# Основы React

## HotKeys

* Ctrl+F+K – автоформатирование
* Ctrl+Shift+F – поиск
* Ctrl+K+C – комментировать в JSX
* Alt+Ctrl – несколько курсоров плюс прыжки по строкам
* Ctrl+ Shift+Стрелка – выделить слово
* Написать div + Tab – быстрое создание тега
* Shift+Tab – уменьшение отступа

## Запуск первого React App

Делаем установку с [https://github.com/facebook/create-react-app через *node.js*](https://github.com/facebook/create-react-app%20через%20node.js) *command prompt.*

Переходим с помощью вышеупомянутой консольной утилиты в папку с будущим проектом и прописываем следующие вещи:

npx create-react-app <my-app-name>

cd my-app

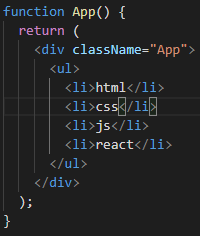
npm start

* npm install – загрузит папку node\_modules. Грузит модули на основании package.json
* \*npx в отличии от npm не только установит модули, но и подготовит их.
* \*вместо my-app можно свое название проекта
* package.json сообщает node.js, что папка является проектом в терминологии node.js

В файле App.js в функции App() должен быть только один корневой элемент

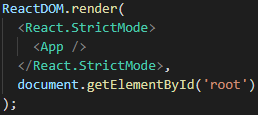
## Компонента

Компонента – это функция (если функция – то это stateless компонента), возвращающая разметку JSX. Пример:

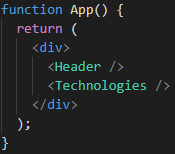
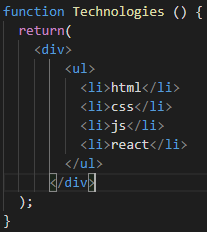
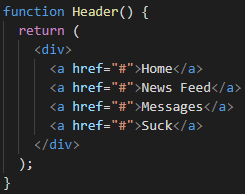


Соответственно появится тег <App/>. То есть также можно сказать, что компонента - это еще и тег.

Этот тег мы вставляем в файле *index.js*:



Одни такой тег можно вставить в другой тег, вот пример в App.js:

## Импорт/Экспорт

В реальном проекте каждая компонента (Technologies, Header), будет лежать в отдельном файле.

* ./ - относительно текущего местоположения
* ./.. – на уровень выше относительно текущего местоположения
* Импорт CSS отличается от импорта JS модулей: import './App.css';
* Модули из папки *node\_modules* импортируются по-другому:

import React from 'react'; - не нужно ставить никаких точек, нужная папка сама найдется

* Чтобы использовать синтаксис JSX нужно сделать импорт:

import React from 'react';

* Импорт картинки: import logo from './logo.png'; Вставка:

<img src={logo} alt='d' className="logo" />

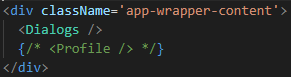
## Организация CSS

Для каждой компоненты нужно создать свой css файл. И подключить уже внутри каждой компоненты свой css файл.

* Чтобы css был изолирован в своих компонентах нужно назвать их так: name.module.css.
* Импорт такого css: import classes from './name.module.css'. classes – это объект, ключами которого являются названия стилей, а значения – сгенерированные для них автоматические названия:

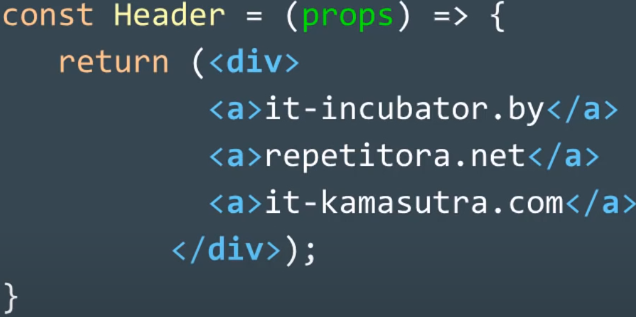


* Вот так применяются классы: <div className={classes.sideBar}>
* Два класса можно применить с помощью форматной строки: {`${s.class1} ${s.class2}`}
* Компонент и CSS группируем в одну папку для удобства
* Если у SPA приложения есть странички, которые меняются, но свойства контейнера у них одинаковые, их можно положить в тег div и вынести для него общие css свойства:



## Props

Пропсы – это параметр функции-компонента, которые содержит некоторые данные для нее:



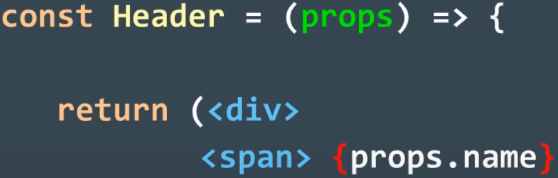
* В качестве пропса всегда приходит какой-то объект. Когда вызываем так: <Header /> - то в пропсе будет пустой объект.
* Чтобы передать объект с параметрами используются атрибуты компоненты:





**Каждый** атрибут со значением становится парой ключ-значение объекта-параметра.

Использование:



**Все атрибуты тега-компоненты превращаются в Пробсы!**

## Маршрутизация: route, browser-router

Система роутинга отслеживает адрес в адресной строке и будет реагировать на изменения в ней.

* Чтобы настроить эту систему нужно установить соответствующий модуль:

npm install --save react-router-dom

save – внесет запись в package.json информацию об установленном нами модуле и после этого при *npm install* этот модуль будет автоматически загружаться в папку *node\_modules.* Сейчас уже автоматом и без save работает.

### <Route/> - компонента

* Не забыть импортировать модули: import { BrowserRouter, Route } from 'react-router-dom';
* Все приложение (например внутри App.js) нужно обернуть в тег-компоненту <BrowserRouter >
* Пути настраиваются с помощью тега-компоненты <Route /> Пример:

Два способа:

1. <Route path='/dialogs' render={ () => <Dialogs />} />
2. <Route path='/profile' component={Profile} />

* Чтобы указать отрисовку компоненты, только в том случае, если url **точно** соответствует указанному, нужно добавить атрибут *exact*:

<Route ***exact*** path='/dialogs' component={Dialogs} /> То есть url вида */dialogs/something* теперь будет проигнорирован и отрисовка не произойдет.

### <NavLink /> - компонента

* import { NavLink } from 'react-router-dom';
* <Navlink><Navlink /> используется вместо тега <a></a> чтобы менять URL и не перезагружать страницу снова, что по умолчанию делает <a>. Пример:

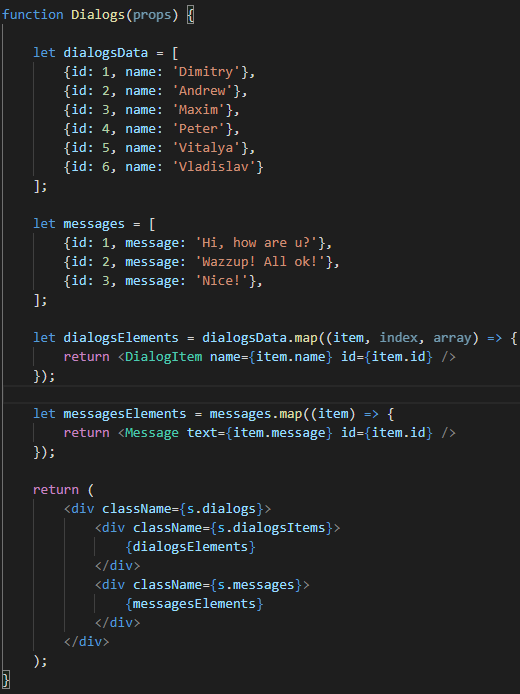
<NavLink to='/profile'>Profile</NavLink>

* Атрибут activeClassName={obj.class} указывает какой класс применить для активной ссылки. По умолчанию ссылке ставится активный класс class=’active’

## UI - BLL, выносим данные в сторону

* BLL (Redux) – это некая data, выносим её отдельно от UI.
* UI – это компоненты (React).
* Данные в компоненту могут прийти в компонент через пробсы.

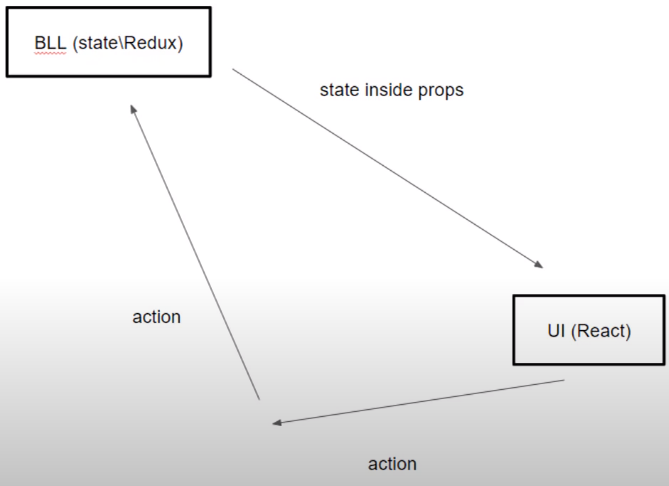
Если в JSX разметке находится массив компонентов, то эти компоненты вынуться из массива и отрисуются друг за другом. Массив компонентов обычно получают из массива объектов с помощью **map**. Пример:



### Прокидка через props

Все данные, такие как dialogsData выше, на самом деле должны приходить извне через пропсы. Если данные, которые пришли извне где-то изменились на уровне BLL, то UI должен перерисоваться. Также функции, которые делают изменения в BLL, тоже нужно прокинуть через props и вызвать в UI. При клике в UI – мы сначала делаем изменения в BLL, а только потом перерисовываем UI. Перерисовка UI при этом запускается из BLL

### FLUX-круговорот



Redux реализовывает FLUX архитектуру.

## VirtualDOM: onClick, ref

### Обработчики событий

* Перехватить событие можно так:



* Обращение к какому-либо html элементу из jsx разметки:

1. нужно дать элементу Реф (**ref**):



1. И теперь, мы можем создать ссылку на этот элемент: 
2. Свойство current ссылается на **нативный** HTML элемент (real DOM):



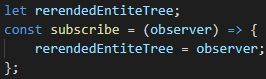
1. В React <textaria> пишется вот так:



Даже если попытаемся ввести символ, то *свойство value* мгновенно вернет текст, который был в этом свойстве, но в обработчике onChange мы можем отловить этот символ, до того, как он затрется.

## Паттерн Observer (callback, subscribe, observer)

В файле *state.js* определена переменная let rerendedEntiteTree. И функция subscribe:





В index.js мы импортируем subscribe и вызываем ее, передав ей в качестве аргумента некую функцию, которую хотим послать в мир state.js:





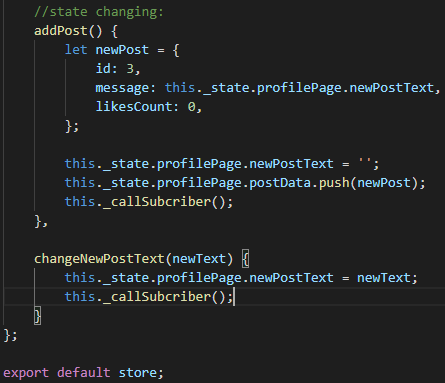
В итоге, в функции subscribe, переменная rerendedEntiteTree получает ссылку на функцию из файла index.js, и мы можем использовать ее в файле state.js

rerenderEntireTree – является подписчиком

### Использование паттерна

До внедрения dispatch и action «самодельный» Redux выглядит так:

* **state.js**



* **index.js:**

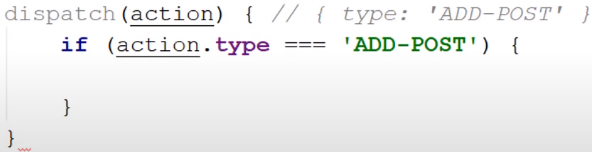


* **MyPosts.jsx** (как пример использования state.js):

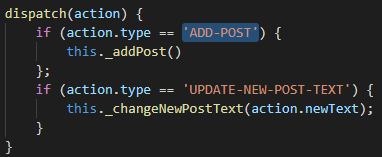


## Dispatch и action

Идея состоит в том, чтобы обращаться к объекту ***store*** через метод *dispatch* (который определим у store) и в качестве аргумента передавать **объект action**, на основании которого будет определяться то, что нужно сделать:

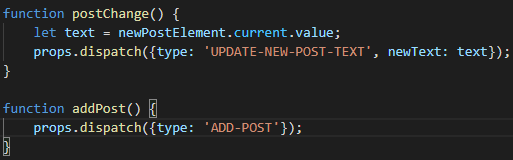


* **state.js** (dispatch)

****

Остальные методы в store остались те же.

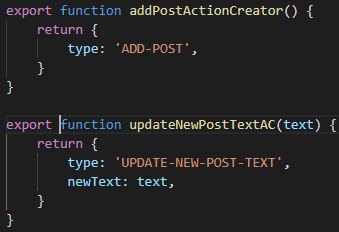
* **MyPosts.jsx** (как пример использования state.js)



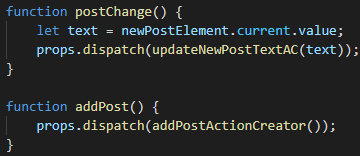
## Action creator, action type

В данной схеме у нас UI создает **объект action** самостоятельно. Это неудобно, т.к. можно ошибиться в названии свойства объекта, его количестве и т.д. Action creator() – это функция, которой мы делегируем создание action.

* **state.js**

****

* **MyPosts.jsx**

****

## Reducer (почти redux)

Редьюсеры позволяют распределить лишний код из метода dispatch.

**Reducer** – это функция, которая принимает state и action и возвращает новый state

В dispatch() вызываются редьюсеры, каждому передается своя часть стейта и объект action. Если в редьюсер пришел action с соответствующим типом события – он начинает работу по измене6нию своей часть стейта.

# Основы Redux (react-redux)

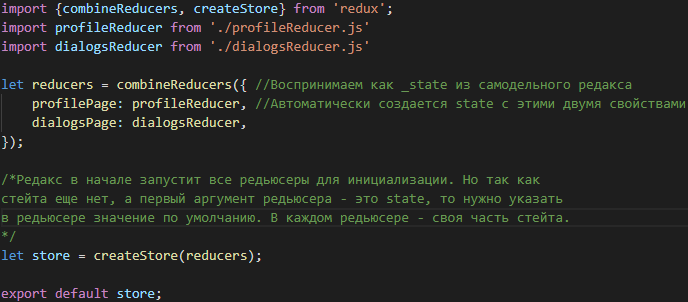
## Установка Redux, подключение

* npm install –-save redux– установить redux с занесением в package

У редакса есть функция, которая создаст нам store на подобии того, который мы делали вручную (*redux-store.js*):

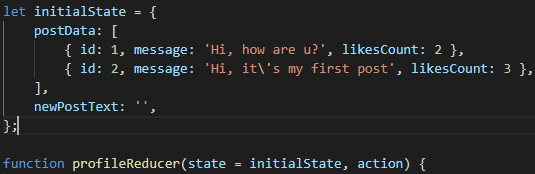


Вместо самодельного редакса подключаем настоящий в новом файле *redux-store.js:*



Функция *combineReducer*s принимает **объект редьюсеров**, на основании которых будет создан \_*state* в объекте *store.* Формат объекта следующий: {ветвьСтейта: редьюсерВетви, …}

Функция *createStore* вызывается от *combineReducers()* и возвращает объект *store.* Редьюсеры должны возвращать свою часть стейта. В начале редакс вызывает сам метод стора *dispatch(action),* внутри которого будут запускаться редьюсеры от своей части стейта и объекта action. НО! В начале у нас нет никакого стейта и редьюсеры будут запускаться от undefined и возвращать его же. Поэтому инициализируем редьюсерам начальные значения их частей стейтов:



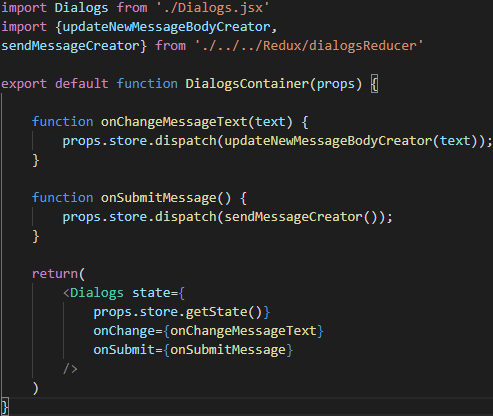


## Container component (контейнерная компонента)

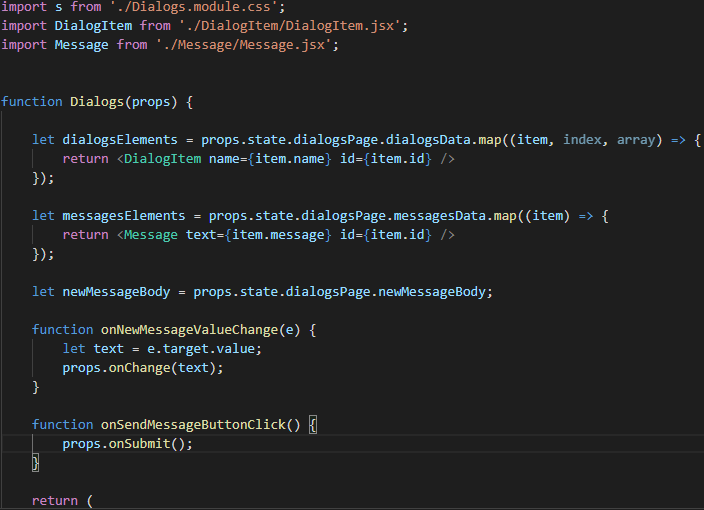
Задача контейнерной компоненты – удовлетворить нуждам презентационной. Презентационная компонента не должна знать что-либо о сторе или стейте, а контейнерная может получать в себя сразу весь стор, это разрешено. То есть контейнерная компонента служит оберткой над презентационной и передает через пропсы нужные коллбеки и/или стейт.

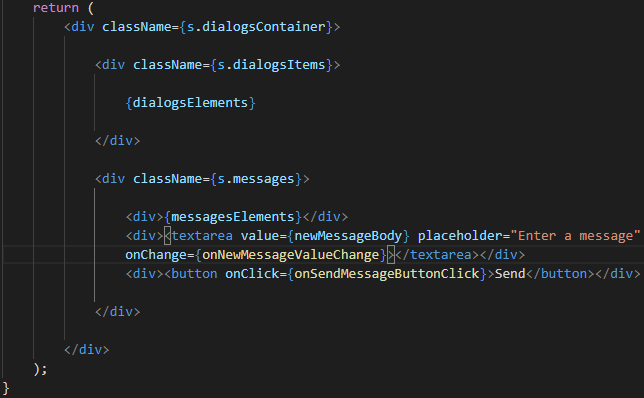
Пример:

* Обертка



* Компонент в обертке:





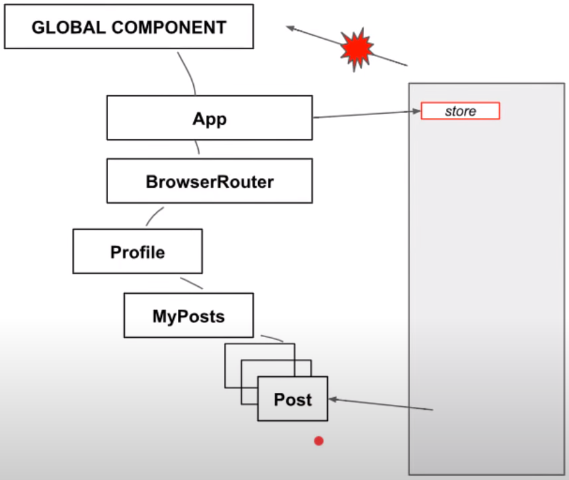
## Context API

### Понятие

Позволяет не перекидывать данные (store, например) напрямую в нужную компоненту, а не передавать через пропсы.

Родительская компонента объявляет некий контекст и добавляет в него данные. Теперь дочерние компоненты могут к обращаться к этим данным.

Родитель родительской компоненты, которая создала контекст, не имеет доступа к этому контексту.

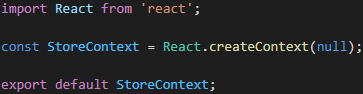


### Использование

* **Создаем контекст, можно в отдельном файле:**

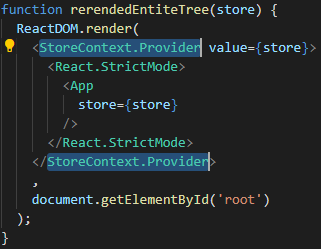


В качестве defaultValue можно передать null. По сути **createContext()** вернет компоненту:



* **Оборачиваем тэг, вокруг которого нужно создать контекст:**

****



Теперь store будет доступен глобально дочерним компонентам <App /> Пропс *store* теперь нет смысла прокидывать.

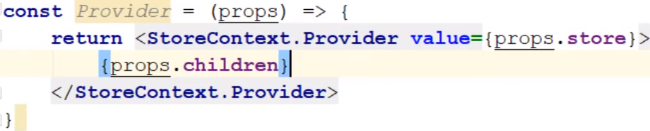
Презентационные компоненты не должны обращаться к контексту, пусть обращаются их контейнерные компоненты.

* **Потребление контекста**

Чтобы компонент был потребителем, нужно отрендерить его внутри тега <StoreContext.Consumer>. Причем внутри этого тега должна быть функция, которая в аргументе получает то самое значение *value*, в которое мы передали *store* () и возвращает JSX-разметку:



* **Вариант инкапсуляции оборачивания**

****

## Основы React-redux: connect, mapStateToProps, mapDispatchToProps

### Понятие

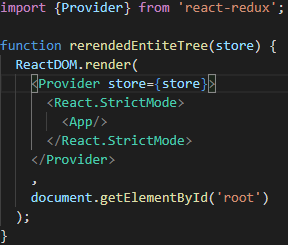
React-redux – это некая прослойка, которая обеспечивает нам более легкое общение между реактом и редаксом. Т.к. изначально редакс не предназначен именно для реакта, а реакт-редакс делает его более удобным для реакта.

* Установка: npm install react-redux

### Использование

* **Создание и оборачивание**

Похожий провайдер, который мы вручную написали выше из реактовского Context.API (**Вариант инкапсуляции оборачивания**), уже есть в библиотеке реакт-редакс:



Причем пропс должен называться именно store!

* **Потребление**

**React-redux** предоставляет удобный способ для создания **контейнерной компоненты**:

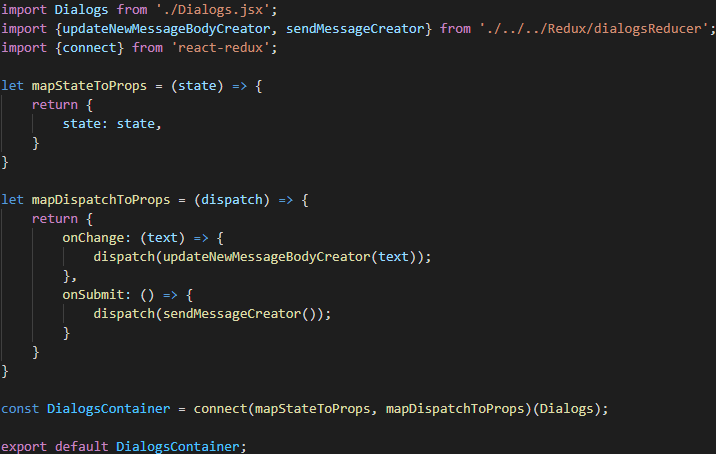


Функция коннект вернет другую функцию, которую нужно вызвать от компонента, который нужно поместить в контейнер:

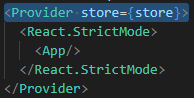


На данном этапе у нас получилась контейнерная компонента, которая просто рендерит Dialogs

* **Потребление: предоставление стейта и коллбеков (через пропсы)**



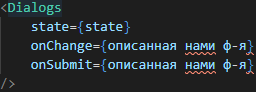
**mapStateToProps** – функция, которая будет автоматически вызвана от ***store****.getState()* который мы передали в пропсы **<Provider>** вот тут:



Данная функция должна возвращать объект с нужными данными стейта, которые будут переданы в пропсы.

**mapDispatchToProps** – автоматически вызовется от забинденого метода **dispatch** стора. В данной функции, в возвращаемый объект, мы должны записать нужные коллбеки, которые попадут в пропсы.

В итоге можно сказать, что в Dialogs попадут следующие пропсы:



### Немного теории

Метод **connect(**) делает автоматический **subscribe()**, и нам теперь не нужно делать так в index.js:

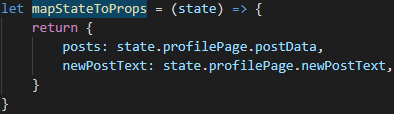


Да и вообще **rerendedEntireTree** поэтому нам больше не нужен, потому что наш контейнер подписан в connect.

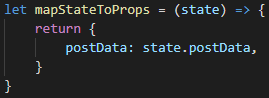
Каждый раз, когда изменяется стейт, запускается вот эта функция:



Как мы помним, эта функция возвращает новую версию части стейта. Если новая версия отличается от старой – то происходит перерисовка.

****

Если в posts будет сидеть новая ссылка – произойдет перерисовка! То есть если в пропсы стейт будет передаваться вот так:

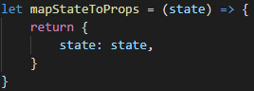


то в редьюсере будет достаточно лишь сделать поверхностную копию первого порядка:



Т.к. в postData уже будет другая ссылка на ветку стейта (копия).

Однако если в пропсы пихать весь стейт:



То все равно будет происходить перерисовка, даже если в редьюсере не делали копию. В этом случае достаточно сделать копию только в каком-то одном из редьюсеров.

По правилам нужно передавать в mapStateToProps только те данные, которые ей нужны, иначе компонент будет перерисовываться, когда изменяются данные в стейте, к которым он не имеет отношения

## Копия глубокая VS поверхностная

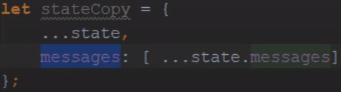
* Shallow copy
* Deep copy
* Spread operator

## Копия состояния в редьюсере

Когда в редьюсер приходит state – он должен сделать копию этого состояния и вернуть уже его. Если он вернет тот же самый объект state, то редакс будет думать, что изменений не было и не будет сравнивать два стейта, т.к. он будет один и тот же. А когда мы создаем копию и возвращаем уже ее, то редакс начинает сравнивать старый стейт и тот, который вернулся из редьюсера (в mapStateToProps). Однако если у обоих стейтов будут ссылочные поля – редакс опять не увидит между ними разницы и не запустит перерисовку, их тоже нужно глубоко скопировать!

Фишки при создании копии стейта:

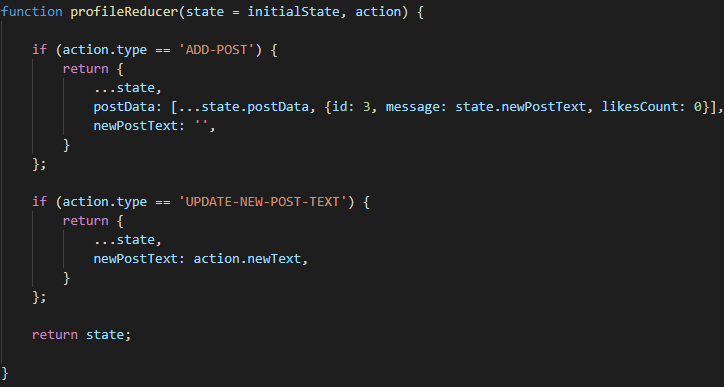
1. Если нужно отследить изменения только в одном свойстве (ссылочном) стейта, то можем создать поверхностную копию всего стейта и глубоко скопировать лишь то свойство, которое будет сравниваться:



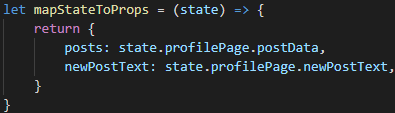
В данном случае мы перезаписали оригинальный messages на его копию.

**Глубоко копируем только то, что планируем изменять! Если объекты, которые сидят внутри messages мы менять не будем, а только добавлять или удалять что-то из массива messages, то достаточно создать лишь копию этого массива.**

Пример грамотного редьюсера:



Его mapStateToProps:

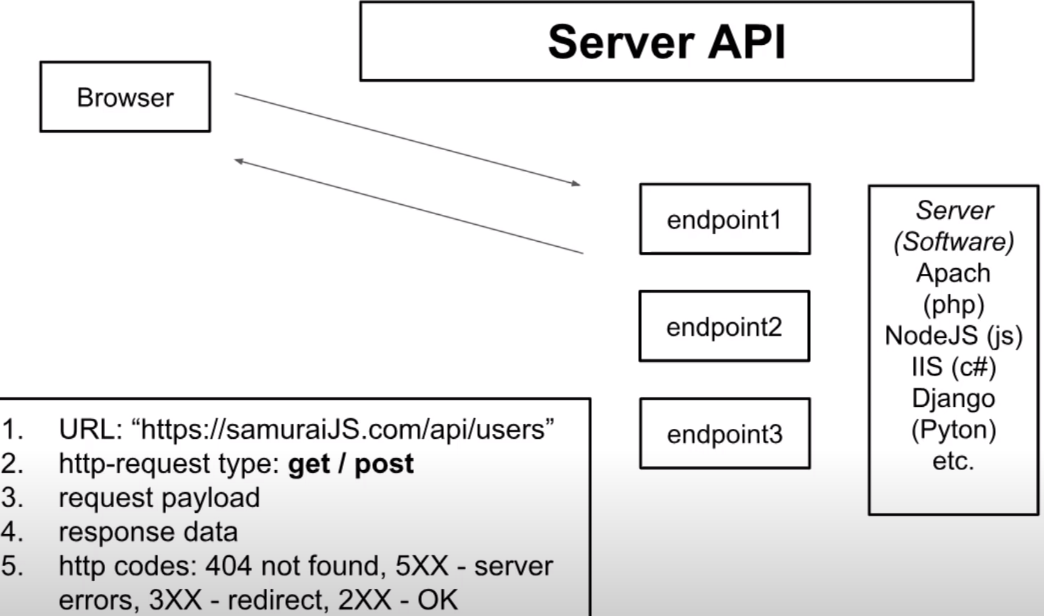


Если хоть какая-то ссылка (posts) или переменная (newPostText) изменятся – произойдет перерисовка.

## REST API

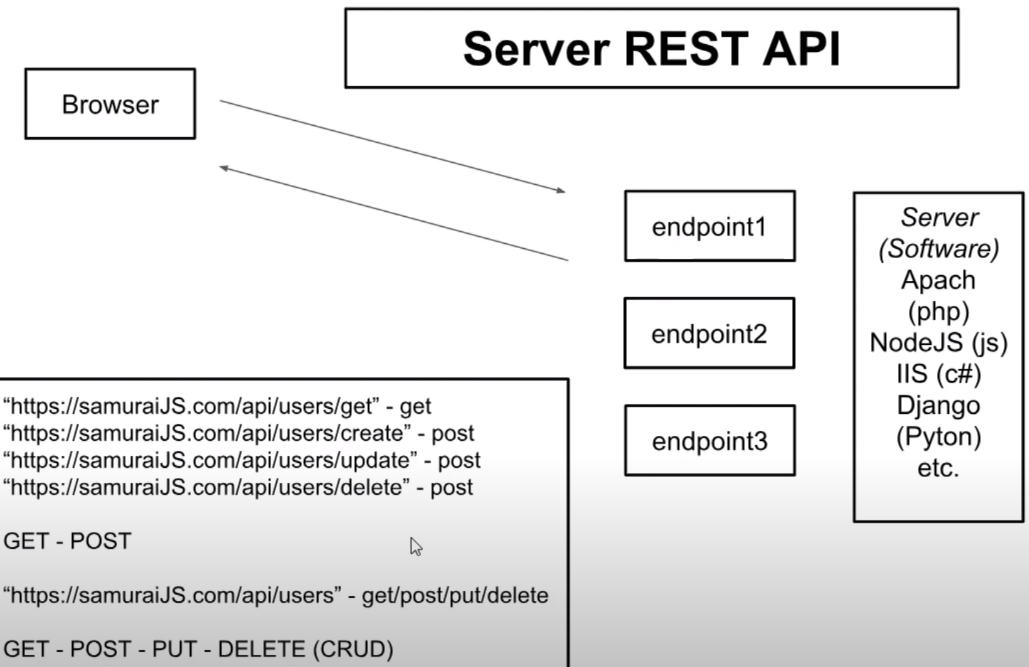
### Понятие Server API

API – Application Programming Interface – интерфейс взаимодействия с программой

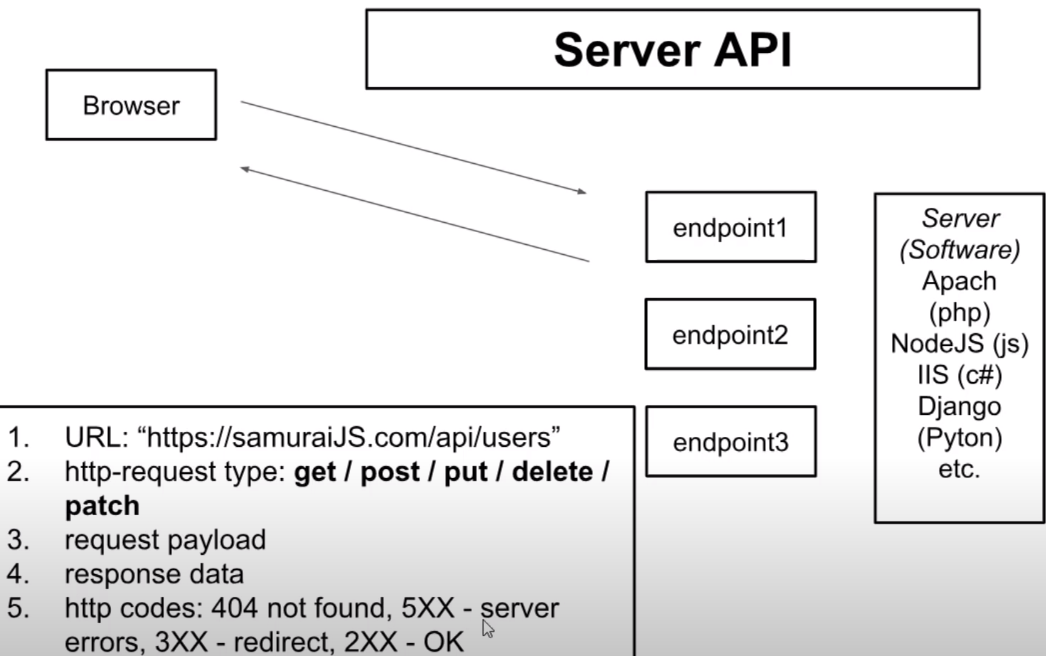
****

1. У каждого программного сервера есть **эндпоинты** – это URL, на которые мы можем делать **запросы**.
2. **Get запрос** используется – когда мы хотим что-то получить от сервера, по сути мы можем отправить только URL с этим запросом. **Post запрос** – когда хотим что-то отправить, с этим запросом уже можно отправить какой-то **Request payload.**
3. **Request payload** – что с запросом мы должны слать на сервер.
4. **Respons data** – какие данные нам придут в запросе
5. **Коды ответа сервера**. Например, 404 – ошибка.

### Понятие Server REST API

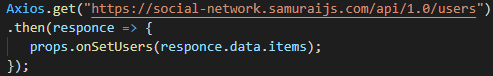


### В итоге



## Запросы. Axios

npm install axios – установка



### URL/URI

**URL:**

* <https://social-network.samuraijs.com/api/1.0/users?count=20&page=3>

и

* <https://social-network.samuraijs.com/api/1.0/users>

считаются одним и тем же URL, просто разные query параметры

**URI:**

Когда говорим про URI параметры – то фактически у нас меняется URL:

* <https://social-network.samuraijs.com/api/1.0/profile/2>

и

* <https://social-network.samuraijs.com/api/1.0/profile/3>

**это уже разные адреса**

## Чистые функции

В чистой функции нельзя допускать сайд эффекты:

1. Делать запросы на сервер
2. Модифицировать входящие параметры
3. Изменять DOM элементы
4. Изменять внешние переменные

Эти сайд эффекты не должны возникать именно при запуске функции. Например, допускается назначить обработчик на кнопку, которая будет делать запрос. Но нельзя чтобы такая функция запускалась автоматически при рендеринге компоненты.

## Методы жизненного цикла в классовой компоненте

Функция (функциональная компонента) обычно рендерит JSX и умирает. Классовая же компонента живет в памяти и возвращает JSX, если его об этом попросят, с помощью метода **render().** Но если мы перешли по другой ссылке (Роутер ее затер) – то объект умрет.

## componentDidMount

* **componentDidMount()** будет вызвана реактом, когда он отрисовал JSX, который вернула компонента. То есть JSX уже доступен на странице.
* Компонента **монтируется** (append) в DOM только один раз, а дальше только меняется.
* **componentDidMount()** вызывается только один раз, когда JSX компоненты был вмонтирован.

В этом методе можно делать запрос на сервер и другие сайд эффекты.

## Pagination. Постраничное поступление данных с сервера

Пользователю никогда не нужно загружать информацию о всех сущностях, которые хранятся на сервере в базе данных. Например, если в базе данных миллион пользователей, сервер разобьет эти данные на страницы и будет постранично возвращать клиенту данные.

### Постраничный запрос

* **Запрос конкретной страницы делается с помощью get запроса:**



После ‘?’ следуют **get-параметры (Query параметры)** в формате <ключ>=<значение>

Размер страницы можем указать также в get запросе:



* Количество страниц можно узнать, разделив кол-во записей на размер страницы

### Стрелочные функции как поля класса

Если метод класса объявить так:

onClickHandler = (e) => {…}

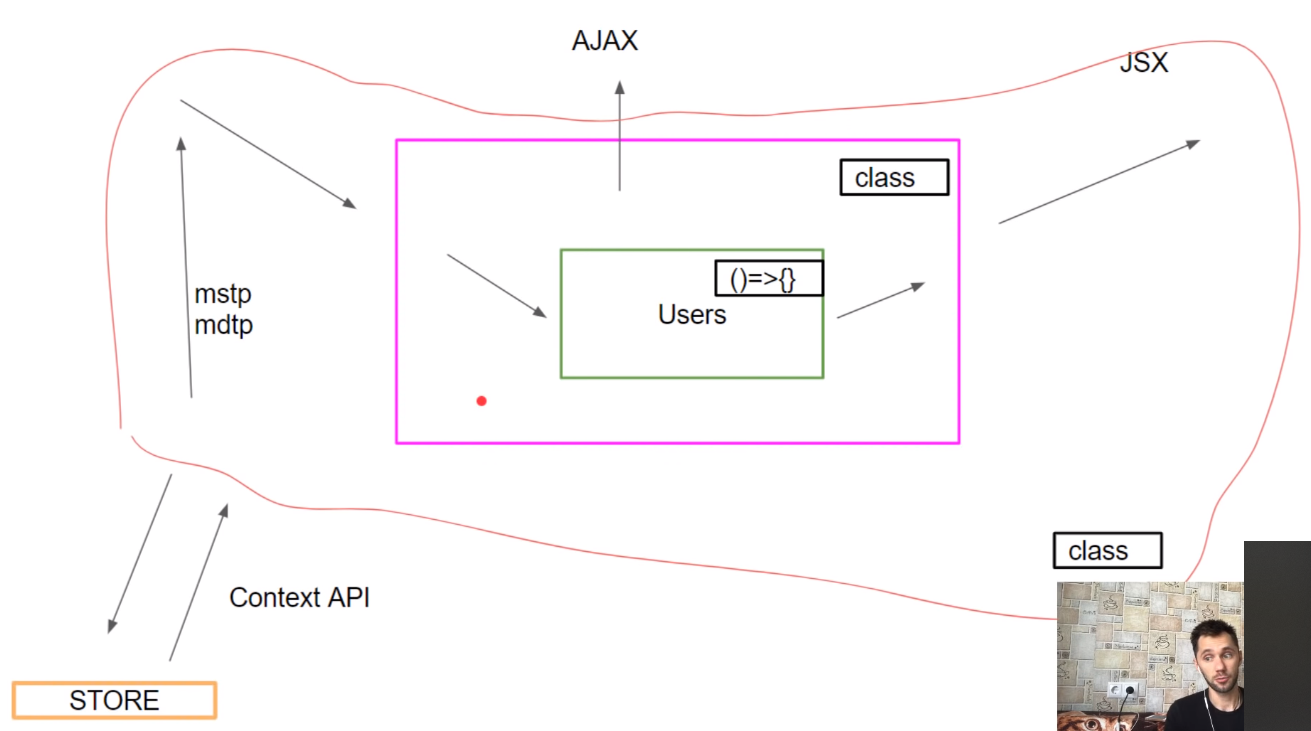
То его не нужно будет биндить, так как в таком случае:

onClick={this.onClickHandler}

onClickHandler не может иметь своего this в принципе, т.к. является стрелочной функцией и берет его из замыкания

## Презентационная и Контейнерная компоненты

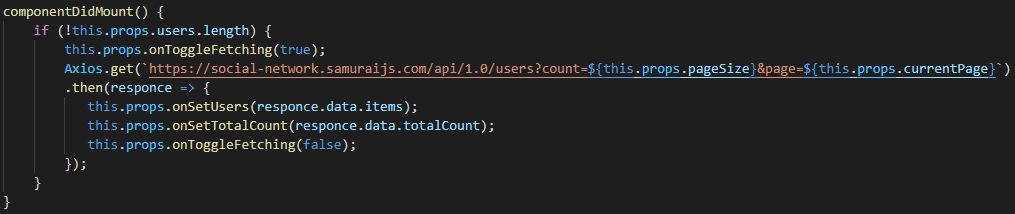
Классовая компонента должна вернуть в render() чистую презентационную компоненту:



То есть контейнерная отрисовывет классовую, а классовая – функциональную.

## Preloader, loader-gif, isFetching indication

Просто показываем картинку, пока в стейте isFatching: true



А когда в промисе получили response – переключаем isFatching в false

## mapDispatchToProps лайф-хак

Такую запись *mapDispatchToProps:*





Можно преобразовать в следующую:



То есть теперь не **mapDispatchToProps** будет возвращать объект с пропсами-обработчиками, а мы напрямую помещаем action-creator`ы в объект, который передается теперь вместо **mapDispatchToProps.** Теперь эти функции будут вызываться в компоненте как обработчик напрямую от нужных аргументов. Эти экшен-креаторы будут уже в коннекте автоматически диспатчиться в connect.

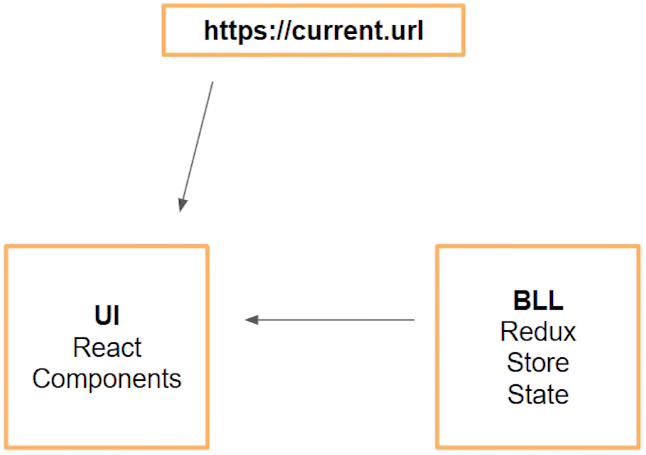
## withRouter

### Два источника правды

**withRouter –** позволяет компонентам реагировать на URL адрес в адресной строке браузера.

Адресная строка – такой же источник правды, наравне с redux

Один из вариантов подхода:



### Работа withRouter

**withRouter(<Component>) –** вернет компоненту, которая будет отрисовывать <Component>, а также будет передавать ей актуальный URL в браузерной строке. Все пропсы, переданные в компонент, который вернет **withRouter**, прокидываются и в **Component.**

В <Route /> параметр URI обозначается так:



Если добавить вопросительный знак к userId – это сделает этот параметр опциональным и компонента будет отрисовываться и при адресе /profile

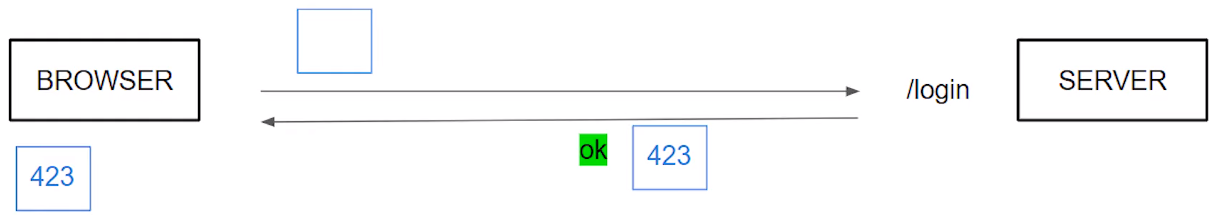
Например тут: [наш сервер/profile/2](https://social-network.samuraijs.com/api/1.0/profile/2), двойка – это параметр.

Благодаря этому, в пропсе match **Component** будет доступен параметр в свойстве **match.params**

## cookie, login в теории, auth/me

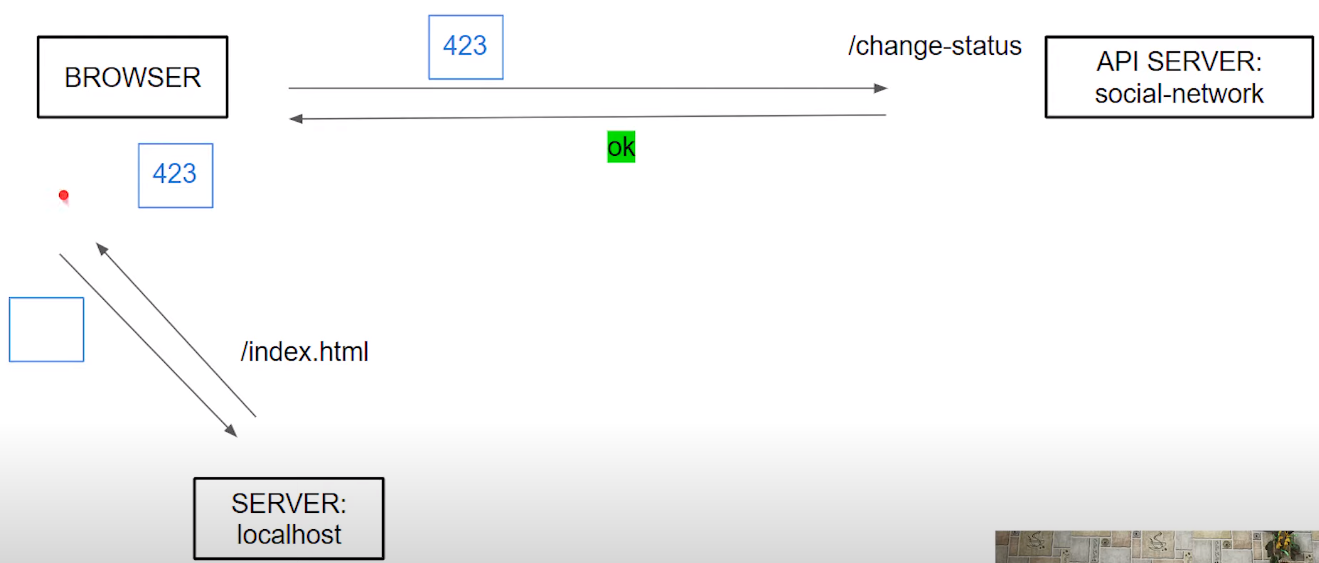
### Cookie

Когда мы логинимся, то сервер отсылает нам куки, в котором зашифрован наш ID. При каждом запросе, например сообщений, мы отсылаем этот зашифрованный ID в куках обратно на сервер, где он расшифровывается и принимается решение отдать данные или нет.

****

### Два сервера. Кросс доменный запрос.

Один сервер на *social-network (API).* Второй – на localhost, на котором развернуто наше SPA приложение:



Куки для этих доменов – разные. Когда идет запрос с локал хоста на на API – это кросс доменный запрос (CORS). Запрещено, загрузившись в браузере с одного домена, делать запросы на другие, только если тот другой домен этого не разрешил.

### Куки и кросс доменный запрос

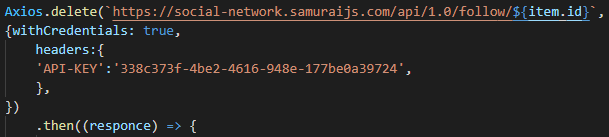
Также при такой схеме (кросс доменный запрос) браузер не отправить куки авоматически на наш API, когда мы загрузились с другого источника (в нашем случае локал хост).

Нужно явно указывать, что мы хотим отправить куки на API сервер:



В post запросе данный объект передается третьим параметром.

Только для get запроса не нужен специальный ключ, для остальных нужен:



В данном случае он отправляется как заголовок запроса (request-заголовок)

## DAL, axios.create

### DAL/Теория

Существует классическое разделение на UI, BLL и DAL (Data Access Layer).

DAL нужен для того, чтобы общаться с API, например, делать запросы. То есть сама логика запроса (строка запроса, параметры запроса) должны быть инкапсулированы в DAL.

### axios.create

axios.create() – создает объект и хранит в нем настройки работы с определенным API.

Пример:

