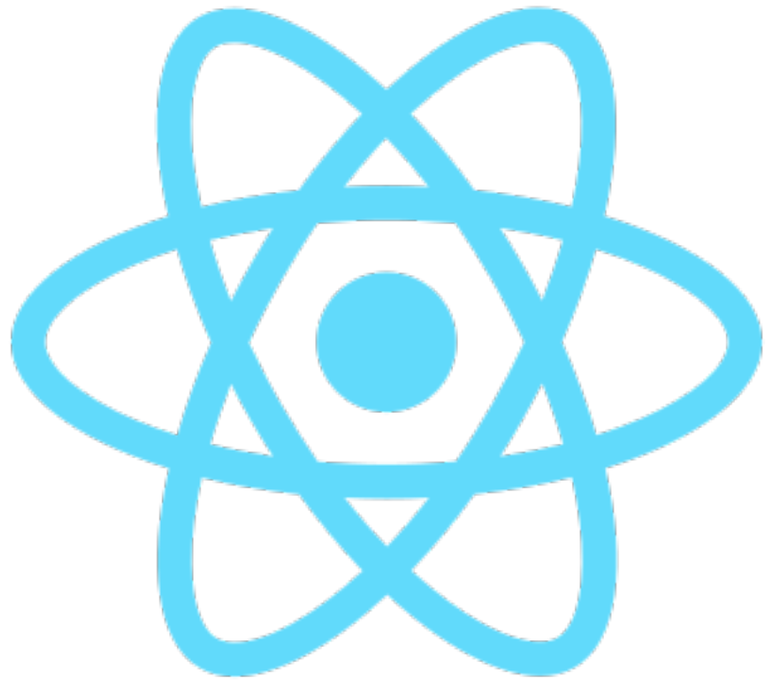
React и Redux

React – JavaScript библиотека для создания UI

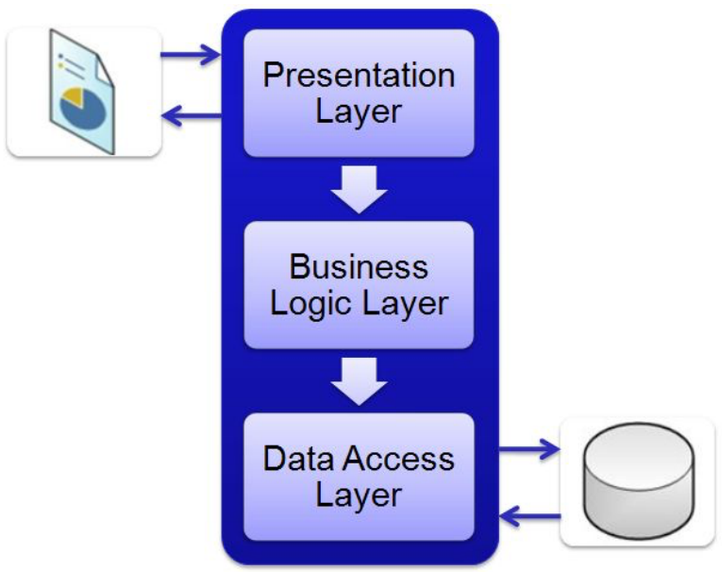
- Open Source
- Создан Facebook
- Новостная лента Facebook (2011)
- Instagram (2012)
- Исходный код открыт в мае 2013 г.
- Используется для создания SPA



Зачем это нужно?

- Декомпозиция
- Переиспользование компонентов
- Делегирование функционала взаимодействия с DOM
- Единый жизненный цикл компонента

Трехуровневая архитектура

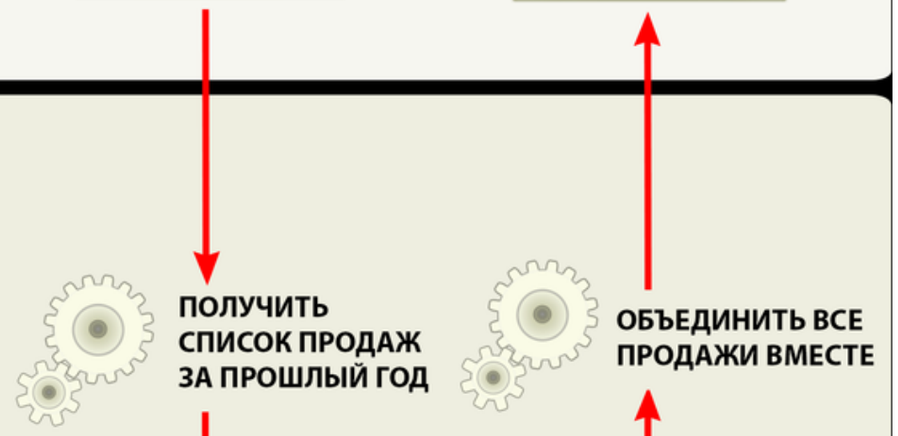


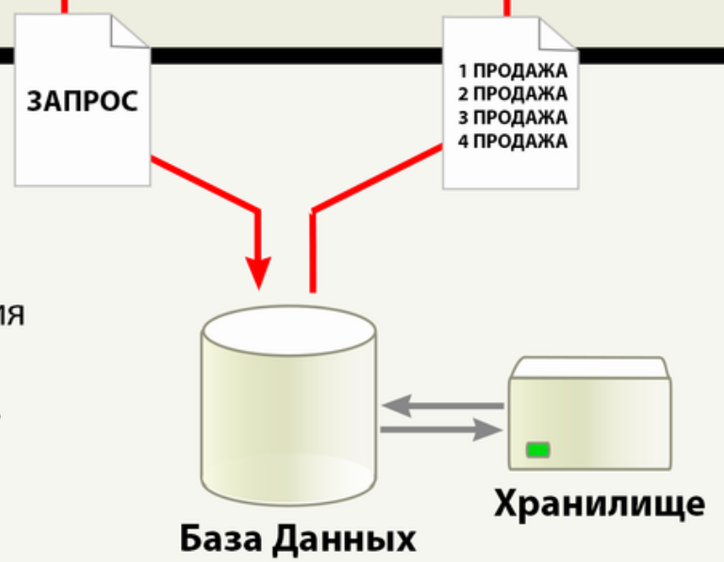
Слой клиента

Самый верхний уровень приложения с интерфейсом пользователя. Главная функция интерфейса представление задач и результатов,понятных пользователю.

Слой логики

Этот слой координирует программу, обрабатывает команды, выполняет логические решения и вычисления, выполняет расчеты. Она также перемещается и обрабатывает данные между двумя окружающими слоями.





Слой данных

Здесь хранится информация и извлекается из базы данных и файловой системы. Информация отправляется в логический слой для обработки и в конечном счете возвращается пользователю.

Single Page Application (SPA)

- Перенос части работы с сервера на клиент
- Единый html документ
- Роутинг с помощью JavaScript посредством HTML5 History API
- Управление состоянием приложения

HTML5 History API

Для чего?

Управление историей браузера посредством JavaScript

Как?

- history: pushState() и replaceState()
- location: href, forward(), back()

- MDN
- Введение в HTML5 History API

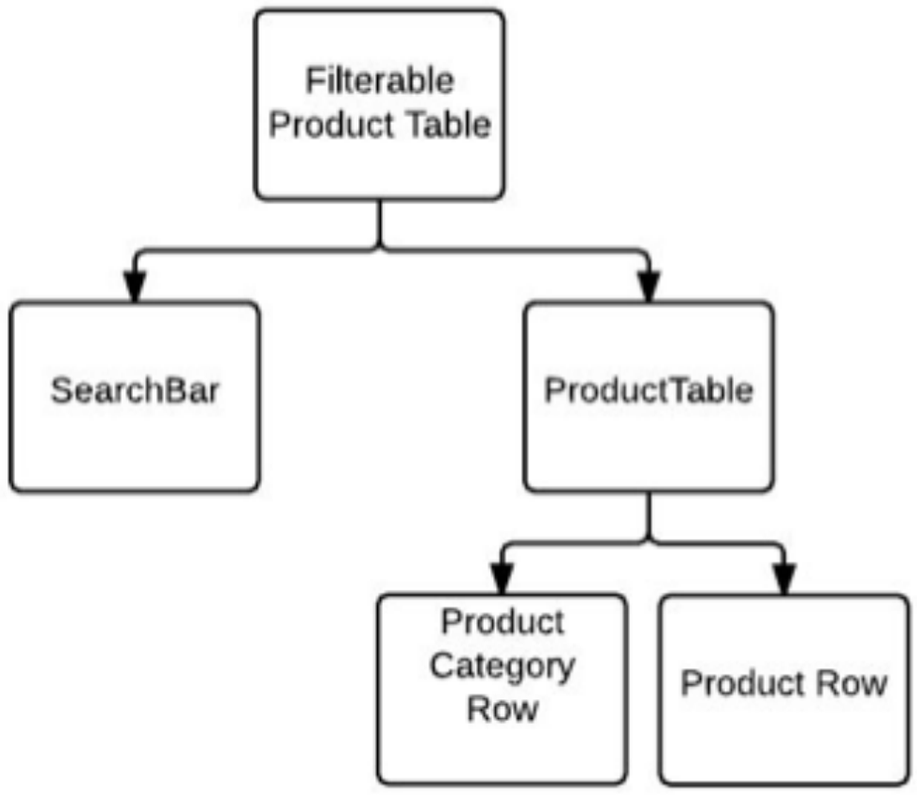
ReactJS: Особенности

- Основан на компонентном подходе
- Декларативность
- One-way data flow
- Virtual DOM

[Введение: знакомство с React]

Компонентный подход в React

Компонентный подход в React (2)



Пример реализации компонента

Объявление пользовательского компонента

```
class Hello extends React.Component {
    render() {
      return React.createElement('div', null,`Hello ${this.props.name}`);
    }
}
```

Отрисовка пользовательского компонента на странице

```
ReactDOM.render(
    React.createElement(Hello, {name: 'Вася'}, null),
   document.getElementById('root'));
);
```

Использование JSX

```
const element = <h1>Hello, {name}!</h1>;
```

- Расширение языка JavaScript
- Синтаксический сахар над React.createElement
- Компилируется в JavaScript с помощью Babel
- Визуально напоминает HTML, язык шаблонов

Знакомство с JSX

Пример реализации компонента (JSX)

Объявление пользовательского компонента

```
class Hello extends React.Component {
    render() {
      return <div> Hello ${this.props.name}</div>;
    }
}
```

Отрисовка пользовательского компонента на странице

```
ReactDOM.render(
    <Hello name="Вася"/>,
    document.getElementById('root'));
);
```

Переиспользование компонентов (JSX)

Объявление компонента

```
class NameForm extends React.Component {
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>Name {this.name}</label>
          <input type="text" value={this.state.value}
            onChange={this.handleChange}
          />

          <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Переиспользование компонентов (JSX)

Подключение компонента NameForm

```
class AnotherOneComponent extends React.Component {
  render() {
    return (
      <div>
        Используем ранее объявленный компонент
        <NameForm />
      </div>
    );
  }
}
```

Встроенные и пользовательские компоненты

```
render() {
  return (
     <div>
        <h1>Header text</h1>
        <ItemsList items={this.state.items}/>
        <form onSubmit={this.handleSubmit}>
          <label>Name {this.name}</label>

           <input type="text" value={this.state.value}
             onChange={this.handleChange}
           />
           <input type="submit" value="Submit" />
        </form>
     </div>
  );
}
```

ItemsList – пользовательский компонент

Нейминг – Название пользовательского компонента с заглавной буквы

div, h1 , form, input – встроенные компоненты

JS выражения в JSX

```
render() {
  let name = 'Константин';
  return (
     <div>
        <h1>Name: {name}</h1>
        <ItemsList items={this.state.items}/>
    </div>
  );
}
```

В фигурных скобках в JSX описываются JavaScript-сущности

JSX – подробности

Условный рендеринг в JSX

Компоненты: классы и функции

Классы

- Наследуются от React.Component
- Можно переопределять методы жизненного цикла
- Обладают внутренним состоянием (state)
- Принимают props от родительских компонентов

Функции

- «Чистые» (pure) функции от props
- Отсутствует внутреннее состояние
- Жизненный цикл есть, но привязки к нему нет

Компоненты и свойства

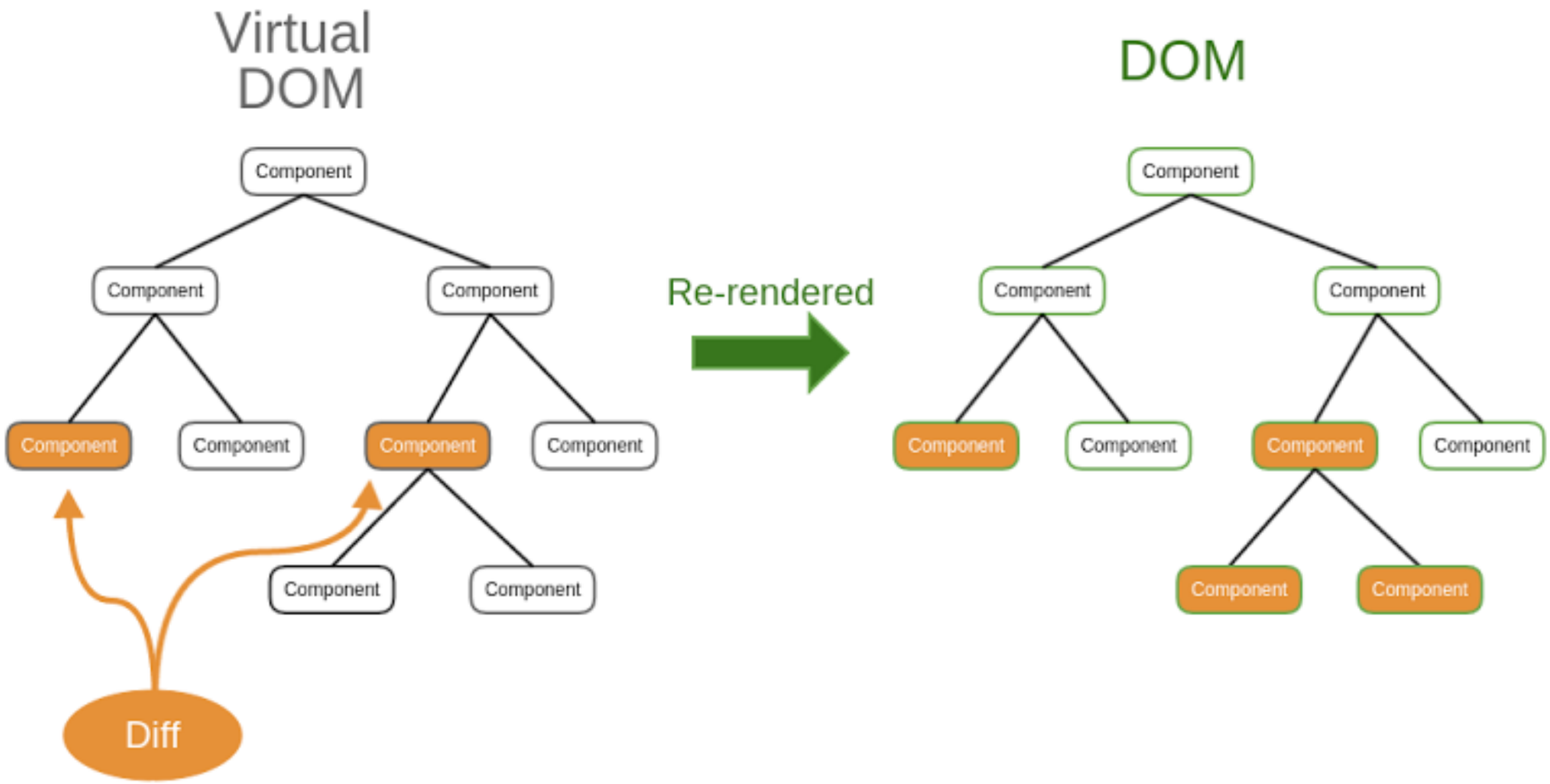
Состояние и жизненный цикл

VirtualDOM

Техника и набор библиотек и алгоритмов, которые позволяют улучшить производительность на клиентской стороне, избегая прямой работы с DOM путем работы с легким JavaScript- объектом, имитирующем DOM-дерево.

[Что такое Virtual DOM?]

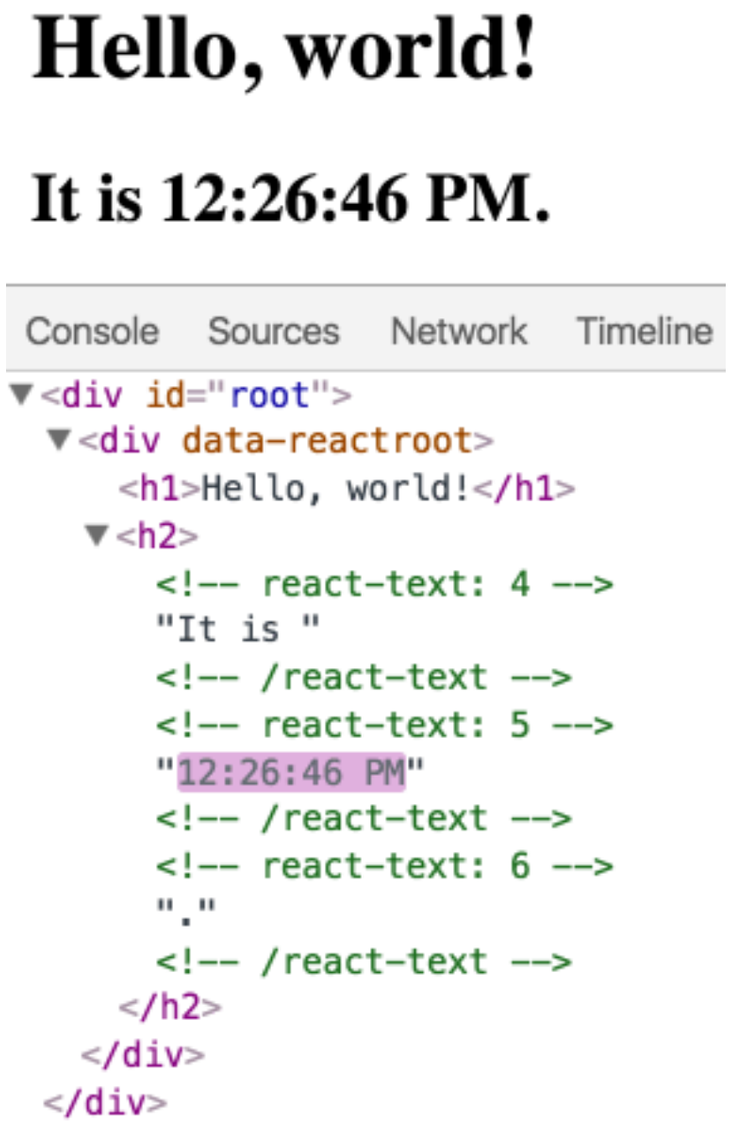
VirtualDOM



Virtual DOM представляет из себя дерево абстрактных компонентов, как их отобразить — задача конкретных рендереров (React DOM, React Fiber, React Native, etc).

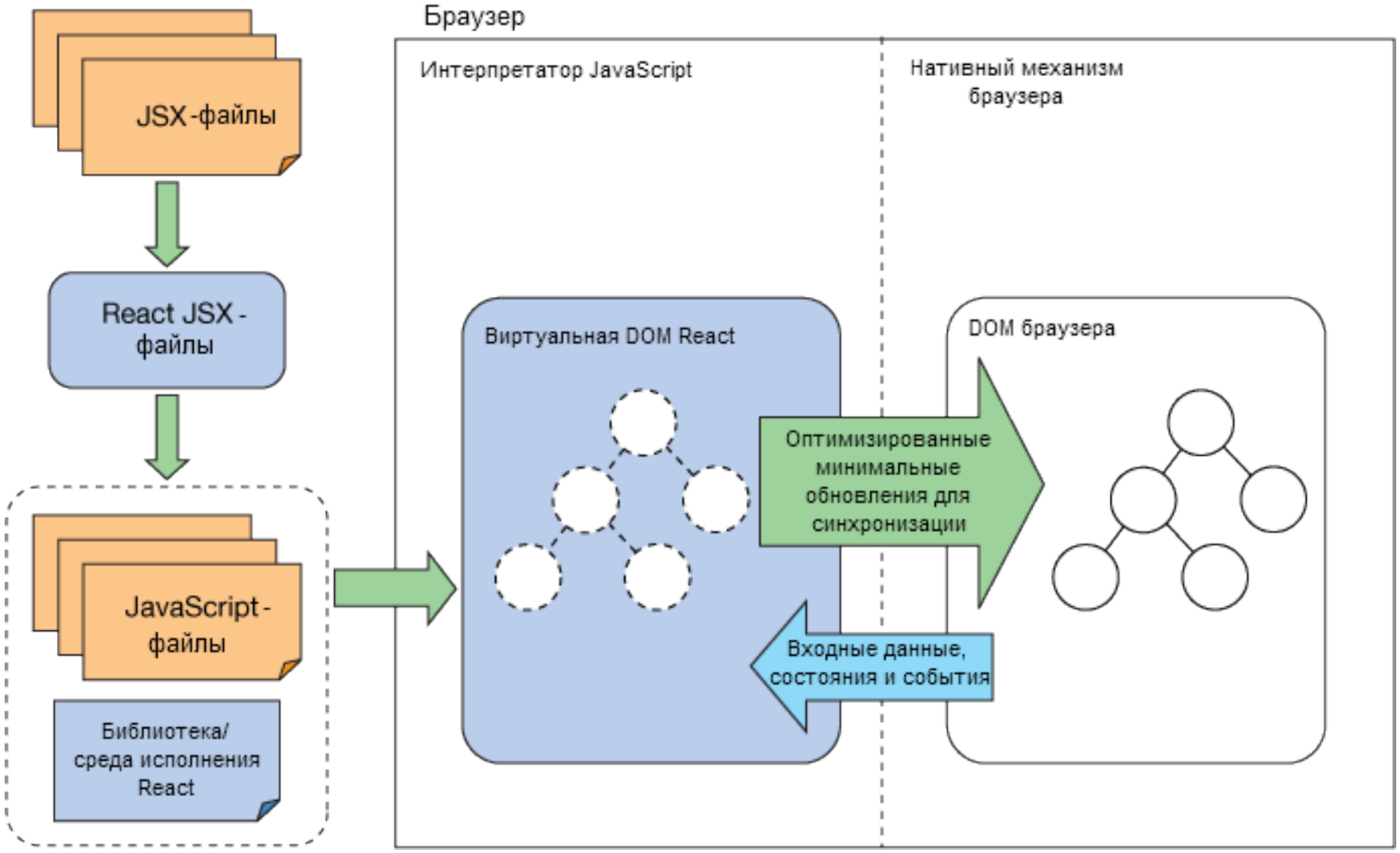
[Виртуальный DOM в React]

Пример



[React обновляет только то, что необходимо]

Общая схема работы



State, props

- Каждый компонент обладает состоянием **state** и свойствами **props**

- Свойства служат для передачи данных между родительским компонентом и дочерним, напоминают атрибуты тегов в XML и HTML

- Состояние является внутренним, потому недоступно другим компонентам (но его можно пробросить через **props**)

- Изменения **state** и **props** приводят к перерисовке компонента

State, props

```
class MyComponent extends React.Component {
  constructor(props) {
     super(props);
     this.state = {myName: 'Иван'};
  }
  render() {
    return (
      <div>
         Используем ранее объявленный компонент
         <NameForm name={this.state.myName}/>
      </div>
    );
  }
}
```

```
class NameForm extends React.Component {
  render() {
    return (
       <form onSubmit={this.handleSubmit}>
           <label>Name {this.props.name}</label>

          <input type="text"
             value={this.state.value}
             onChange={this.handleChange}
           />

          <input type="submit" value="Submit" />
       </form>
    );
  }
}
```

State

- Классы могут обладать своими полями, но это не имеет никакого отношения к state (с точки зрения React)
- Единственный правильный способ изменения состояния — вызов метода `setState(newState)`
- `setState` не меняет состояние целиком, а объединяет старое и новое

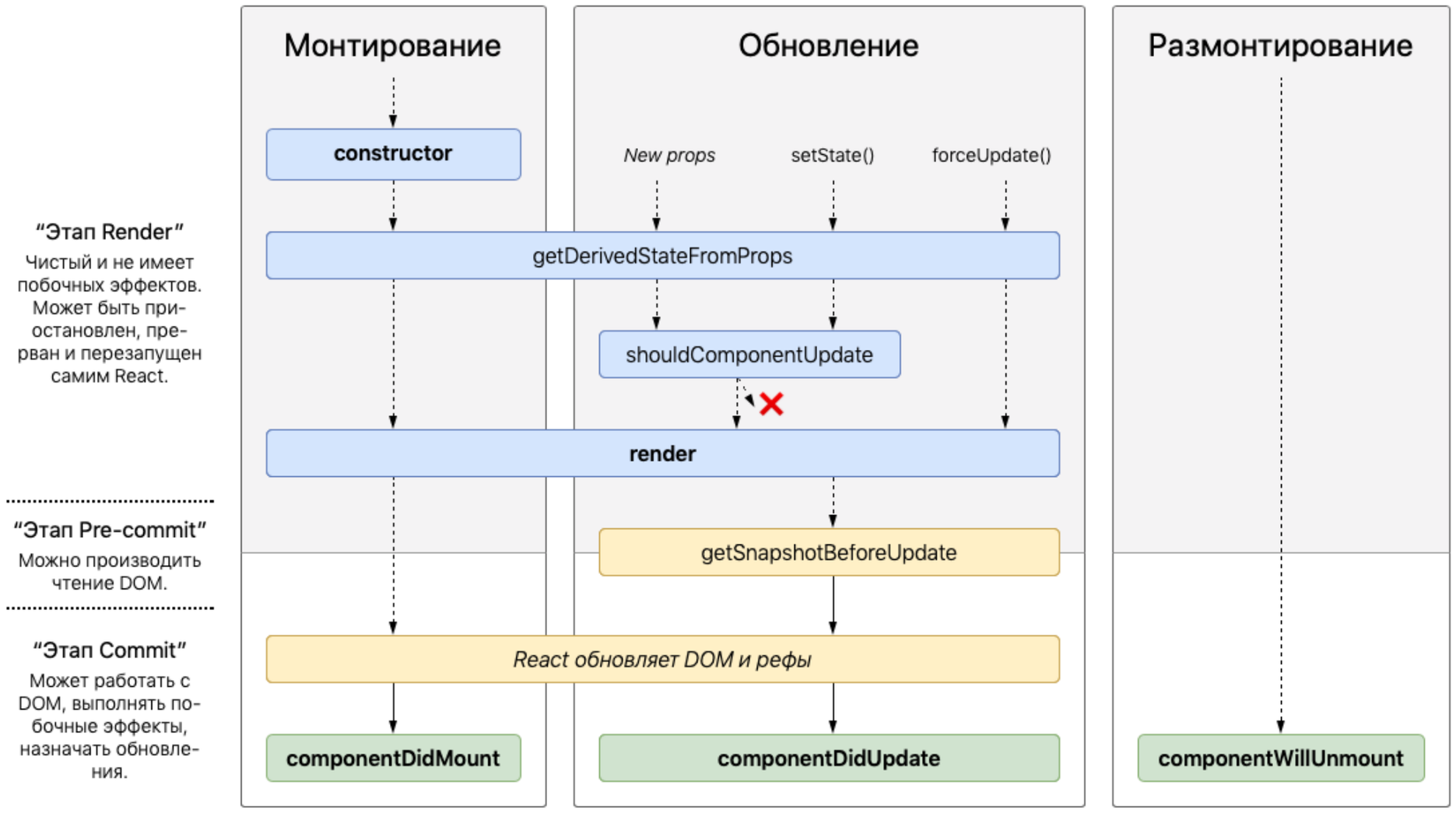
Обработка событий

- Обработка событий есть в React
- В обработчик событий передается не нативное событие DOM’а, а обертка `SyntheticEvent`
- Обработчики событий именуются в camelCase

```
<button type='submit'
    onMouseOver={this.onMouseEnter}
    onClick={this.props.onClick}
    onMouseDown={(event) => alert('Mouse down!’)}
/>
```

Обработка событий

Жизненный цикл компонента React



[В интерактивном режиме и со ссылками на описание состояний]

[Описание состояний жизненного цикла]

[Жизненный цикл компонента]

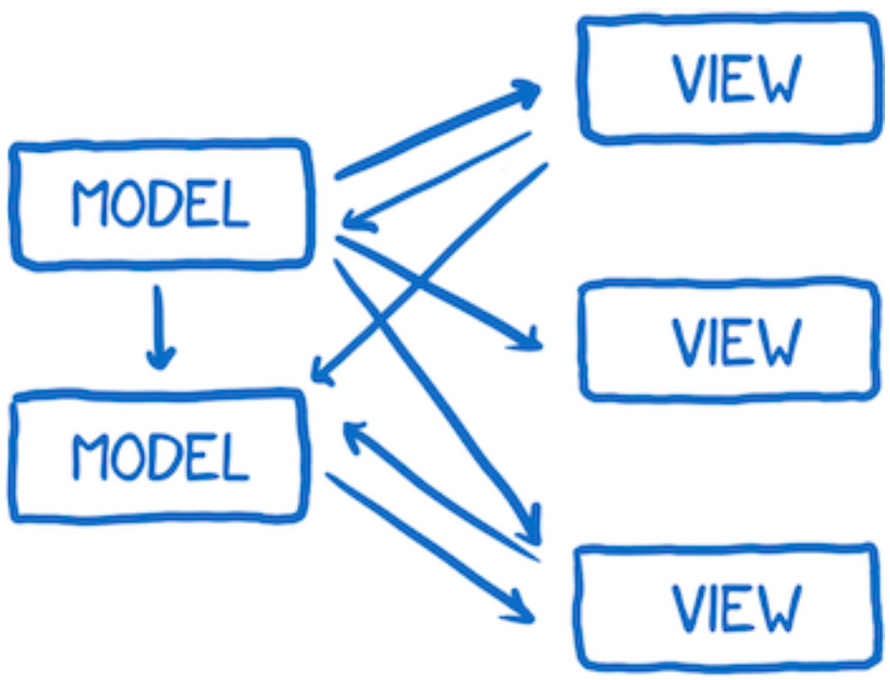
Полезные ссылки

- Философия React на примере создания компонента
- Способы передачи данных между компонентами в React

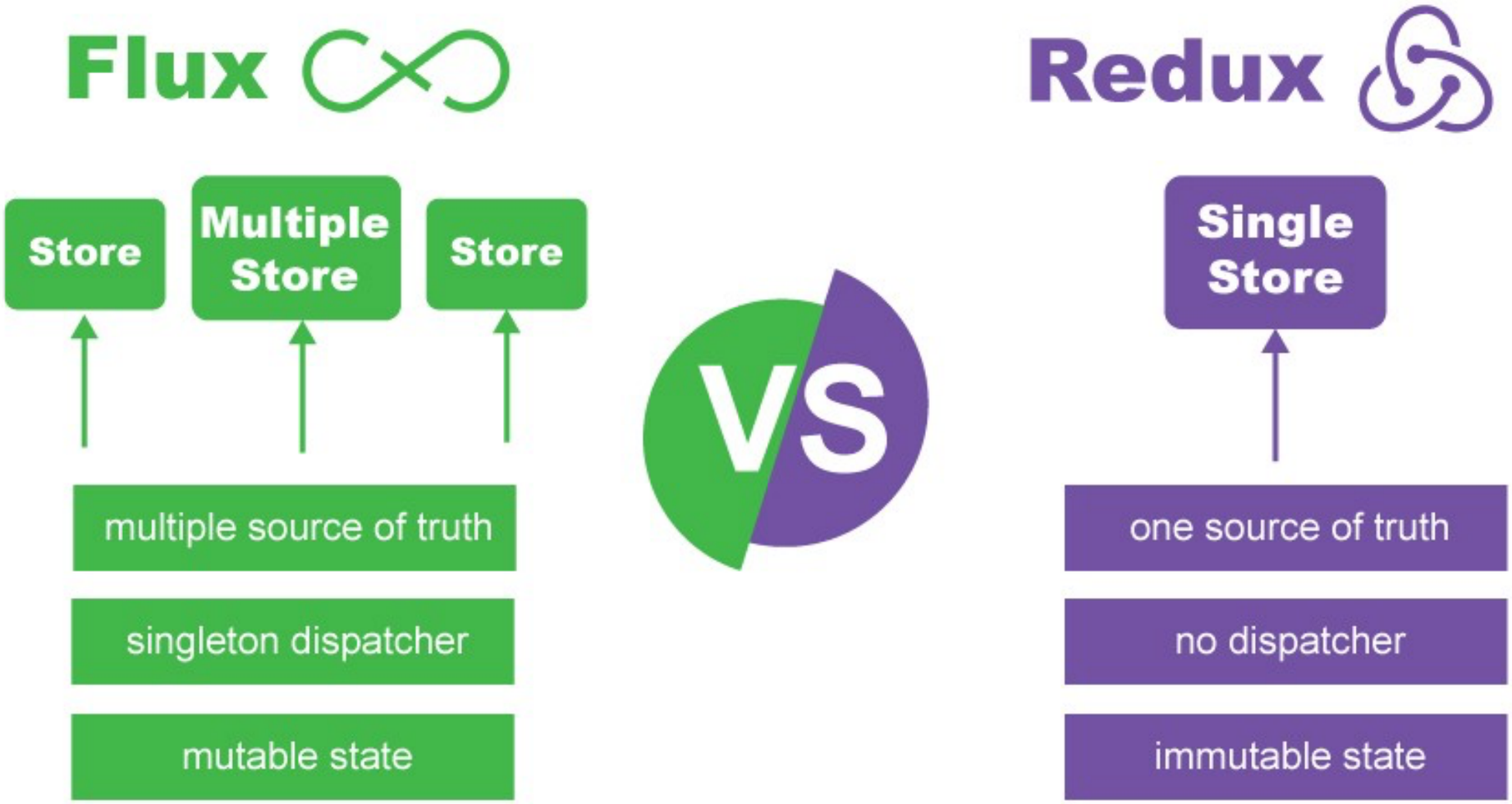
Состояние приложения: проблема

Проблема: многие представления меняют многие модели.
Многие модели отражаются во многих других представлениях.

Как синхронизировать состояние?



Flux и Redux



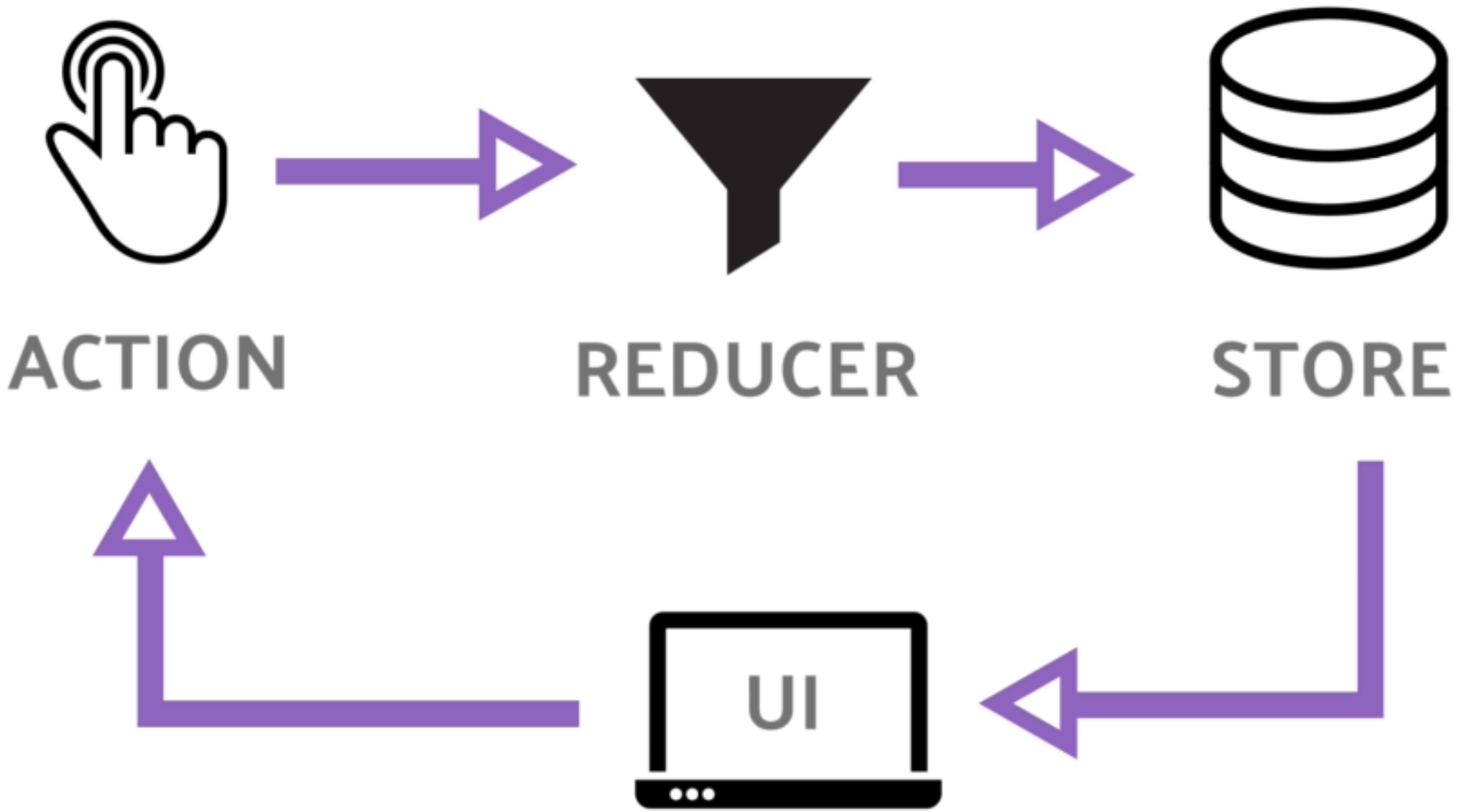
[Flux в картинках]

[Flux и Redux]

[Flux vs. Redux: A Comparison]

Redux

Redux – менеджер состояния приложения, единственный источник истины о данных приложения



Redux

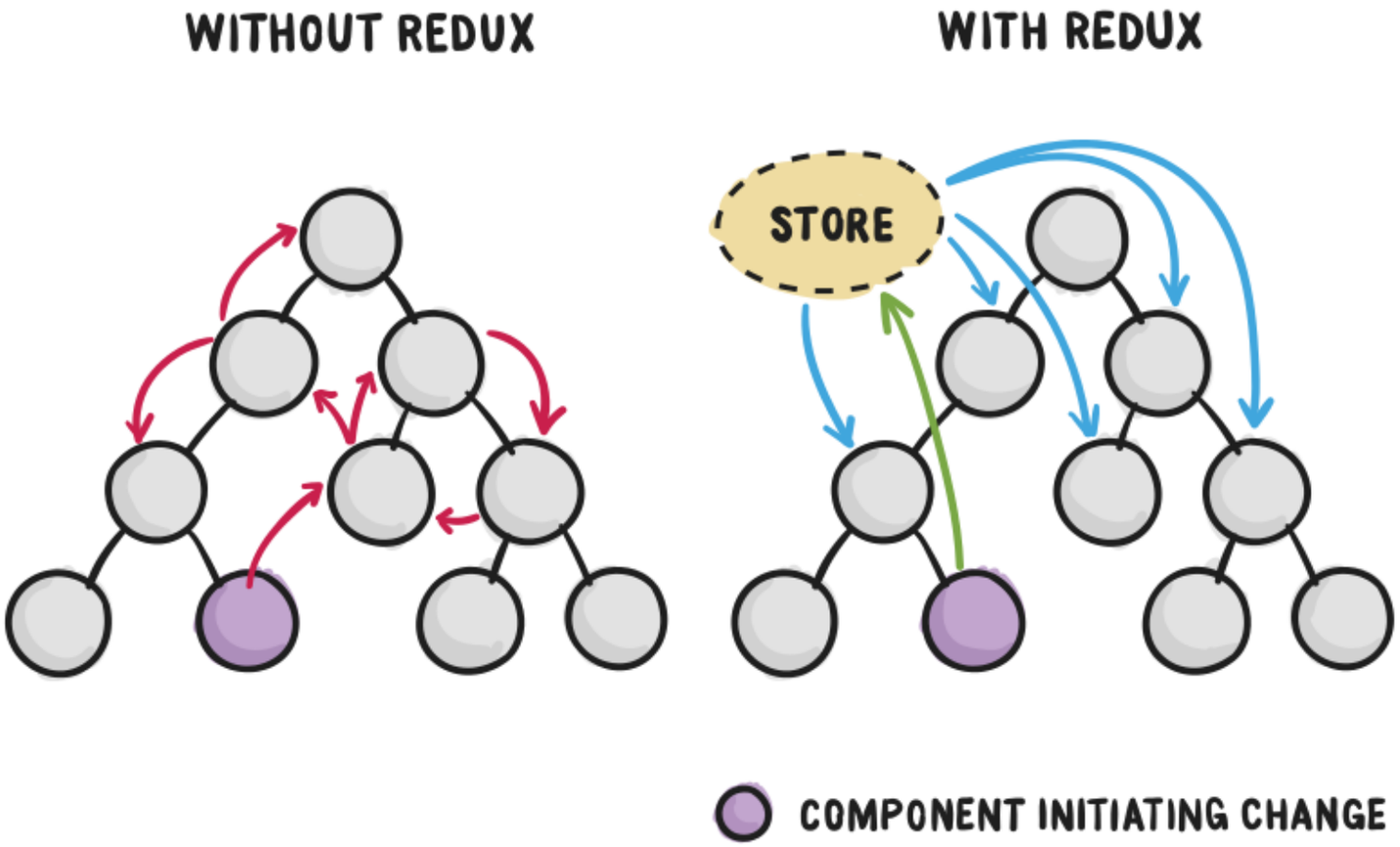
Преимущества

- Возможность сохранения истории всех изменений
- Надежность и простота отладки
- Единственный «источник правды» о состоянии приложения – все компоненты отображают одно и то же состояние
- Компоненты не имеют своего глобального состояния – легче покрыть тест-кейсами

Недостатки

- Даже для простых вещей нужно писать больше кода
- …

Store

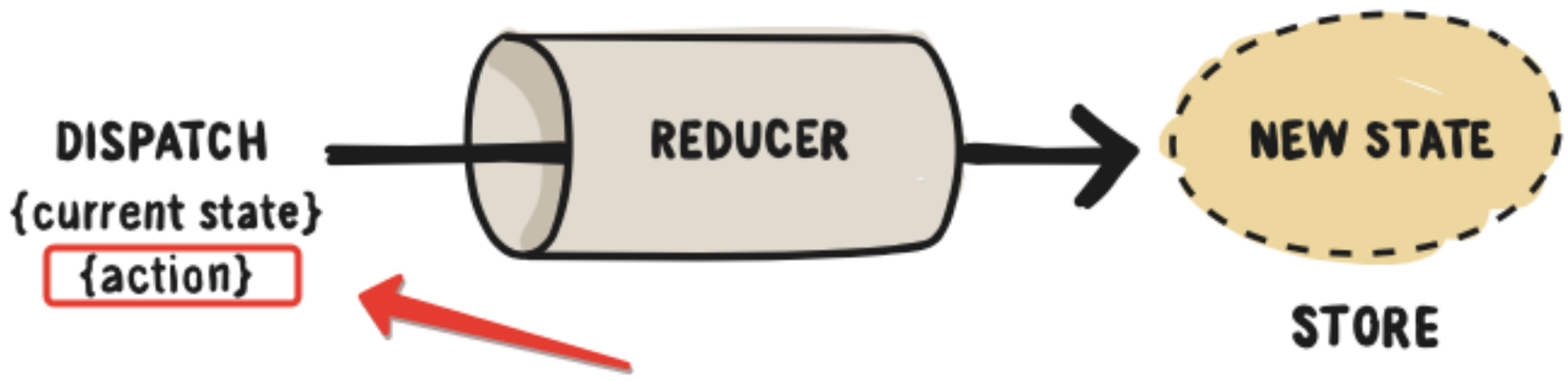


Redux предоставляет единое хранилище состояния.

Ключевой его особенностью является ***неизменяемость*** — все изменения в него вносятся посредством создания нового объекта состояния.

Store

Action



- Событие, которое должно привести к изменению состояния – созданию нового объекта состояния
- Например – клик по кнопке, получение данных с сервера

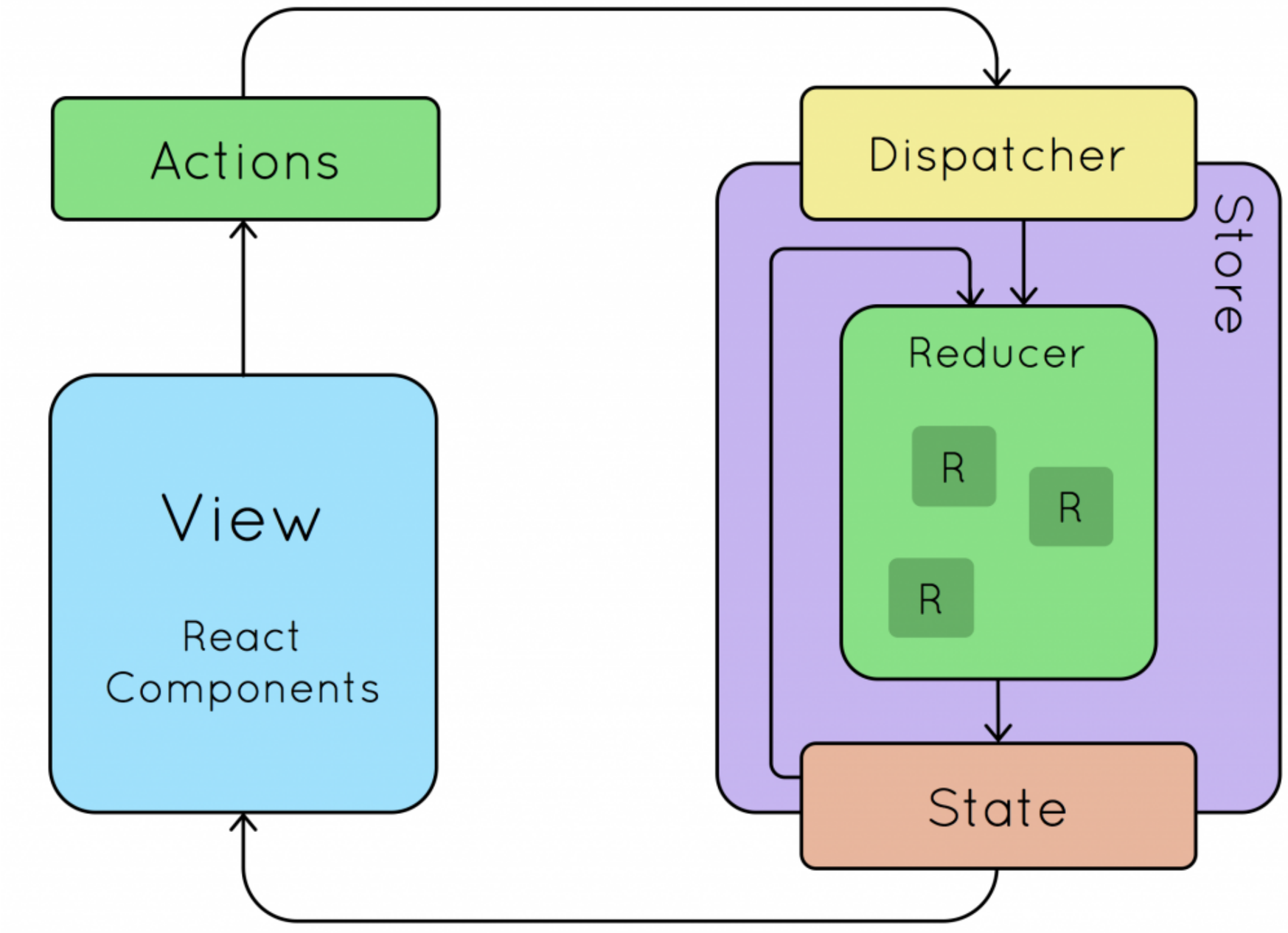
Actions

Reducer

- Чистая функция: выходное значение зависит только от входных параметров
- Возвращает новое состояние, текущее состояние остается неизменным

Reducers

Redux flow


Actions
Dispatcher
Store
Reducer
R
R
R
View
React
Components
State

Создание Store

```
createStore(reducer, preloadedState, enhancer)
```

- `reducer` – функция, создающая новый state на основе текущего state и action'а
- `preloadedState` – начальный state приложения
- `enhancer` – цепочка middleware

createStore()

Reducer

```
export default function user(state, action) {
    switch (action.type) {
        case USER_INFO_REQUEST: {
          return Object.assign({}, state, {fetchingInfo: true});
        }

        case USER_INFO_FAIL: {
            return Object.assign({}, state, {
                fetchingInfo: false,
                fetchInfoError: action.error
            });
        }
    }
}
```

Использование нескольких reducer’ов

`export default combineReducers(reducers)`

reducers/todos.js

```
export default function todos(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      return state.concat([action.text]);
    default:
      return state;
  }
}
```

reducers/counter.js

```
export default function counter(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1;
    case 'DECREMENT':
      return state - 1;
    default:
      return state;
  }
}
```

reducers/index.js

```
import { combineReducers } from 'redux';
import todos from './todos';
import counter from './counter';

export default combineReducers({
  todos,
  counter
})
```

App.js

```
import { createStore } from 'redux';
import reducer from './reducers/index';

const store = createStore(reducer);
```

Action

```
export const addTodo = (name, date, comment) => ({
    type: types.ADD_TODO,
    payload: {
        name,
        date,
        comment
    }
})
```

dispatch

```
import {addTodo} from '../actions/todo';
import dispatch from 'redux';
func() {
   dispatch(addTodo);
}
```

Middleware

```
applyMiddleware(middlewares)
```

Предоставляет стороннюю точку расширения между отправкой действия и моментом, когда это действие достигает reducer’a

```
middleware = (store) => (next) => (action) => {
    ...
}
```

[Пример middleware]

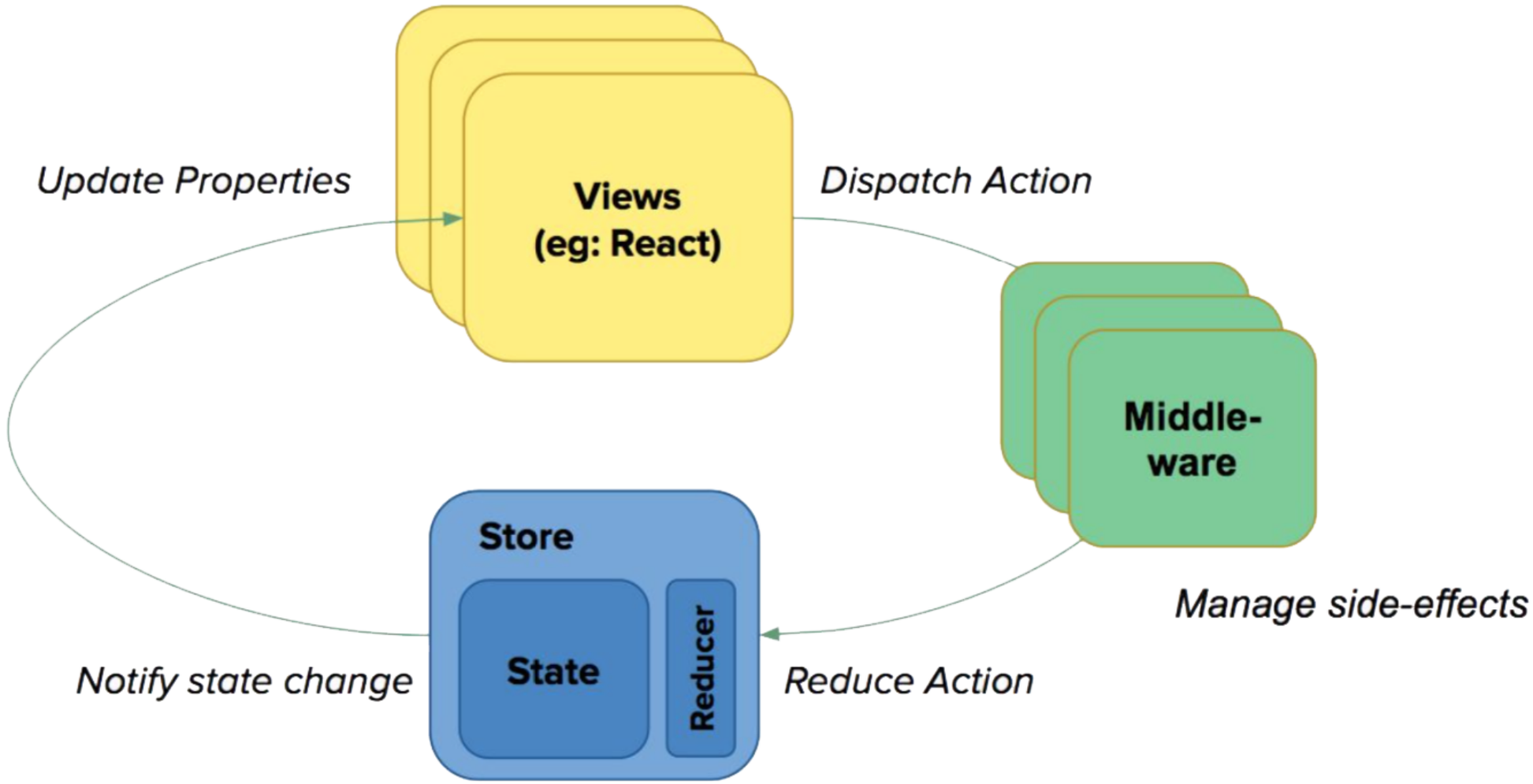
Примеры использования:

- логирование
- сообщения об ошибках
- взаимодействие с асинхронным API
- роутинг

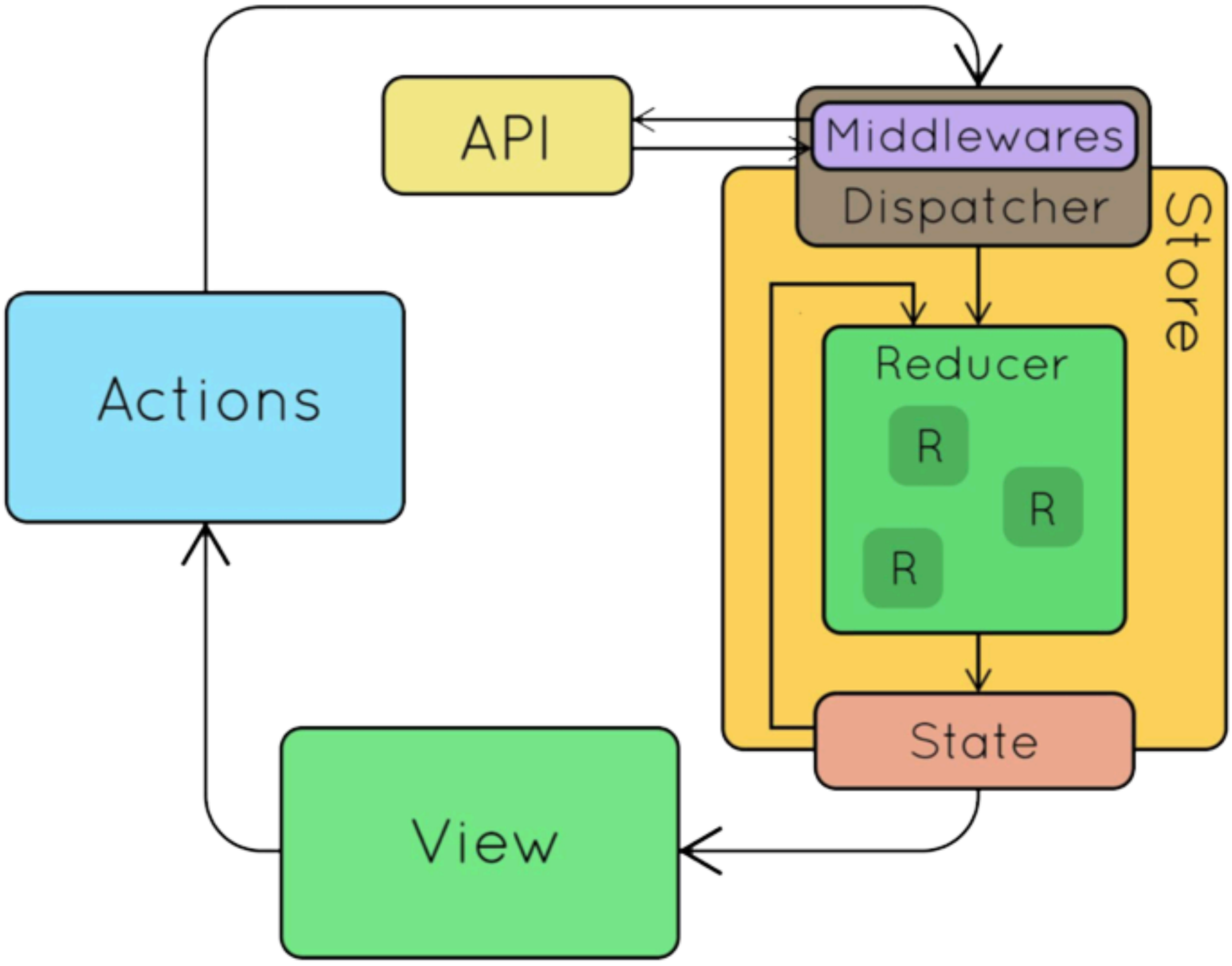
[Еще больше примеров использования middleware]

[applyMiddleware()]

Middleware


Update Properties
Views
(eg: React)
Dispatch Action
Middle-
ware
Manage side-effects
Store
State
Reducer
Notify state change
Reduce Action

Middleware в Redux



Полезные ссылки

Redux: шаг за шагом

🔥 Погружение в React: Redux

🔥 React Redux Tutorial на русском языке

React Redux Quick Start

create-react-app

Библиотеки и технологии

Вместе с React используются:

- npm
- babel
- webpack
- redux
- react-router

NPM

Пакетный менеджер для NodeJS.

Используется для поставки зависимостей приложения

Как подключались библиотеки раньше:

Найти и скачать в интернете нужную библиотеку

Положить в проект

Подключить тегом script в html файле

Подключить тегом script в html файле через CDN-сервис

А теперь:

npm

NPM

Устанавливаемые зависимости попадают в директорию node_modules

Установить локально

```
npm i <package>
```

Добавить в package.json

```
npm i --save <package>
```

Установить глобально

```
npm i -g <package>
```

Запустить скрипт start

```
npm run start
```

```
package.json
{
        "name": "my_package",
        "version": "1.0.0",
        "engine": {
             "node": ">=6",
             "npm": ">=3.10.1"
        },
        "dependencies": {
             "react": "^15.3.1",
             "react-router": "^2.8.1",
        },
        "devDependencies" : {
             babel-core": "^6.13.2",
             "webpack": "^1.13.1",
        },
        "scripts": {
             "start": "npm run build && npm run watch",
             "build": "webpack —config webpack.config.js",
             "watch": "webpack —watch"
        }
}
```

[Шпаргалка по npm]

[npm Tutorial For Non Node.js Developers]

Babel

Транспайлер (компилятор),
преобразующий один язык в другой

ES2015, JSX, Typescript, Coffeescript,
Clojurescript, …

Например, можно писать код на ES6 и преобразовывать его в ES5,
чтобы оно работало в IE или старых браузерах

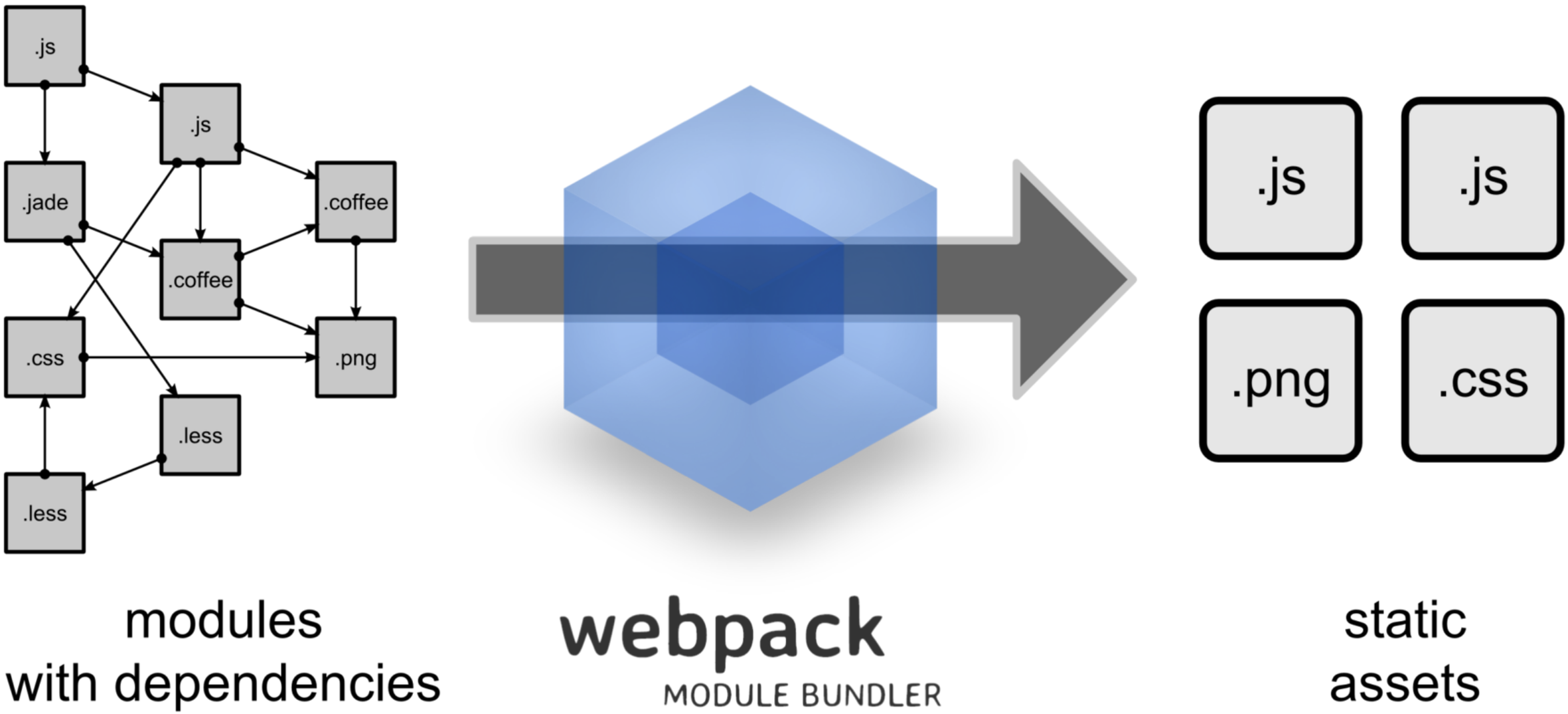
BabelJS

Описание Babel

Babel Demo Sandbox

Настройка окружения – Babel

Webpack


.js
.js
.jade
.coffee
.coffee
.css
.png
.less
.less
.js
.js
.png
.css
modules
with dependencies
webpack
MODULE BUNDLER
static
assets

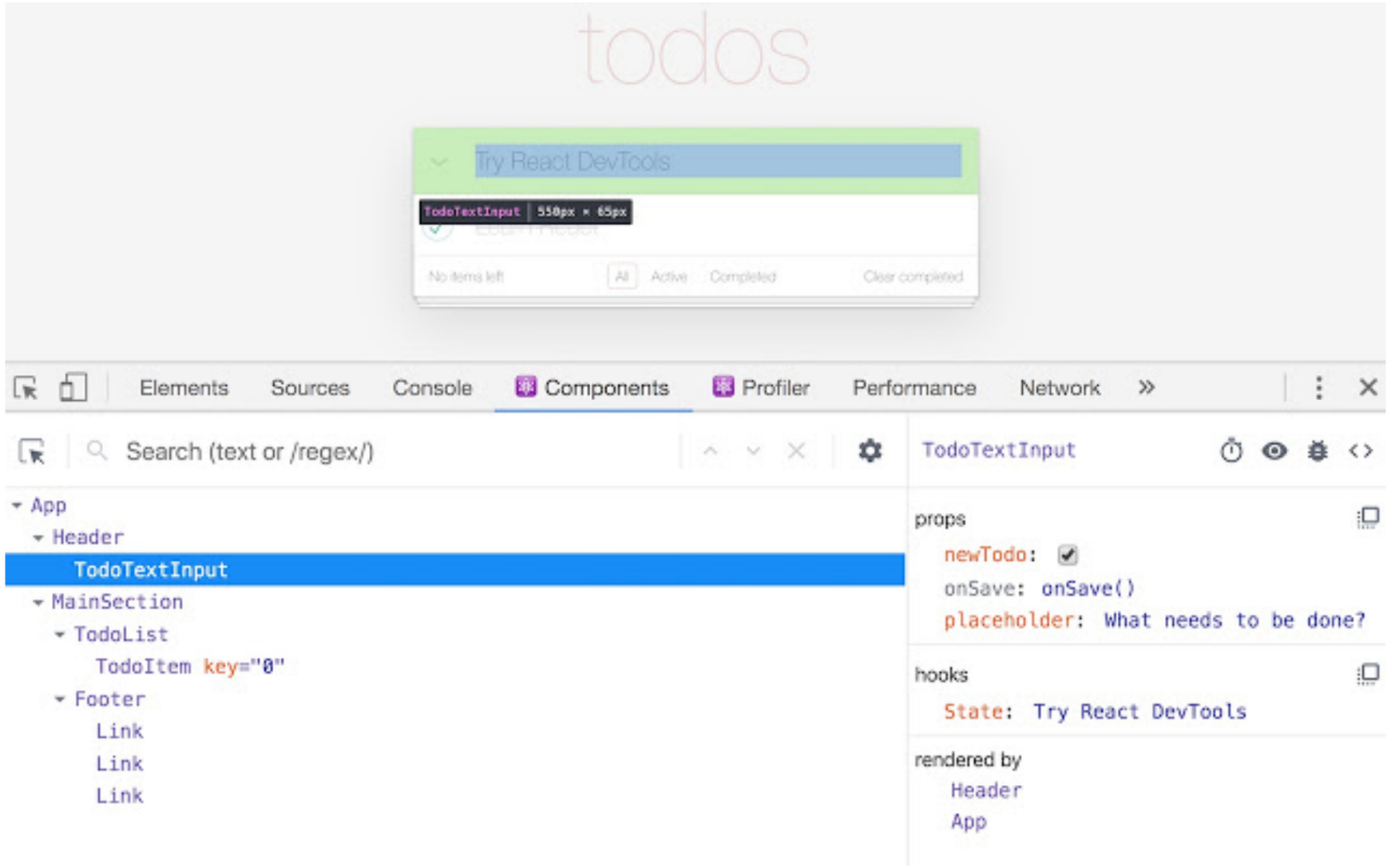
Webpack

- Сборщик модулей
- Преобразует множество модулей (бибиотек) в единый bundle
- Конфигурация – в `webpack.config.js`
- Модули и плагины – точки расширения для управление сборкой и дополнительной обработкой модулей: минификация, css-препроцессоры, babel, …
- hot-reloading через webpack-dev-server

[Webpack: 7 бед — один ответ]

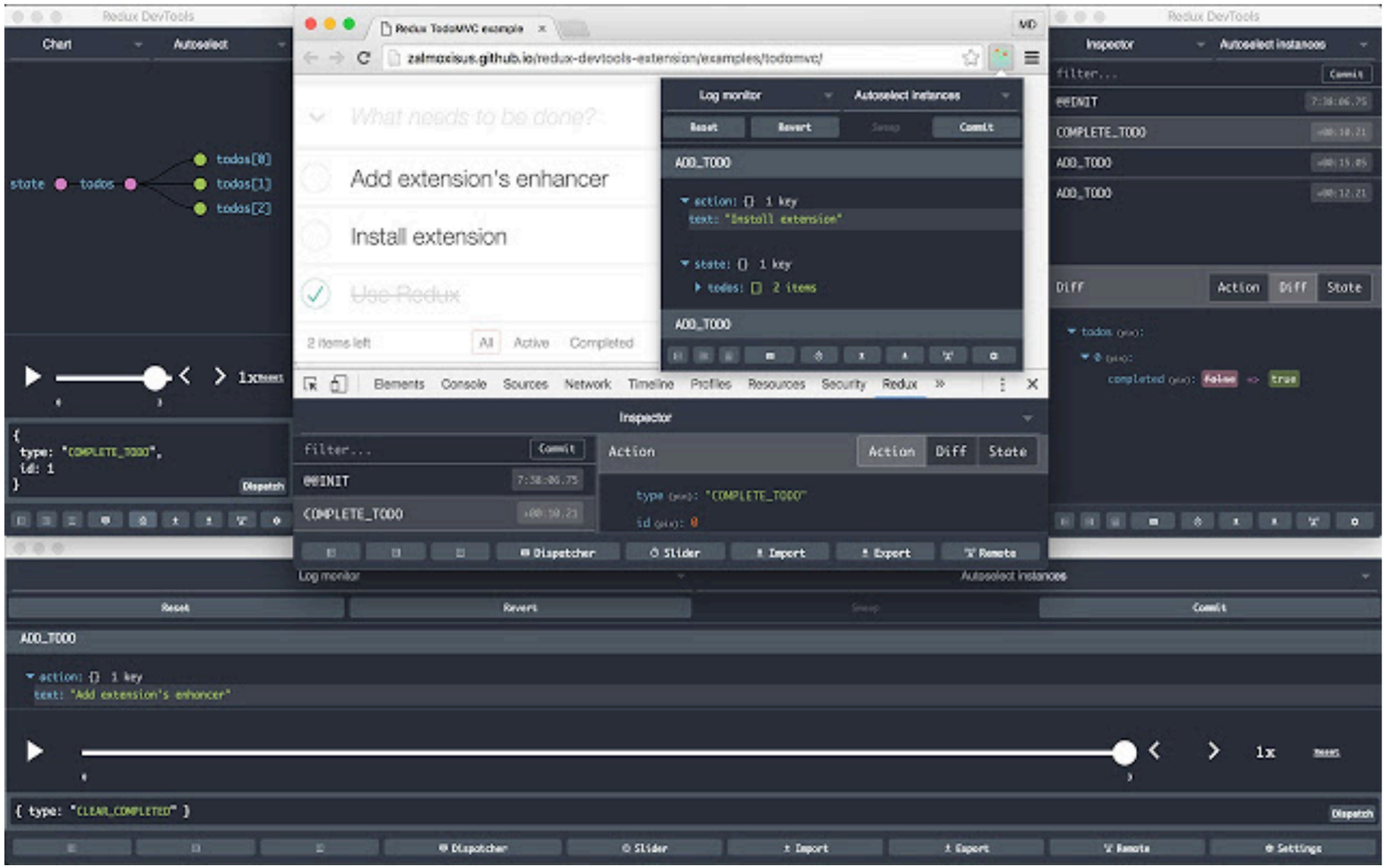
[Основные концепции Webpack]

Developer tools Extensions



[React Developer Tools Chrome Extension]

Developer tools Extensions



[Redux Developer Tools Chrome Extension]

Лабораторная работа 2

Работа публикуется на Github.

Создать приложение с использованием библиотеки React.
При первой загрузке страницы происходит запрос пользователя на получение данных о геолокации с использованием HTML5 Geolocation API. Если пользователь соглашается предоставить данные о геолокации – получаем из внешнего API данные о погоде. Если нет – запрашиваем информацию для города по умолчанию (город по умолчанию можно выбрать самостоятельно). Информация о городе, данные о погоде (температура, ветер, давление, влажность), иконка погоды, координаты отрисовываются на странице в соответствии с макетом. Иконка и все необходимые данные есть в API https://openweathermap.org.
В интерфейсе также есть кнопка с повторным запросом геолокации пользователя.

У пользователя есть возможность добавления и удаления городов в избранное. Информация о погоде отображается для всех городов из избранного в соответствии с макетом. Избранное сохраняется в LocalStorage.
Пока происходит загрузка данных по конкретному городу/локации – показываем loader и/или сообщение об ожидании загрузки данных.

Работа с глобальным состоянием приложения (например, список избранных городов) реализуется с помощью Redux.
Локальное состояние компонента (например, состояние ожидания загрузки данных) – через локальный state компонента.

Лабораторная работа 2