

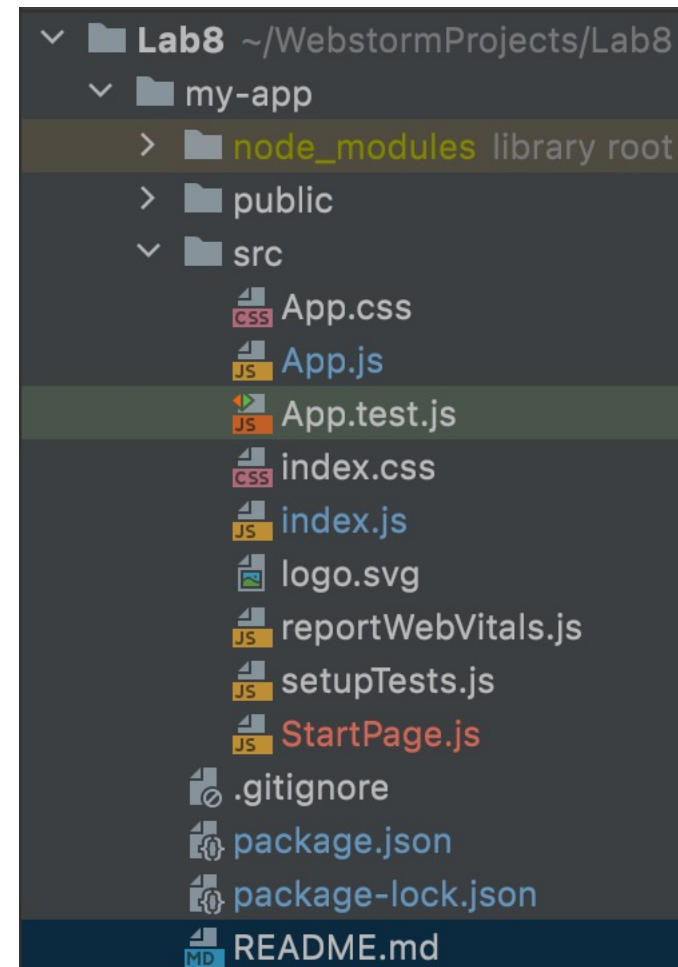
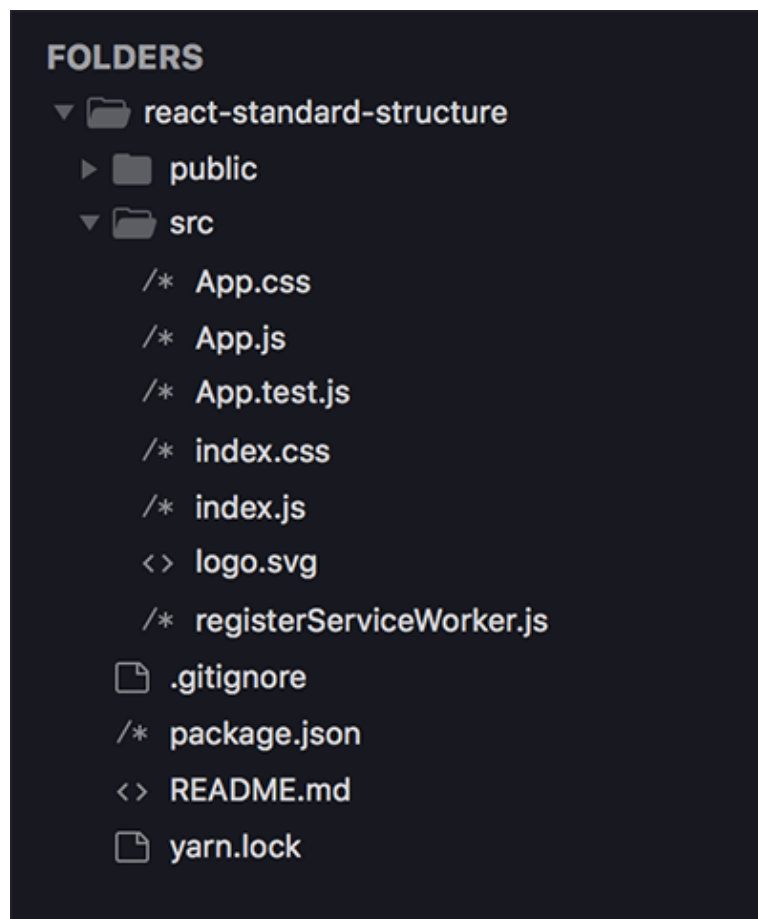


# React



# Структура проекта

- компоненты: .js, .css
- картинки: .svg



# ES6

- ECMAScript 2015

```
function foo(x, y, z) {  
    console.log(x, y, z);  
}
```

```
let arr = [1, 2, 3];  
foo(...arr); // 1 2 3
```

```
var a = 2;  
{  
    let a = 3;  
    console.log(a); // 3  
}  
console.log(a); // 2
```

```
class Task {  
    constructor() {  
        console.log("Создан экземпляр task!");  
    }  
}
```

```
showId() {  
    console.log(23);  
}
```

```
static loadAll() {  
    console.log("Загружаем все tasks...");  
}
```

```
}
```

```
function foo(...args) {  
    console.log(args);  
}  
foo(1, 2, 3, 4, 5); // [1, 2, 3, 4, 5]
```

```
// Классическое функциональное выражение  
let addition = function(a, b) {  
    return a + b;  
};
```

```
// Стрелочная функция  
let addition = (a, b) => a + b;
```

# Babel

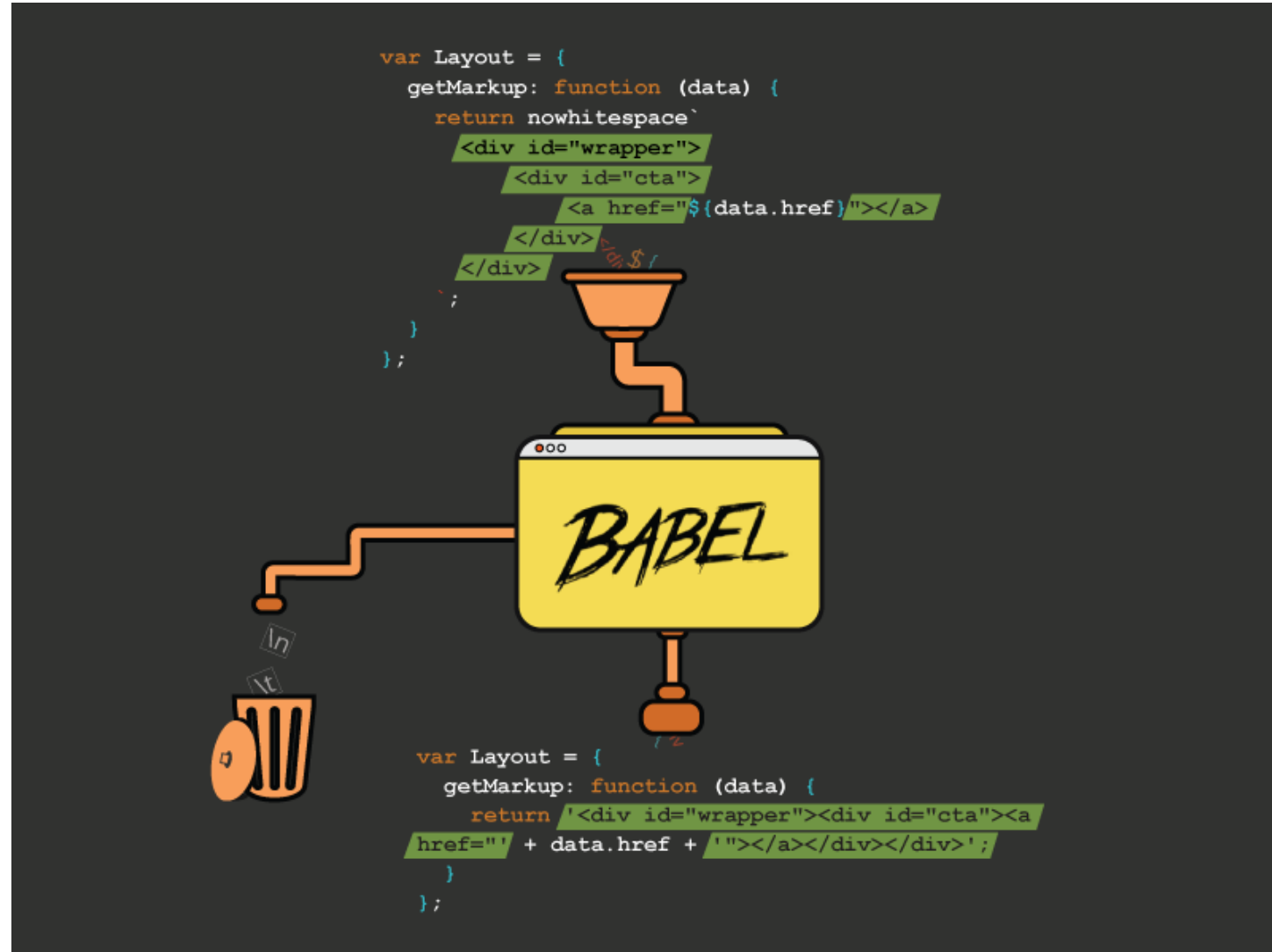
- Компилятор JS

- It turns ES2015:

```
const adding = (a, b) => a + b
```

- into old JavaScript:




```
'use strict';  
var adding = function adding(a, b) {  
  return a + b;  
};
```



# WebPack

- Сборщик модулей JS
- webpack принимает модули с зависимостями и генерирует статические ресурсы, представляющие эти модули



	index.html	1.99 KB
	index.bundle.js	19.92 KB
	index.bundle.js.map	19.98 KB
	style.bundle.css	12.64 KB
	favicon.ico	23.24 KB

```
▼ cra-app ~/Sites/oss/cli-blog/cra-app
  ▼ build
    ▼ static
      ► css
      ► js
      ► media
      {} asset-manifest.json
      🖼️ favicon.ico
      📄 index.html
  ► node_modules library root
  ► public
  ▼ src
    📄 App.css
    📄 App.js
    📄 App.test.js
    📄 index.css
    📄 index.js
    📄 logo.svg
  📄 .gitignore
  📄 package.json
  📄 README.md
  📄 yarn.lock
```

Artifact

Source files



# Index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

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

# Страница

```
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <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"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;
```

# JSX

- JSX — расширение синтаксиса JavaScript.
- Этот синтаксис выглядит как язык шаблонов, но наделён всеми языковыми возможностями JavaScript.
- В результате компиляции JSX возникают простые объекты — «React-элементы».
- React DOM использует стиль именованя camelCase для свойств вместо обычных имён HTML-атрибутов.
- Например, в JSX атрибут tabIndex станет tabIndex.
- В то время как атрибут class записывается как className, поскольку слово class уже зарезервировано в JavaScript

# Routing

```
import { BrowserRouter, Route, Switch } from "react-router-dom";

function App() {

  return (
    <BrowserRouter basename="/">
      <Switch>
        <Route exact path="/">
          <h1>Это наша стартовая страница</h1>
        </Route>
        <Route path="/new">
          <h1>Это наша страница с чем-то новеньким</h1>
        </Route>
      </Switch>
    </BrowserRouter>
  );
}

export default App;
```

# Router

- Ссылки на страницы
- Использование router

```
import { BrowserRouter, Route, Link, Switch } from "react-router-dom";

function App() {

  return (
    <BrowserRouter basename="/" >
      <div>
        <ul>
          <li>
            <Link to="/">Старт</Link>
          </li>
          <li>
            <Link to="/new">Хочу на страницу с чем-то новеньким</Link>
          </li>
        </ul>
        <hr />
        <Switch>
          <Route exact path="/">
            <h1>Это наша стартовая страница</h1>
          </Route>
          <Route path="/new">
            <h1>Это наша страница с чем-то новеньким</h1>
          </Route>
        </Switch>
      </div>
    </BrowserRouter>
  );
}

export default App;
```

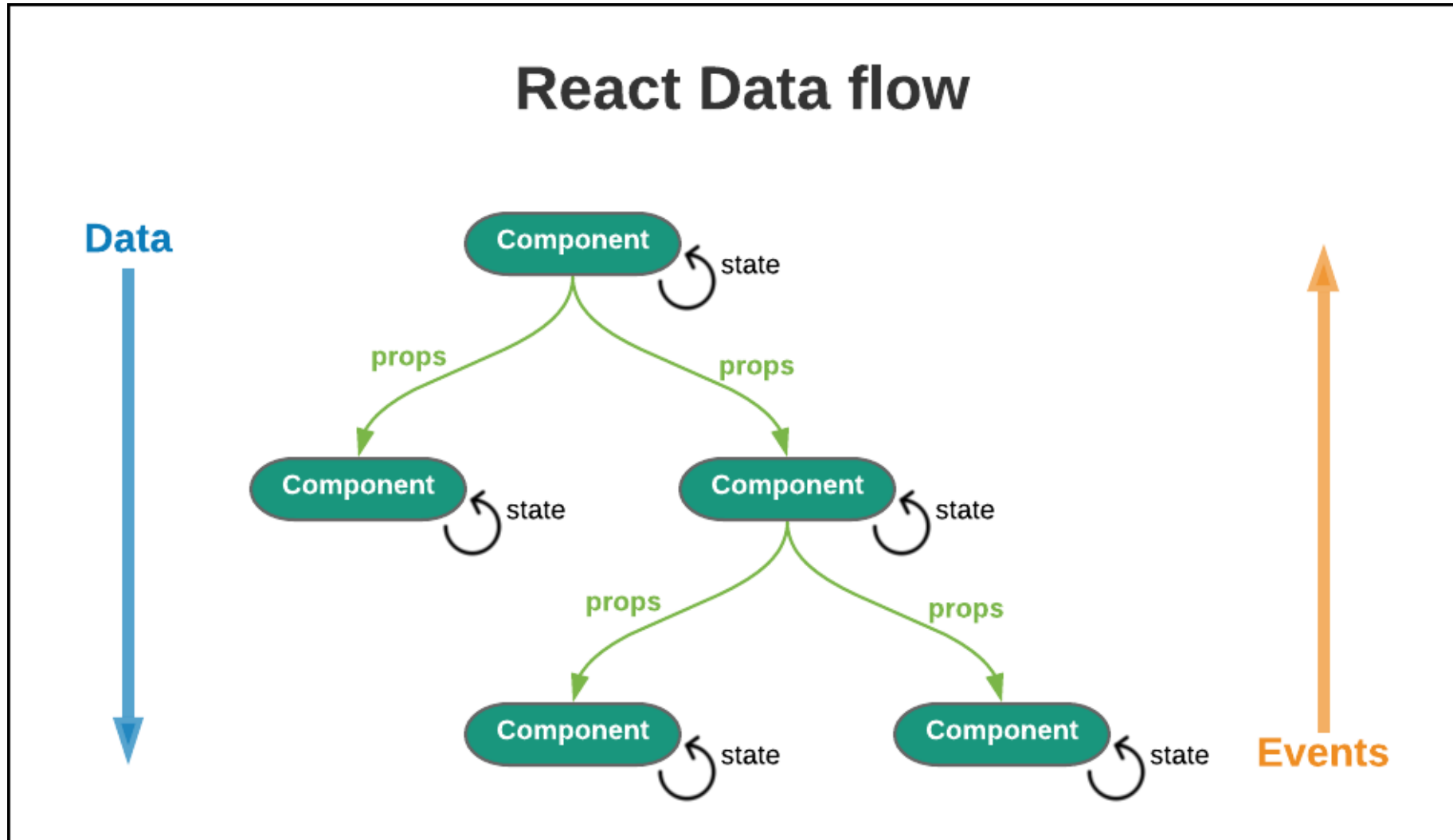
# Компонент

- React-компоненты — это повторно используемые части кода, которые возвращают React-элементы для отображения на странице.

```
function Welcome(props) {  
  return <h1>Привет, {props.name}</h1>;  
}
```

```
class Welcome extends React.Component {  
  render() {  
    return <h1>Привет, {this.props.name}</h1>;  
  }  
}
```

# Поток данных и сообщений



# Props

- props (пропсы) — это входные данные React-компонентов, передаваемые от родительского компонента дочернему компоненту.
- В любом компоненте доступны props.children. Это контент между открывающим и закрывающим тегом компонента.
- Для классовых компонентов используйте this.props.children

```
export interface BrowserRouterProps {  
  basename?: string | undefined;  
  children?: React.ReactNode;  
  getUserConfirmation?: (message: string, callback: (ok:  
    forceRefresh?: boolean | undefined;  
    keyLength?: number | undefined;  
}
```

```
<BrowserRouter basename="/">  
  <Switch>  
    <Route exact path="/">  
      <h1>Это наша стартовая страница</h1>  
    </Route>  
    <Route path="/new">  
      <h1>Это наша страница с чем-то новеньким</h1>  
    </Route>  
  </Switch>  
</BrowserRouter>
```



# Состояние

- Компонент нуждается в state, когда данные в нём со временем изменяются.
- Например, компоненту Checkbox может понадобиться состояние isChecked.
- Разница между пропсами и состоянием заключается в основном в том, что состояние нужно для управления компонентом, а пропсы для получения информации.

# Функциональные компоненты

- Описание компонентов с помощью чистых функций создает меньше кода, а значит его легче поддерживать.
- Чистые функции намного проще тестировать. Вы просто передаете props на вход и ожидаете какую то разметку.
- В будущем чистые функции будут выигрывать по скорости работы в сравнении с классами из-за отсутствия методов жизненного цикла

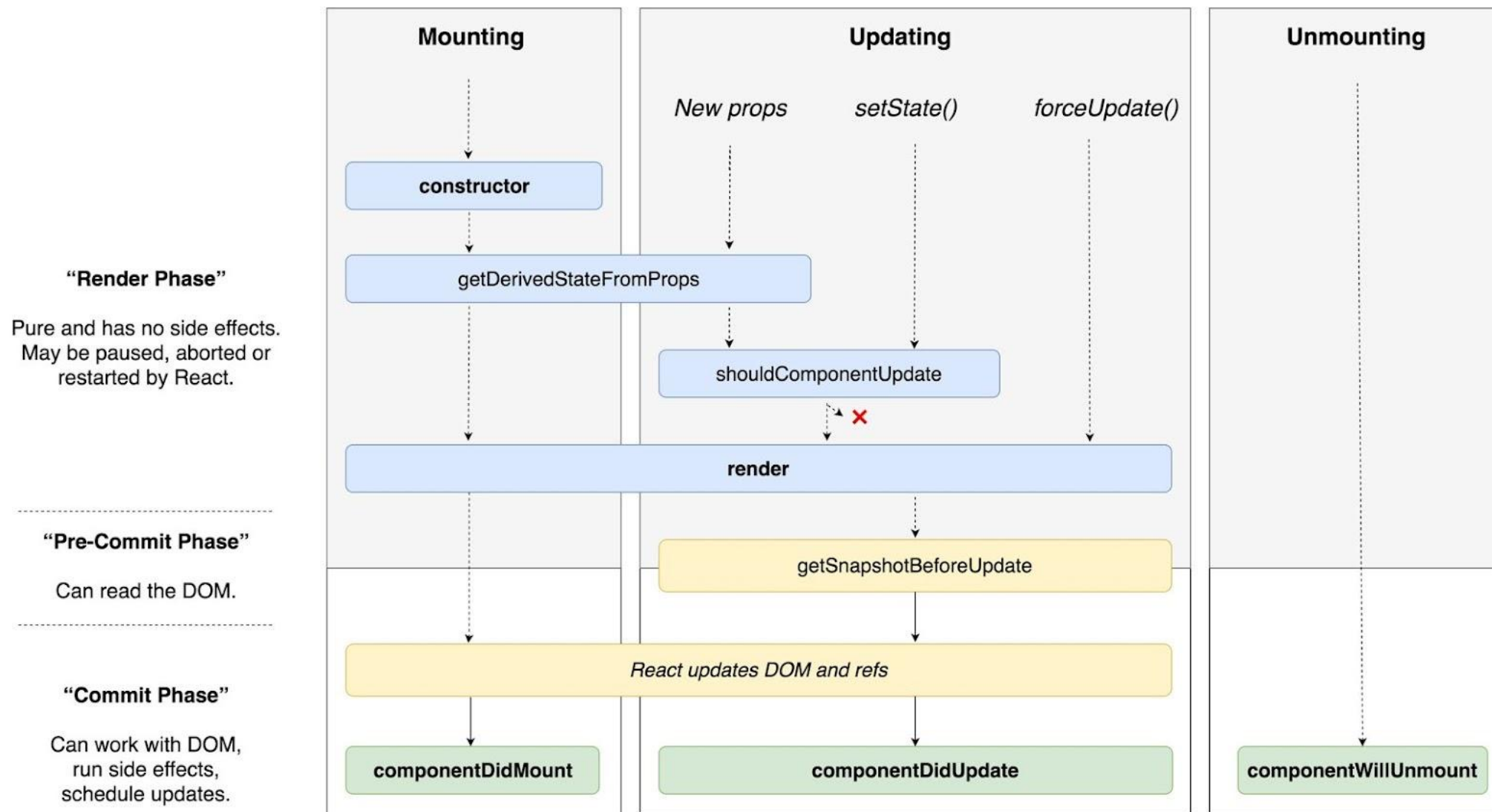
# Хуки. useState

- Хуки позволяют работать с состоянием компонентов, с методами их жизненного цикла, с другими механизмами React без использования классов.

```
class Example extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      count: 0  
    };  
  }  
}
```

```
import React, { useState } from 'react';  
  
function Example() {  
  // Объявление новой переменной состояния «count»  
  const [count, setCount] = useState(0);  
  return <div onClick={()=>setCount(count=>count++)}>{count}</div>  
}
```

# Методы жизненного цикла компонента



# Хуки

```
import React, { useEffect, useState } from 'react'
import Axios from 'axios'
```

```
export default function Hello() {
```

```
  const [Name, setName] = useState('')
```

```
  useEffect(() => {
    Axios.get('/api/user/name')
      .then(response => {
        setName(response.data.name)
      })
  }, [])
```

```
  return (
    <div>
      My name is {Name}
    </div>
  )
}
```

```
import React, { Component } from 'react'
import Axios from 'axios'
```

```
export default class Hello extends Component {
```

```
  constructor(props) {
    super(props);
    this.state = { name: '' };
  }
```

```
  componentDidMount() {
    Axios.get('/api/user/name')
      .then(response => {
        this.setState({ name: response.data.name })
      })
  }
```

```
  render() {
    return (
      <div>
        My name is {this.state.name}
      </div>
    )
  }
```

# useEffect

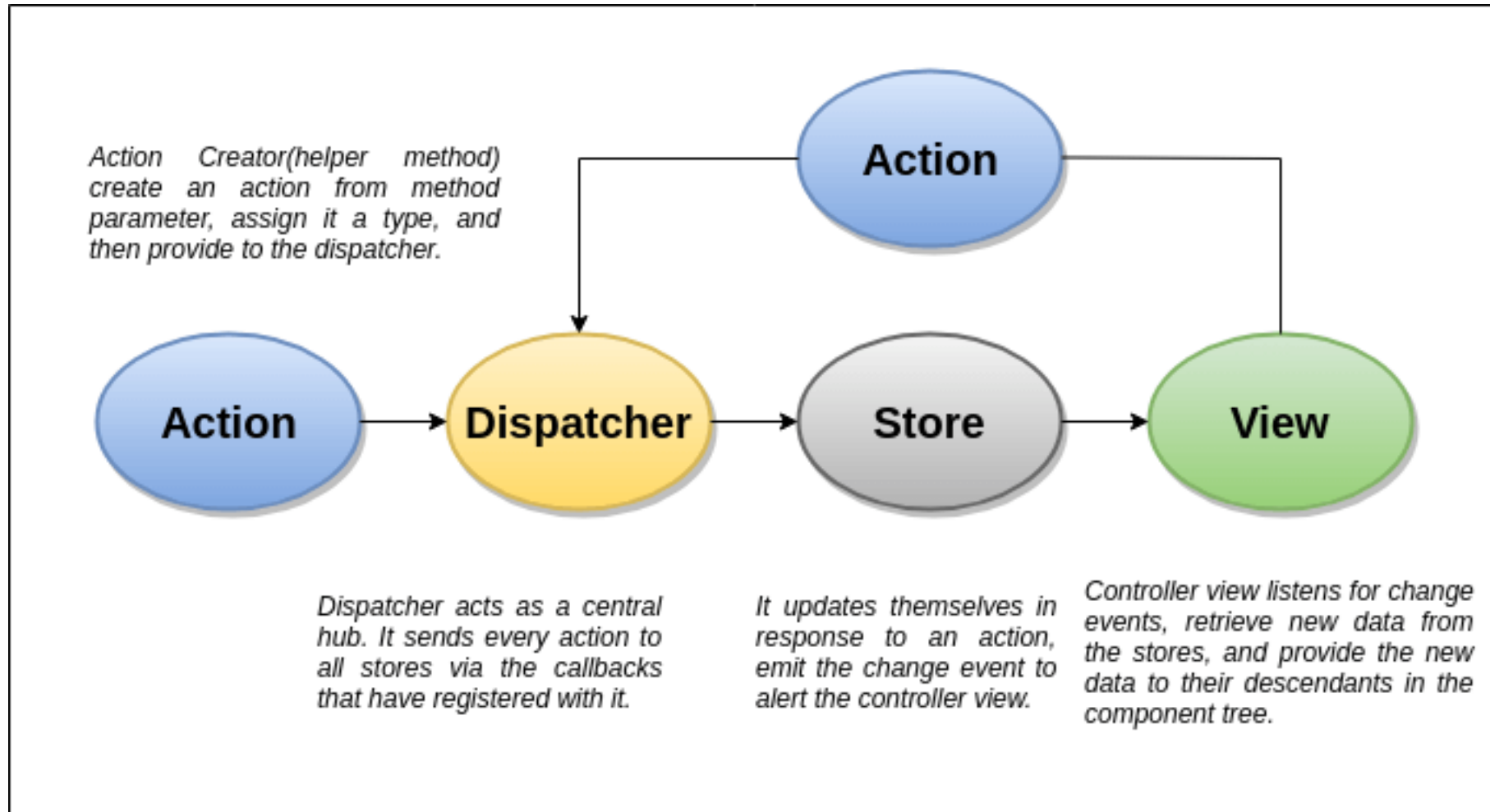
- componentDidMount()
- componentDidUpdate()
- componentWillUnmount()

```
useEffect( effect: ()=>{  
    console.log('Этот код выполняется только на первом рендере компонента')  
    // В данном примере можно наблюдать Spread syntax (Троеточие перед массивом)  
    setNames( value: names=>[...names, 'Бедный студент'])  
  
    return () => {  
        console.log('Этот код выполняется, когда компонент будет размонтирован')  
    }  
}, deps: [])  
  
useEffect( effect: ()=>{  
    console.log('Этот код выполняется каждый раз, когда изменится состояние showNames ')  
    setRandomName(names[Math.floor( x: Math.random()*names.length)])  
}, deps: [showNames])
```

# Другие хуки

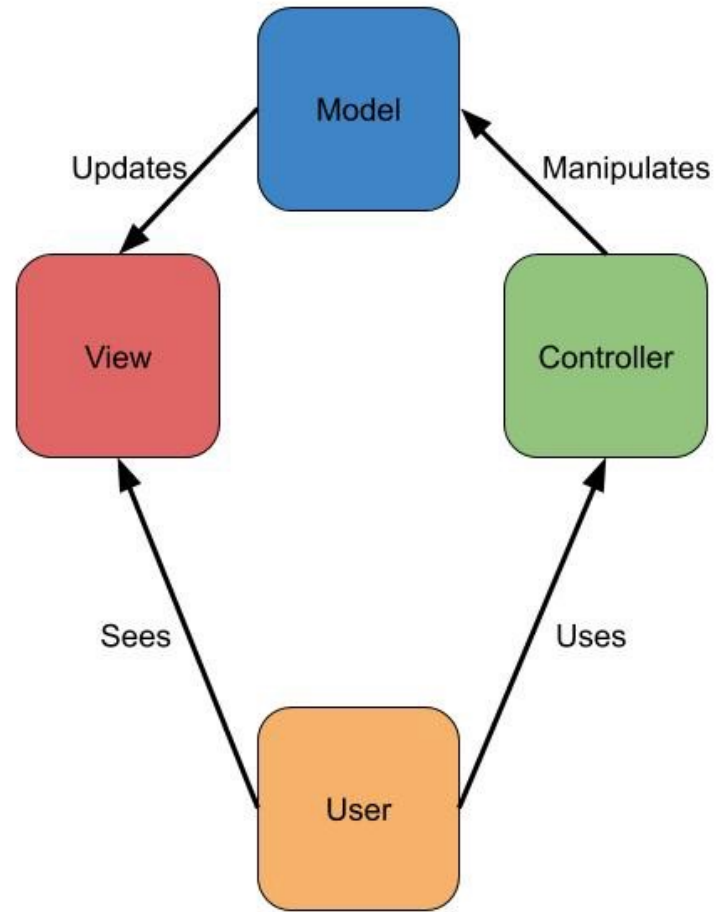
- `useContext`: позволяет работать с контекстом — с механизмом, используемым для организации совместного доступа к данным без необходимости передачи свойств.
- `useRef`: позволяет напрямую обращаться к DOM в функциональных компонентах. Обратите внимание на то, что `useRef`, в отличие от `setState`, не вызывает повторный рендеринг компонента.
- `useReducer`: хранит текущее значение состояния. Его можно сравнить с `Redux`.
- `useMemo`: используется для возврата мемоизированного значения. Может применяться в случаях, когда нужно, чтобы функция возвратила бы кешированное значение.
- `useCallback`: применяется в ситуациях, когда дочерние элементы компонента подвергаются постоянному повторному рендерингу. Он возвращает мемоизированную версию коллбэка, которая меняется лишь тогда, когда меняется одна из зависимостей.

# React MVC

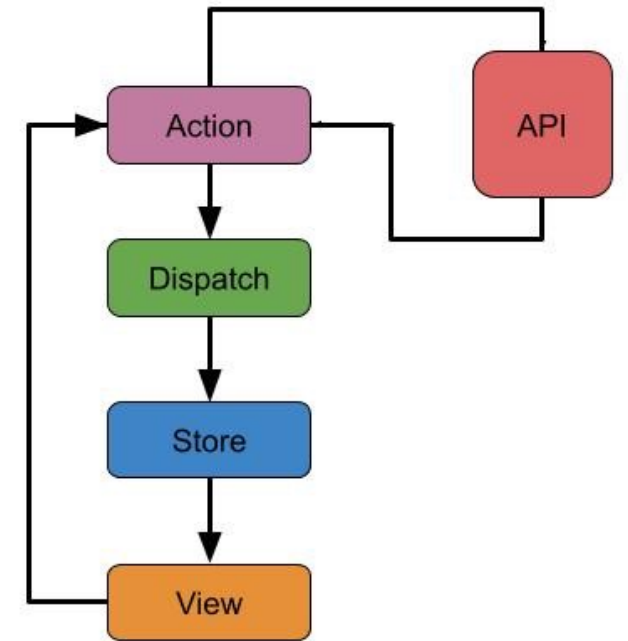




# MVC vs React



MVC base Model



React's MVC model

# React MVC

