Начало курса – 4 мая (до этого 1-2 денька в перерывах посмотрел уроки примерно 10). Из них проёбано 18 мая и полвина 19 мая

Что такоей реакт.

Реакт это библиотека, которая отрисовывает пользовательский интерфейс, так же надо уточнить, что реакт это функциональное программирование, но там есть элементы ООП.

функциональное программирование — это парадигма программирования, в которой процесс вычисления трактуется как вычисление значений функций в математическом понимании последних. Функциональное программирование предполагает обходиться вычислением результатов функций от исходных данных и результатов других функций, и не предполагает явного хранения состояния программы. Соответственно, не предполагает оно и изменяемости этого состояния.

Для каких проектов подойдёт React

Резюмируя все особенности, можно выделить несколько типов проектов, которым подойдет React.

Если проект планирует расширяться, и над ним работает или будет работать команда разработчиков. Потому что в таком случае использование известной технологии поможет проще договариваться между разработчиками и лучше поддерживать код.

Средним и крупным проектам будет полезен компонентный подход, который в том числе лежит в основах React. Это упростит структурирование и переиспользование кода и даст выигрыш в долгосрочной перспективе.

Legacy-проектам, которым предстоит пройти через рефакторинг. Потому что React можно добавлять в уже существующий проект, обновляя кодовую базу постепенно и точечно.

React, возможно, не подойдет для простых приложений (одностраничных и одноразовых сайтов), потому что в этом случае для того, чтобы разобраться с настройкой проекта и окружения, уже понадобится много времени и труда.

Ещё React будет не самым удачным выбором для реализации частей проекта, которые очень чувствительны к потребляемым ресурсам. В этом случае возможно гибридное использование: когда само приложение по большей части написано на React, а требовательные к производительности места с ним не взаимодействуют – библиотека никак не мешает делать такие интеграции. Однако и тут следует подходить к вопросу без фанатизма: React достаточно быстрый в большинстве случаев, а оптимизировать узкие места всегда можно по факту.

2. Зачем NodeJS на фронте

Изначально frontend разработка – это разработка под браузер. И frontend разработчик писал много js, который обрабатывался по итогу в браузере. Но люди понял, что неплохо бы не только html странички в браузере отображать, ну а выполянть какие-то более сложные операции (работа с БД, с файловой системой, обслуживание http запросов и тд). Для этого создали альтернативную браузеру программу.

Node.js это программа (платформа), которая понимает js и выполняет его для своих целей, по сути это консолька. И браузер и node.js внутри себя используют движок v8, просто браузер дополняет движок своими элементами (модель DOM, события и тд) и также нода добавляет своими расширениями v8 (там же надо например с бд работать).

Зачем node.js фронтендеру – на node.js создано куча модулей (минификация, сборка проекта и тд)

NPM – пакетный менеджер, который устанавливается вместе с node.js, с его помощью мы ставим все пакеты (модули) и держим актуальные версии.

3. Установка NodeJS

https://nodejs.org/en/download/

4. create-react-app

Это программка, которая запустится с помощью NPX, чтобы по быстрому развернуть проект на реакте.

<https://create-react-app.dev/>

Либо на гитхабе: <https://github.com/facebook/create-react-app>

Пишешь в терминале: npx create-react-app my-app

Запуск проекта npm start

5. Как работает SPA

SPA – это маленькая полупустая html страничка в которую помимо немного html приходит много js.

Приходит одна страница единожды. Мало html, но много js (отсюда и появились frontend-разработчики, т.к. js стало до жопы). Js, который уже «сидит в браузере» делает запрос на сервачок, тот ему ворачивает json и js генерирует html. При этом перезагружать страничку не надо, потому что на js на клиенте время от времени шлёт запросики (не появилось ли чего там у вас).

url. Страничка не перезагружается при переходе, он перехватывает url и динамически генерирует новый контент, затирая старый.

6 урок index.js, App, JSX

Package.json является необходимым файлом, чтобы папка являлась проектом.

Jsx – это расширение синтаксиса JavaScript, которое позволяет использовать HTML-подобный синтаксис для описания структуры интерфейса (позволяет писать html внутри js (как html только внутри js – реакт это конечно не как html интерпретирует, а как особоые инструкции по созданию virtual dom элементов – все наши теги превращаются в дом элементы (объекты) ). React можно использовать и без JSX, но большинство людей ценит его за наглядность при работе с UI, живущем в JavaScript-коде. Помимо этого, JSX помогает React делать сообщения об ошибках и предупреждениях понятнее.

Файлы Jsx - В react v17 запилили новый jsx transform. И с версии 17 больше не нужно импортировать React, а раньше в каждом jsx файле писали в начале страницы 

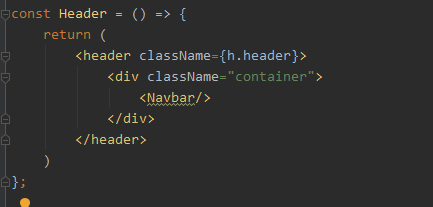
после return ставятся скобочки если то, что после return начинается с новой строки.

Точка и слеш значат что импортируем что-то, что лежит в той же папке.

7. что такое Компонент

Компоненты позволяют разбить интерфейс на независимые части, про которые легко думать в отдельности.

Компонент – функция, возвращающая разметку jsx (мы сами эту функцию не вызываем – её вызывает реакт где-то за кадром) или это класс, который наследуется от React.Component и который возвращает jsx (как миниум у него есть метод render(). Можно условно сказать, что компонента это тег, который можно вставлять в наше приложение. Называть принято с большой буквы. Во многом компоненты ведут себя как обычные функции JavaScript. Они принимают произвольные входные данные (так называемые «пропсы») и возвращают React-элементы, описывающие, что мы хотим увидеть на экране. Возвращать компонента должна один корневой тег. Вот пример компоненты:



Это пример так называемой функциональной компоненты, еще бывают классовые.. Об этом потом.

Дак вот при SPA наш index.html приходит в браузер единожды и дальше всегда чтобы мы там не запросили к нам будет приходит этот index.html и много js, который будет отрисовывать содержимое как раз таки внутри вот тега с id = 'root' (внутрь засунёт всё то, что мы пишем в js файлах – сейчас конкретно в index.js). Почему засунет в этот див? Потому что это укзано здесь:

И будет искать как раз таки функцию с именем App, т.к. в рендере мы вставили те jsx под именем <App />.

Компоненты можно вставлять друг в друга.

Смысл в том, что мы дробим приложение на компненты, напрмер <Header />, <navbar />, <Socials\_icons /> и т.д.

В одном файле можем описывать несколько компонентов, но обычно так не делают, всё разносят по отдельным файлам.

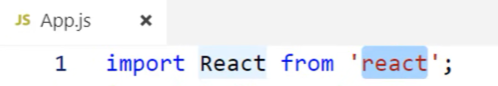
8. import\export

Всё, что объявлено в одном файле в реакте, будет жить только в этом файле. И поэтому используются импорты и экспорты:

Устроено так что если файл использует import, то в файле, откуда импортируется нужно написать export:

Здесь вот он импорт делал с маленькой буквы, но например ты хочешь сделать импорт компоненты с шапкой и назвал его header, дак когда будешь вставлять получишь хер:

Так что фигачь с большой буквы.

Если импорт указан как просто имя без точек и слешей, то это импорт из node\_modules: 

10. Использование export default

Если мы хотим экспортировать единственное значение или иметь резервное значение (fallback) для данного модуля, мы можем использовать export default (можно сказать что, что-то главное экспортирутеся, что-то по дефолту, что-то одно).

Короче при экспорте мы обычно даём одноимённые названия: Но, есть опасность если проебунькаешься с названием и экспорт сделаешь дефолтом, то по дефолту он привязку никакую не делает по имени и ты типо из Header.js по дефолту сделал экспорт, а в файле где делаешь импорт назвал Technologies и всё приплывёшь потом, запутаешься. То есть где импортируют могут дать любое другое имя, а не то которое ты пишешь когда экспортируешь.

Еще там про плагин для vs code: Дима ставил и короче он может иногда вытянуть (подключтиь путь) не то, что нужно.

11. вёрстка, grid

В файлах js в реакте мы пишем у дивов не class, а classname.

Свойство grid-area даёт название элементу чтобы можно было ссылаться на него с помощью шаблона созданного через grid-template-areas свойство.

Свойство grid-area является сокращенным свойством для grid-row-start, grid-column-start, grid-row-end и grid-column-end, определяя размер и расположение элемента сетки.

12. Разбивка на компоненты

Разбивал на практике вёрстку.

13. git-репозиторий

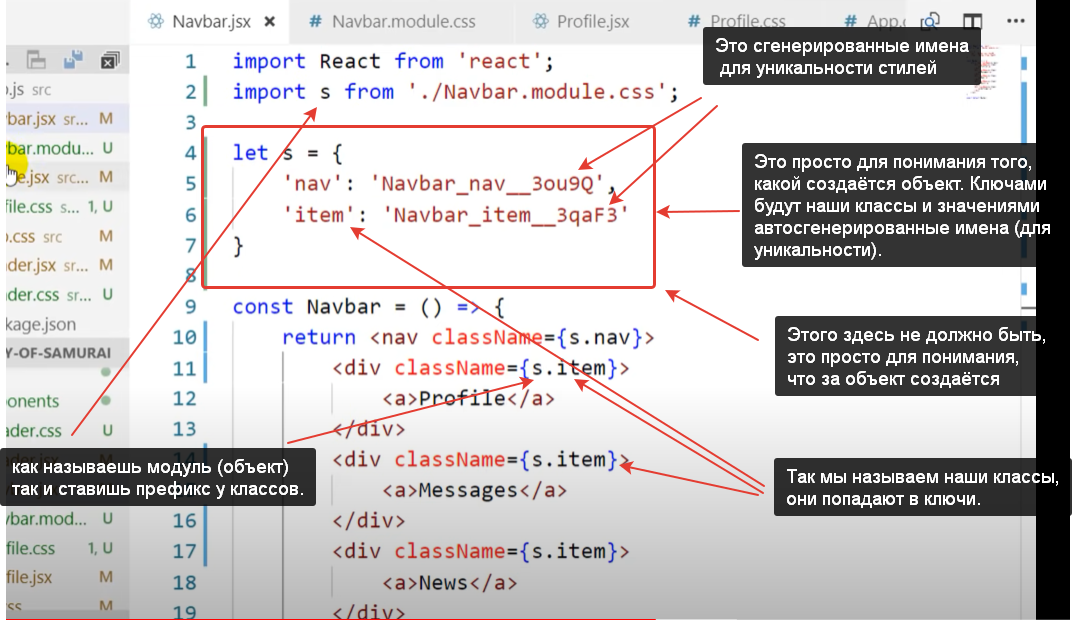
<https://github.com/it-kamasutra/react-way-of-samurai>

В гит репозиторий никогда не закидываем папку node\_modules (она у нас всегда в гитигноре).

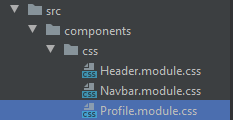
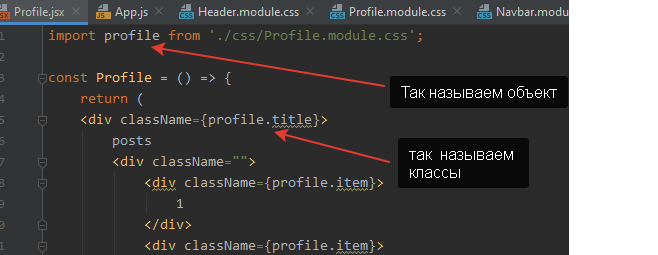
gitk –all& это графическая оболочка, отображает изменения в репозитории или выбранном наборе коммитов

14. css-модули, css-modules

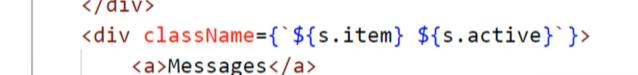
Зачем: Модули нужны, чтобы не париться с названиями классов (не пересекаться с другими стилями и не писать все стили в 1 файле.

Как работает: Есть такой Bundle.js – единственный js файл, в который превращаются все наши jsx файлы, index.js, App.js. В данном уроке для нас это означает, что если мы в какой-то отдельный созданный файл стилей для компонента например Header.cssпишем стили для класса item (к примеру), то эти стили будут действовать на все компоненты (если например в Navbar.jsx есть элемент с классом item, то на него тоже подействует). Соответственно нам нужны модули, когда мы превращаем файл стилей в модуль (дописываем «.module.») реакт понимает, что теперь это не обычный css, а это теперь модуль css (он же css объект – структура данных у которой есть ключ и значение). Реакт для каждого класса сгенерирует префиксы и постфикс и они получатся уникальными. 

Реализация:

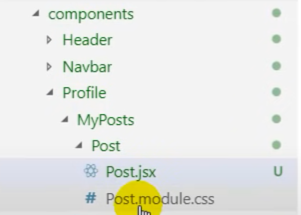
1. Делим стили на файлы и в название добавляем «.module.»: 
2. в jsx файле делаем импорт указывая названия css объекта и в classname пишем специальную конструкцию: 

Нюансы:

1. в classname мы не можем писать дефис, вернее можем, но это не удобно потому, что конструкцию придётся оборачивать в спец скобки и тд. Так что договорённость не писать дефис, пишем через верблюжью нотацию.
2. Чтобы добавить к элементу несколько классов (например плюсом к основному клссу добавить .active) можно применить шаблонные строки: 

Либо так: 

15. структура папок, новые компоненты

В папке components мы дробим компоненты на опдкомпоненты : 

Делаем так: одна компонента – одна папка (называть с большой буквы – ну так и Димыкак бы), в ней jsx и css модуль.

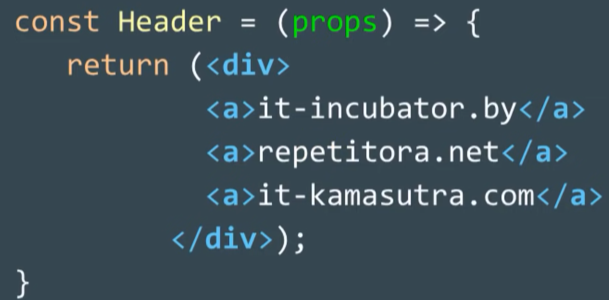
Если оставляешь просто css ,а не модуль, то импортирушеь без приблуд вот так: 

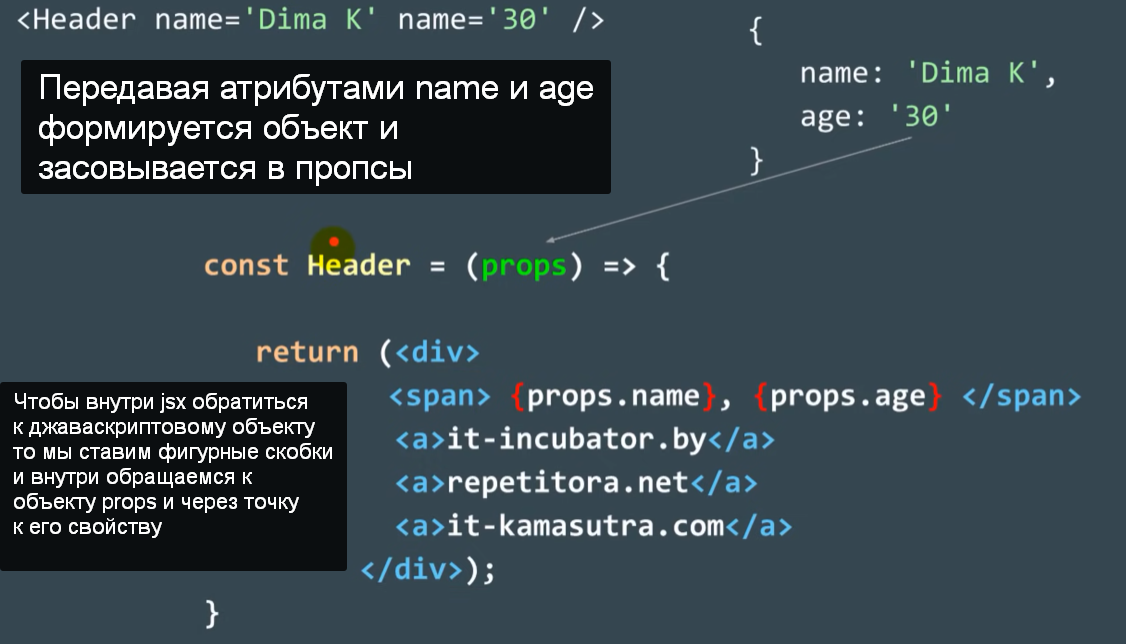
А модуль уже как объект импортируешь: 

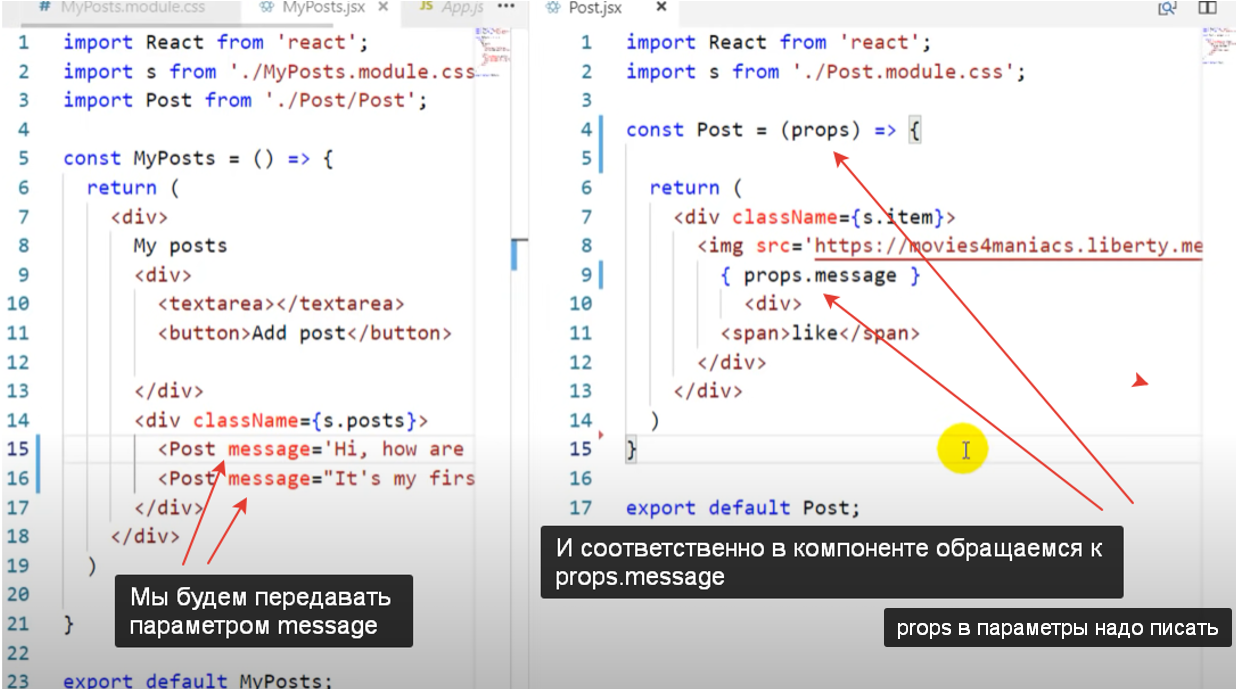
Как правило стараются называть файл по имени компоненты, которую он экспортирует из себя.

На каждую страничку будет отдельная компонента.

16. props

Компонента это функция, она всегда вызываетя с параметрами, всегда! Реакт когда вызывает компоненту, он всегда вызывает её с параметрами, самый главный параметр принято называть props (в реакте). Даже если ты не передаёшь ничего, props (объект) в компоненту приходит всегда.

Но вызывать функцию на прямую нельзя и чтобы передать параметры мы просто пишем их в атрибуты (можно провести аналогию, что компонента это тег и как любой тег например input, у него есть атрибуты: 



Debugger; - останавливается выполнение в этом месте и мжоно посмотреть значения переменных.

Моментик еще такой, что принято не пихать в функцию всё подряд. То есть не надо передавать компоненте через пропсы то, что ты не будешь использовать, то что этой компоненте не нужно, ну типо компонента отображает пользователей, а ты туда запихал посты. Так же не принято, что функция работала с какими-то глобальными переменнымми (нарушается принцип функций), ну для этого и есть собственно пропсы.

17. остановочка «react за час, часть 1»

Когда нужно создать с нуля приложение на реакте – создаём через create react app (для большинства нужд будет достаточно).

С реактом ставится куча модулей (их рил дофига) – они лежат в папке node\_modules и помогают нам в разработке (live reload браузера, автоматическая сборка, минификатор, реакт библиотека, соединение файлов в один и дохеара дургого).

19. route, browser-router, маршрутизация

Задача роута следать за адресной строкой браузера и если увидел изменения, которые подходят под роут – отрисовал что-то.

npm install react-router-dom – установка пакета для роутинга.

После установки пакета появится специальный компонент Route и BrowserRouter.

Такс.. после установки нихуя не работает npm start. Вот типы в коментах решали так: Для корректной работы сервака нужно: 1. Выйти с проекта: cd ../ После прописать: 1.1. npm cache verify 1.2. npm install -g create-react-app 1.3. npx create-react-app my-app 2. Зайти в папку проекта my-app > зайти в папку packaje.json и заменить код на "start": "react-scripts --openssl-legacy-provider start" Всю инфу взял у чуваков снизу, для удобства соединил вместе 3. также возможно понадобится установить доп. пакет - npm install [react-scripts@2.1.8](mailto:react-scripts@2.1.8)

Я в итоге всё наебнул нечаянно и установил по новой реакт ☺

Использование:

Импортируем: import {BrowserRouter, Routes, Route} from "react-router-dom";

Далее корневой тег в jsx оборачиваем в компонент <BrowserRouter> </BrowserRouter>

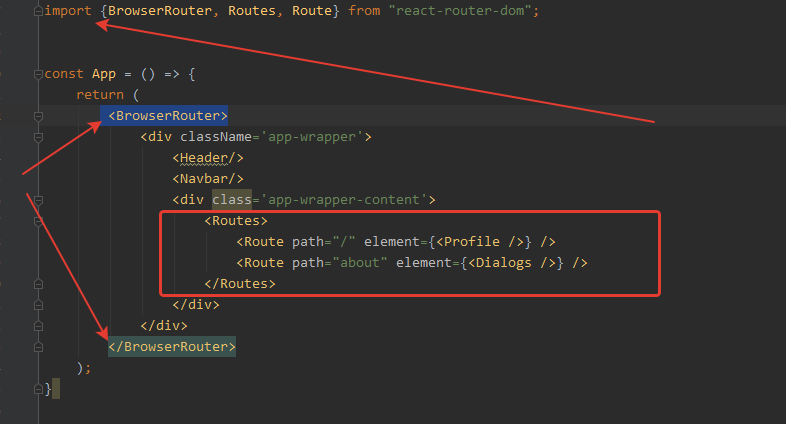
Далее нужный компонент рендеришь через такой синтаксис:

<Routes>

<Route path="/" element={<Profile />} />

</Routes>

(т.е. находясь на домашней странице с url = ‘/’ мы нам отрендерится компонент <Profile />



20. NavLink

Используется как замена для обычных ссылок (<a>), чтобы не перезагружалась страница.

Использование:

Импортируем: import {NavLink} from "react-router-dom";

<a> заменяем на <NavLink> и href менем на to:

<NavLink href="/settings">Настройки</NavLink>

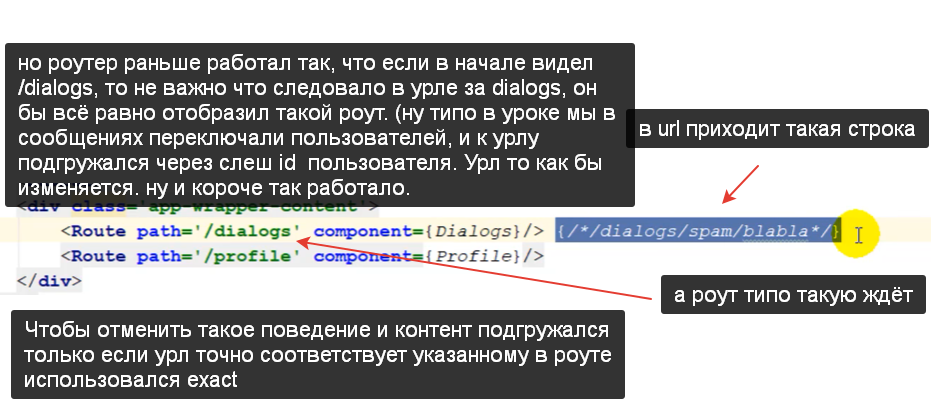
Класс active для <NavLink> По умолчанию NavLink добавляет class active (типо прослушивает и когда перехватил url то сам добавит active к ссылке, это уже сделано самим реактом, но не всегда удобно где-то в общих стилях описывать действия для таких ссылок. И поэтому поступают так:

<NavLink **className={ navData => navData.isActive ? s.active : "" }** to="/messages">Сообщения</NavLink>

А в css модуле пишешь к примеру:

.nav\_\_item a {  
 color: #ffffff;  
}  
.nav\_\_item .active {  
 color: red;  
}

22. Route exact

Точный параметр отключает частичное соответствие для маршрута и гарантирует, что он возвращает маршрут только в том случае, если путь ТОЧНО совпадает с текущим URL-адресом. Напрмер:  Дима писал еще на старой версии React Router, но в React Router v6 и выше поведение изменилось. Теперь по умолчанию роутер работает только на полное совпадение, чтобы сделать поведение как без exact ("React exact" используется по дефолту сейчас) роуты пишем так:

<Route path="/profile/\*" element={<Profile/>}/> (добавляем слеш и звёздочку после названия роута).

24. метод массива – map

Короче у любого массива можем вызвать метод map. Он принимает стрелочную функцию (это его единственный параметр). Эта функция вызывается проходится по каждому элементу массива. И он нам нужен чтобы преобразовывать в компоненты массивы, которые прилетают с сервака. Пример:

let dialogsData = [  
 {id: 1, name: 'Антон'},  
 {id: 2, name: 'Анна'},  
 {id: 3, name: 'Диана'},  
 {id: 4, name: 'Милана'}  
];  
  
let dialogsElements = dialogsData.map(dialog => <DialogItem name={dialog.name} id={dialog.id}/>)

Тут как бы DialogItem описан выше вот так:

const DialogItem = (props) => {  
 return (  
 <div className={s.users\_\_item}>  
 <NavLink to={"/messages/" + props.id}>{props.name}</NavLink>  
 </div>  
 )  
};

Дальше просто вставляем полученный масив с компонентами, реакт их выведет друз за другом.

<div className={s.users}>  
 {dialogsElements}  
</div>

26. структура проекта

Грубо говоря есть 2 слоя: UI(user interface) и BLL (business logic layer). Суть в том, что данные должны находиться в BLL. А конечный пользователь работает уже с UI, UI по сути это наш реакт с компонентами (отображение элементов в браузере и взаимодействия с кнопочками там и прочим).

27. прокидываем props через Route

Вот пример:

let posts = [  
 {id: 1, message: 'Дратути', likesCount: 12},  
 {id: 2, message: 'js php css html go react.', likesCount: 10}  
];  
  
const App = () => {  
 return (  
 <BrowserRouter>  
 <div className='app-wrapper'>  
 <Header/>  
 <Navbar/>  
 <div class='app-wrapper-content'>  
 <Routes>  
 <Route path="/profile/\*" element={ <Profile posts={posts} />}/>  
 <Route path="/messages/\*" element={<Dialogs/>}/>  
 </Routes>

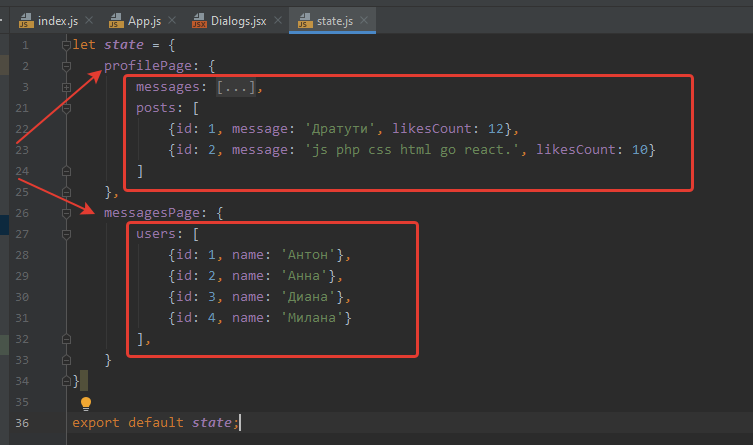
Вот так Карл:

<Route path="/profile/\*" element={ <Profile posts={posts} />}/>

29. (упаковываем данные в state)

Index.js не является для нас UI слоем, по этому мы можем сделать импорт всех данных (нашего стэйта) сюда.

И дальше уже через пропсы прокидываем куда нужно (по всем правилам функционального программирования, чистые функции все дела – если вводим одни и те же параметры, то чистая функция всегда возвращает один и тот же результат).

Концепция такая: все данные хранятся в одном объекте state. Дима так делал: в объекте делал «ветки» и каждая ветка обслуживает одну страницу:

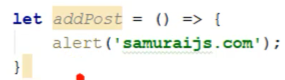
31. onClick, ref, VirtualDOM

onClick – событие «клик» (что будем делать во время клика по элементу), делается обычно через стрелочную функцию: Это концепция колбеков, когда мы сами не вызываем функцию, а отдаём кому-то, чтобы её кто-то другой вызвал. То есть функция обработчик события является прмером колбека.

Функция в js является объектом (который можно вызвать или передать кому-то).

Частая ошибка когда передают вызванную функцию (типо со скобками): 

Вот еще пример как можно (надо) БЕЗ СКОБОК!!!:





DOM элементы – это браузерная технология, предназначенная в основном для определения области видимости переменных и CSS в веб-компонентах. Главная проблема DOM — он никогда не был рассчитан для создания динамического пользовательского интерфейса (UI)

VirtualDOM – Виртуальный DOM (VDOM) — это концепция программирования, в которой идеальное или «виртуальное» представление пользовательского интерфейса хранится в памяти и синхронизируется с «настоящим» DOM при помощи библиотеки, такой как ReactDOM. Вместо того, чтобы взаимодействовать с DOM напрямую, мы работаем с его легковесной копией. Мы можем вносить изменения в копию, исходя из наших потребностей, а после этого применять изменения к реальному DOM.

И именно реакт занимается тем, что из virtual dom приносит всё в DOM (перерисовывает, обрабатывает и тд). Мы не имеем права лезти на прямую в ДОМ, и такие штуки как document.getElementById и т.п. как раз лезут на прямую в ДОМ – они запрещены в реакте. И чтобы получить значения из каких-то элементов (textarea например) – мы должны дать ссылку REF:

1.Создаём сыслку: 

2.Привязываем к textarea: 

3.Потом можем к этой ссылке обращаться (чтобы обратиться к элементу, на который эта ссылка ссылается, нужно у этой ссылки еще взять current и только потом обращаться с элементом:

Но тут момент с помощью ref мы стучимся на прямую к дом элементам, а мы должны работать с VDOM (дима говорит считать не так страшно, но изменить уже проблема – далее проработаем).

НахуйНадо: Короче в 40 уроке говорит что почти всегда мы рефы будем избегать, вместо этого когда у textarea (к примеру) срабатывает событие, то она присваивает функции обработчику этого события объект события event. И вот с помощью этого event мы можем достучаться до объекта, с которым произошло событие, от туда и взять value.

32. прокидываем callback через props. Экспорт без default

Функция, которая меняет state – дожна лежать в этом стэйте. Поэтому пилим в state:

export let addPost = (postMessage) => {  
 debugger;  
 let newPost = {  
 id: 5,  
 message: postMessage,  
 likesCount: 0  
 }  
 ***state***.profilePage.posts.push(newPost);  
}

Сначала импорт:

import {addPost} from './redux/state';

Дальше так же как с данными прокидываем до нужного файла:

<App state={***state***} addPost={addPost}/>

Дальше по файлам:

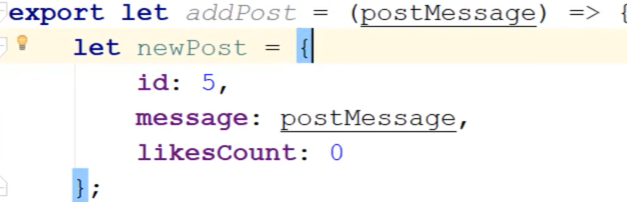
<Route path="/profile/\*" element={<Profile  
 posts={props.state.profilePage.posts}  
 addPost = {props.addPost}/>  
}/>

В Иотге в конечном файле что-то типа:

let newPostElement = React.createRef();  
let addPost = () => {  
 let text = newPostElement.current.value;  
 props.addPost(text);  
}

<div className="">  
 <textarea ref={newPostElement}></textarea>  
</div>  
<button onClick={addPost}>Добавить сообщение</button>

Создавая переменную, её можно сразу экспортировать (если без дефолта – то есть импортировать мы её должны немножко иначе). Пример:

1. Экспорт: 
2. Импорт:  (добавляются фигур. скобки и тут уже в названии (конкретно тут addPost) мы не можем другое описание дать – он не найдёт).

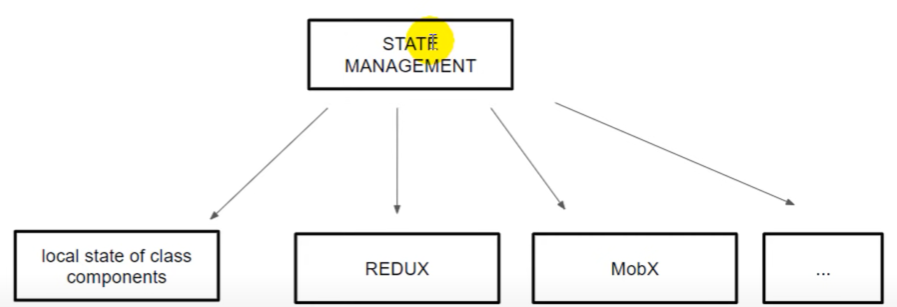
33. добавление поста на стену (зачатки FLUX-концепции)

Нам нужно на каждое изменение в дереве перерисовывать дом. Только state знает, что что-то произошло, так как данные находятся там. Получается надо вызывать функцию в нём. Но ререндер происходит из index.js, и так сделать мы не можем из-за циклической зависимости

Поэтому внедряем отдельный файл например render.js чтобы избежать циклической зависимости (импорт делаем в оба файла): 

FLUX – Архитектура организации потока данных, когда на UI что-то кликается, первым делом мы должны изменить state и только потом идёт перерисовка.

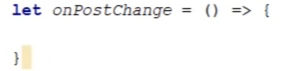
34. (FLUX-круговорот на каждый символ)

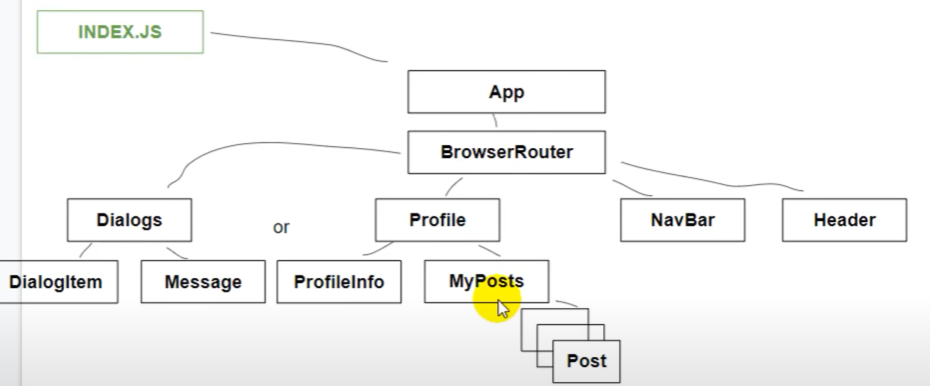
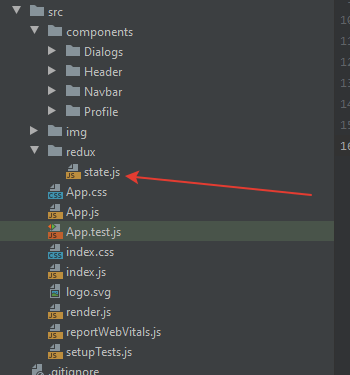
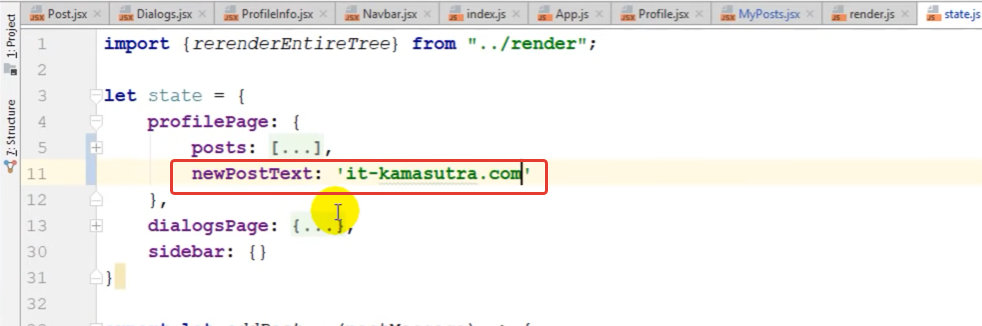
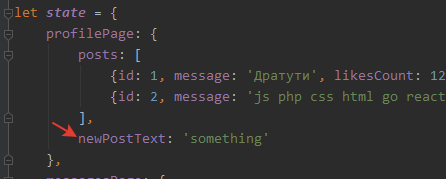
Систем архитектур (, управление состоянием, или по другому построение логики – что за что отвечает, где лежит BLL и UI) много: 

Мы будем двигаться на redux – она реализует flux архитектуру (Когда нужно придерживаться концепции когда что-то вводится в UI и тут же попадает в state. Например чел вводит какое –то значение в инпут к примеру, каждый новый символ должен сразу попадать в состояние и уже от туда рендериться в UI ).

Дальше по уроку по шагам:

1. Определяем в файле MyPosts (там у нас добавляются посты) обработчик события onChange на нашу textarea:  Он (обработчик) будет срабатывать каждый раз, когда будет меняться внутренне содержимое textarea (либо input и тп)
2. Выше в этом файле определяем функцию onPostChange



1. Нам надо чтобы value нашей textarea была равна чему-то из пропсов, например newPostText: 
2. Дальше вот наша структура и нам надо прокинуть то, что мы придумали (newPostText ) по нашему дереву и добавить еще в state
3. Идём на самый верхний уровень в state.js  добавляем 
4. Теперь newPostText прокидываем по дереву (как прокинуть должен сам уже уметь). После этого у тебя получится, что в поле ввода будет то, что написано в твоём стейте. НО нам надо это менять. 
5. Тут юзаем обработчик onChange. Можем обратиться через ref (Напоминалка:

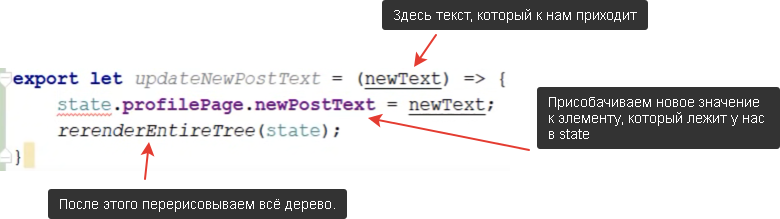
let newPostElement = React.createRef();

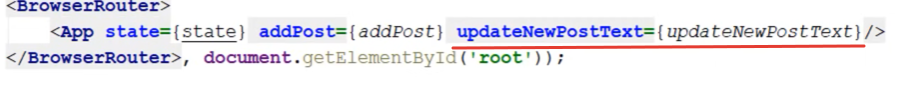
<textarea ref={newPostElement}/>

)

к нашему элементу (textarea) взять его Value (

) и запинддюрить в state.

1. Наш стейт меняется только с помощью функций, которые определены здесь же в state. Так что создаём новую: 
2. Дальше эту функцию так же нужно прокинуть через пропсы. Пишем в атрибут в App ( в файле render.js) не забывая про импорт тк функция находится в стейте 

10. в постах :

let addPost = () => {  
 let text = newPostElement.current.value;  
 props.addPost(text);  
 props.updateNewPostText('');  
};  
  
let onPostChange = () => {  
 let text = newPostElement.current.value;  
 props.updateNewPostText(text);  
}

1. В state.js:



35. callback, subscribe, observer

Короче суть урока – избавиться от циклической зависимости – у нас render импортировал addPost и updateNewPostText из state.js И ТАКЖЕ state.js импортировал из рендера rerenderEntireTree. Короче это херня (цикл. Зависимость) и надо убирать такое из архитектуры.

Нам надо вызывать функцию из state.js и в этой функции передать rerenderEntireTree. Берём и rerenderEntireTree суём в index.js чтобы отдать её state.js как колбек (когда вызываем какую то функцию и внутрь передаём другую функцию)

Реалзиация:

На даннный момент наше состояние (tate.js) это подобие ООПэшного объекта.

1. В state.js делаем метод (мы его будем перезаписывать потом на функцию колбек, которую передадим из индекса):

\_callSubscriber() {  
 ***console***.log('state changed');  
},

1. В state.js также создаём метод:

subscribe(observer) {  
 this.\_callSubscriber = observer;  
},

1. В index.js импортируем наш store:

import ***store*** from "./redux/state";

1. Дальше ключевыми в index.js будут:

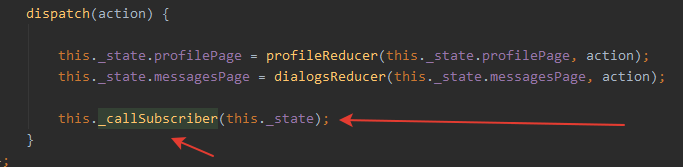
rerenderEntireTree(***store***.getState());

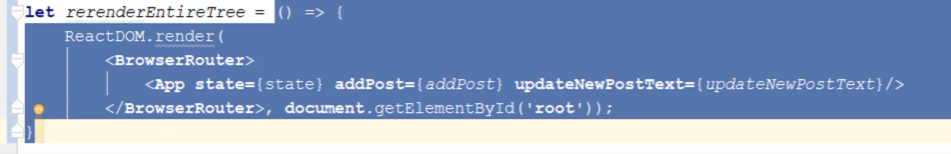
То есть перед перерисовкой UI – мы должны получить актуальный стейт.

***store***.subscribe(rerenderEntireTree);

Вызываем тут же (в index.js) функцию из стейта, которая чё сделает? Она ререндер с уже измененным стейтом (выше обновили) засунет как бы в state.js 

1. И потом мы уже сможем в диспатче (грубо говоря это основной метод для имземенения нашего UI) вызывать \_callSubscriber, передавая ему стейт:



То есть отдаём в «тот мир» фнкцию рендера: 

То что мы тут провернули – это патерн программирования obserrver («наблюдатель»).

36. про ООП поверхностно (зачем нам объекты)

Реакт вообще проповедует функциональное программирование, но есть элементы ООП.

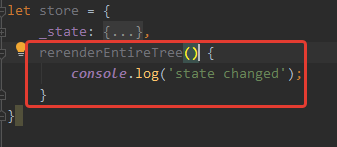
ООПэшный объект имеет свойства и методы, отличие от простых «тупых» объектов заключается в наличии методов.< Через свойства и методы мы с ним взаимодействуем.

This внутри методов объекта – это сам объект.

Инкапсуляиция – сокрытие деталей.

Олдскульная херня – когда свойство или метод начинается с нижнегго подчёркивания – значит разработчик не хочет чтобы пользовались этим во внешнем мире (типо через ссылку на объект), работать с ним через сеттер или геттер, другими словами это как бы приватный методод или свойство или типо это свойствои ил иметодо не является интерфейсом (нехуй типо с ним взаимодействовать).

Методы в ООПешном объекте пишут так:



37. store, state, ООП, рефакторинг

Подъёб с this - 

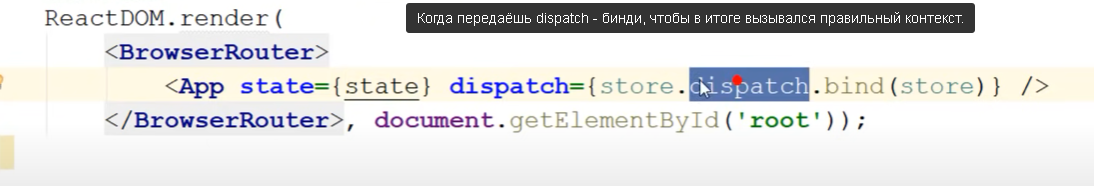
Короче там раз addPost прокидывается, а не сразу вызывается, получается прикол с контекстом.. типо потом уже в глубине в постах this оказывается равен пропсам, а не исходному объекту store. Чё делать? Забиндить метод (сделать так чтобы остался контекст исходного, начального метода): 

Bind короче возвращает равнозначную функцию, но внутри неё this всегда будет равен store.

38. dispatch и action

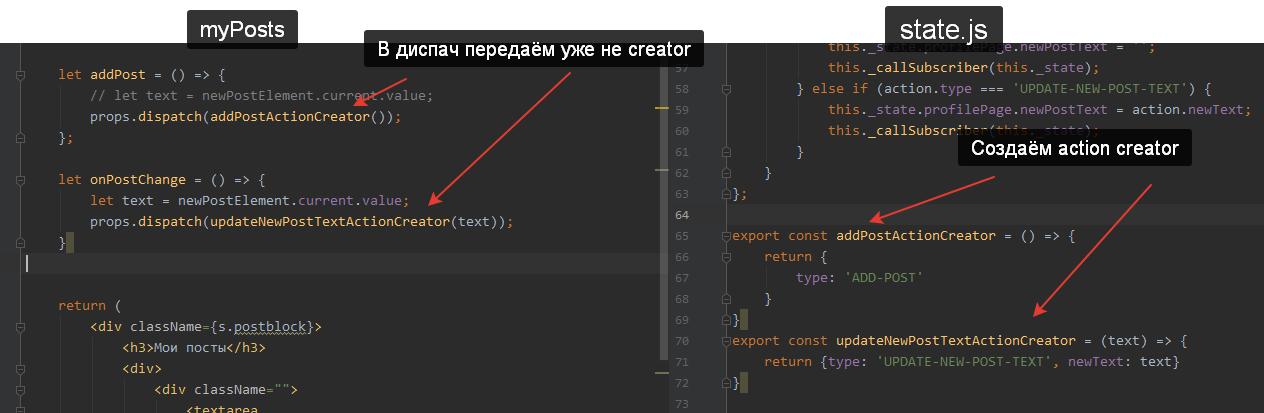
Dispatch – метод, который разработчики реакта придумали для того, чтобы через него вызывать все методы твоего store.

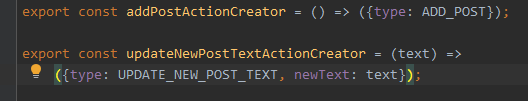
Action – объект, который описывает какое действие надо совершить. Мы его будем засовывать в dispatch чтобы понять какой метод вызывать. У actiona должно быть текстовое свойство type – в нем должно быть написано название действия. 

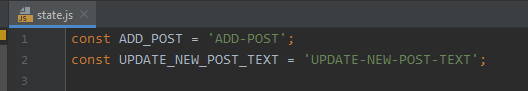
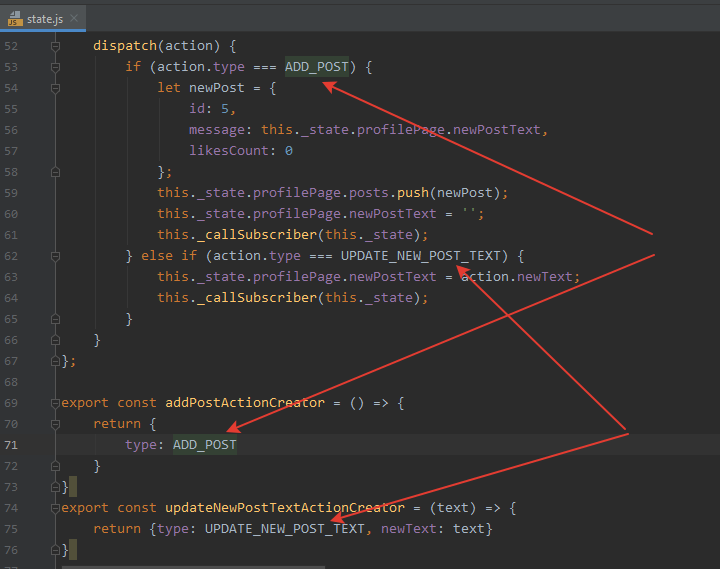


39 - action creator, action type

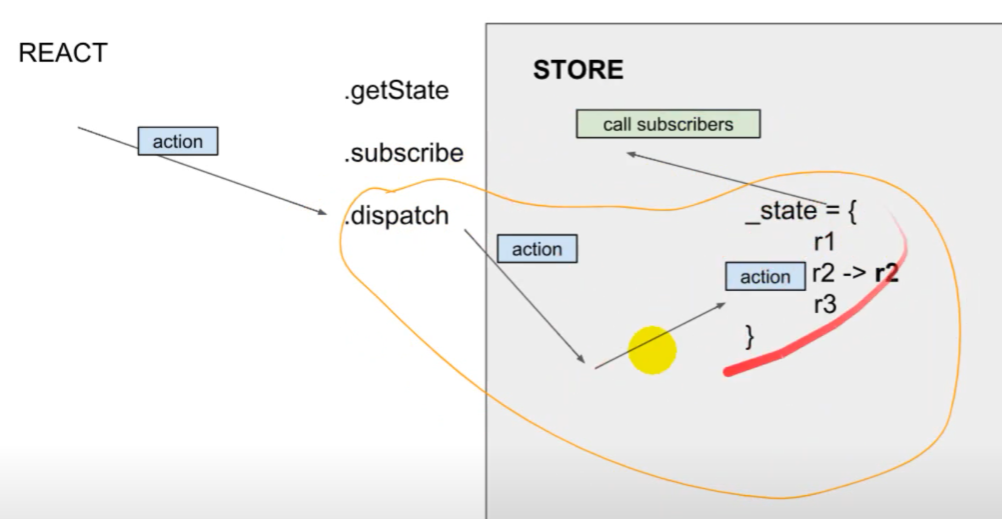
action creator, action type – используются для уменьшения вероятности ошибки.

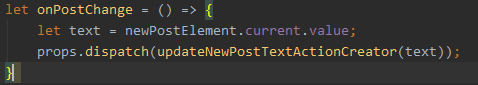
action creator – Функция, которая может принимать какие-то параметры и возвращает экшен (объект для метода dispatch, у которого есть как минимум свойство type). короче чтобы не наебаться с созданием объекта (типо свойство какое-нить не так назвал и тп) делают вот action Creator (названия позже изменим – типо пока для понимания так, но вроде так не называют ):: 

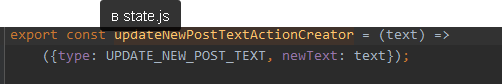
Стрелочные можно сократить: 

action type – Тупо выносим в костанты название типов:  

41. reducer

Как работает без редьюсеров: 

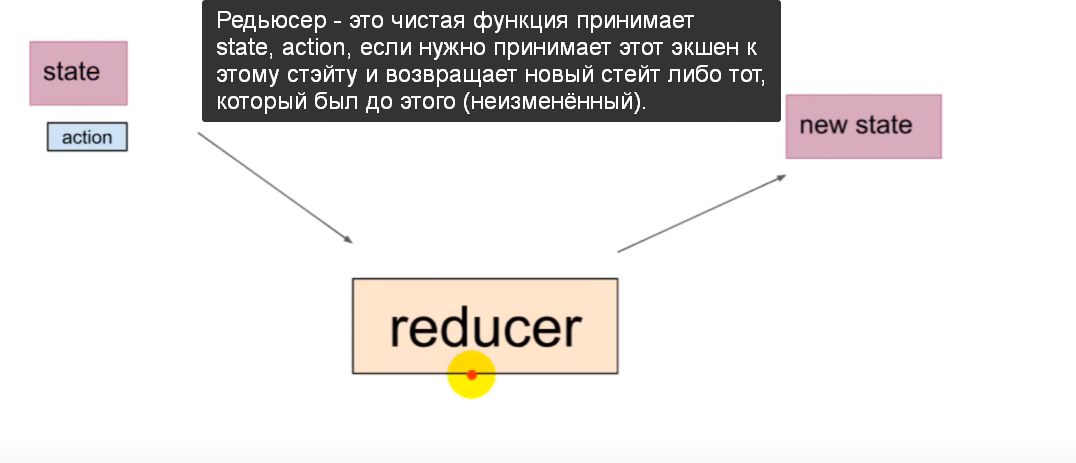
1. Action приходит из реакта, например на нашей textarea на OnChange стоит такая функция: 

Здесь в диспатч передаётся экшен (вернее тут action creator который как раз вернёт экшен) конкретно тут вот такой: 

Короче диспатчим экшен (конкретный экшен как правило изменяет конкретную подчасть в стэйте).

1. «Уведомляем» «подписчиков» (subscribe) которые подписались, чтобы подписчиики могли вызвать заново .getState, потому что стэейт изменился.

Вывод: логики получается очень много в диспатче, поэтому логику преобразования стейта не держат в диспатче, а используют редьюсеры.

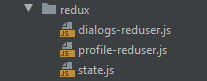


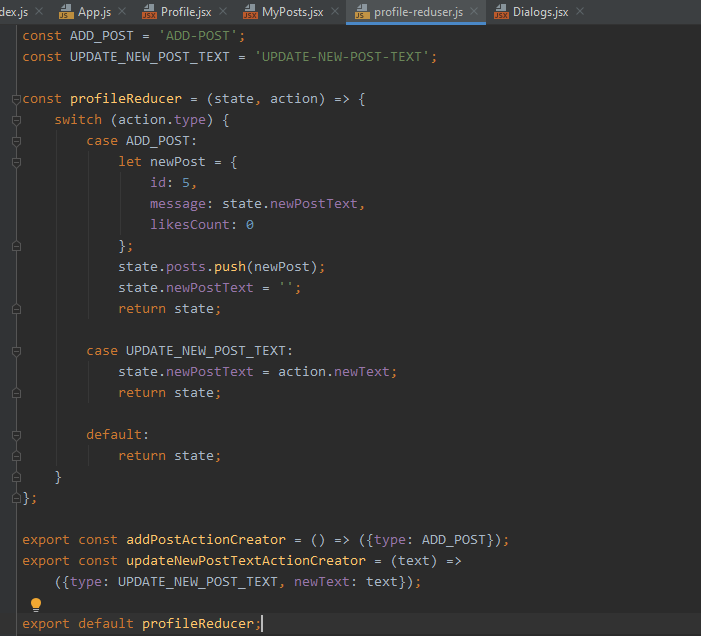
Где и сколько: Сколько у нас в объекте (state) подчастей – столько и редьюсеров создаём. Создаём в той же папке, где у нас был state.js. Дима называл через дефис, но вроде похер можно и через сс. 

Каждый редьюсер получает экшен. Момент: экшен один диспатчится и по фатку только одному редьюсеру нужен этот экшен, но store то этого не знает, ведь редьюсеры не внури стора, это отдельные функции, лежащие снаружи. Поэтому store по порядку отправляет каждому редьюсеру экшен, при этом экшена мало, редьюсеру ещё нужен state, но не весь, а каждый редьюсер получит свою составную подчасть.

Принцип такой, что редьюсер никого не «уведомляет» он просто должен как-то обработать state и вернуть его либо вернуть старый state.

Реализация:

1. Создаём файлы с редьюсерами в папке с нашим состоянием. 
2. Пишем сам редьюсер, пример:

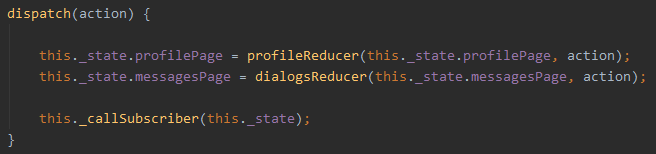


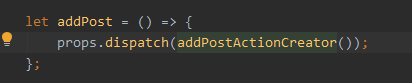
1. Делаем импорт в наше состояние (state.js):





1. В state у нас так и остаётся диспач, только теперь там мы пишем только редьюсеры и уведомляем подписчика:



1. И соответственно при обработке в UI – туда мы предварительно прокидываем через пропсы диспач и нужные нам данные, и потом на обработчике диспатчим с нужным нам редьюсером (в диспач передаётся экшен (создаётся через action creator, который лежит в файле с редьюсером) 

экшен это тупо объектик с простыми полями(могут быть разные в зависимости от того чё делает редьюсер) короче диспач запустился с экшеном(объектиком) ну и получается что диспач пробегается по всем редьюсерам (как я понимаю) передевая нужный стейт (типо не весь а только который нужен для того или иного редьюсера) : 

Получается в диспач передаёшь экшен, диспач внутри себя запускает редьюсеры используя переданный в диспач экшен ну и потом вызывает колбек (ререндер), который мы передали из index.js

или как Дима говорит уведомляем всех подписчиков.

42 – Redux

Redux - это библиотека, предназначенная для создания контейнеров, используемых для хранения состояния приложения. Она предлагает разработчику понятные инструменты для управления состоянием, которые ведут себя предсказуемо. Данная библиотека подходит как для приложений, написанных на чистом JavaScript, так и для проектов, при разработке которых использовались какие-нибудь фреймворки. Redux отличается маленькими размерами, но при этом позволяет писать надёжные приложения, работающие в различных средах. Дальше будет мой пример.

Установка: npm install redux

Реализация:

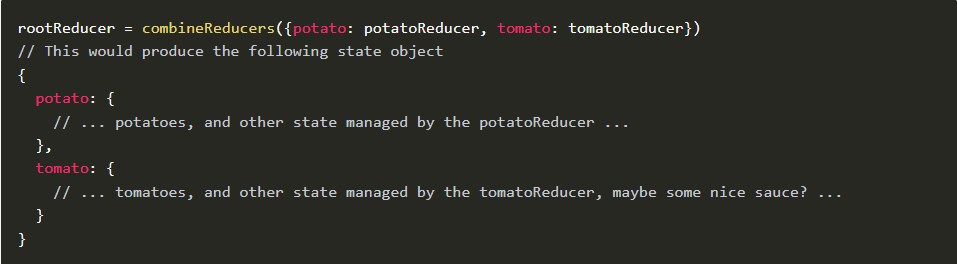
1. Создаём файлик например redux-store.js
2. Надо упаковать редьюсеры, есть спец. функция combineReducers.

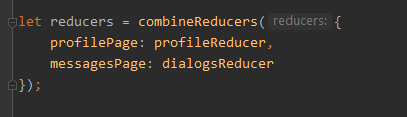
Комбинирование редьюсеров. Зачем? Когда наше приложение разрастается, хочется еще больше модульности, чтобы каждый кусочек кода отвечал за конкретную часть (для этого мы расхерачили на несколько редьюсеров) и с помощью combineReducers из пакета redux собрать их воедино. Причем, абсолютно никакой магии, combineReducers просто возвращает "составной" редьюсер.

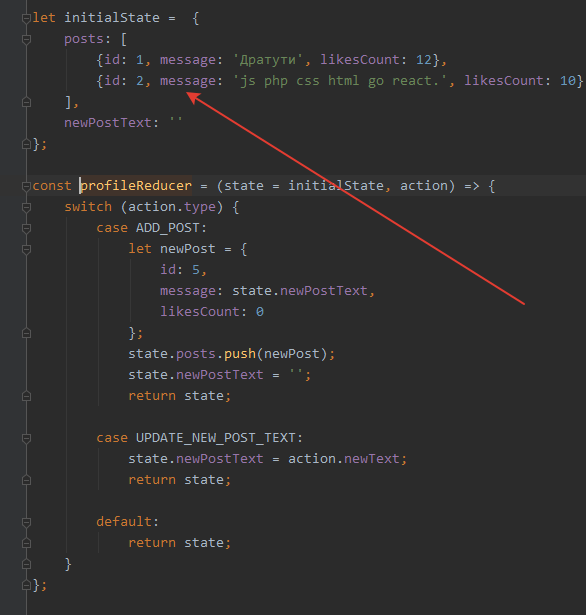
Вспомогательная функция combineReducers преобразует объект, значениями которого являются различные функции редьюсеры, в одну функцию редьюсер, которую можно передать в метод createStore. Результирующий редьюсер вызывает вложенные редьюсеры и собирает их результаты в единый объект состояния.

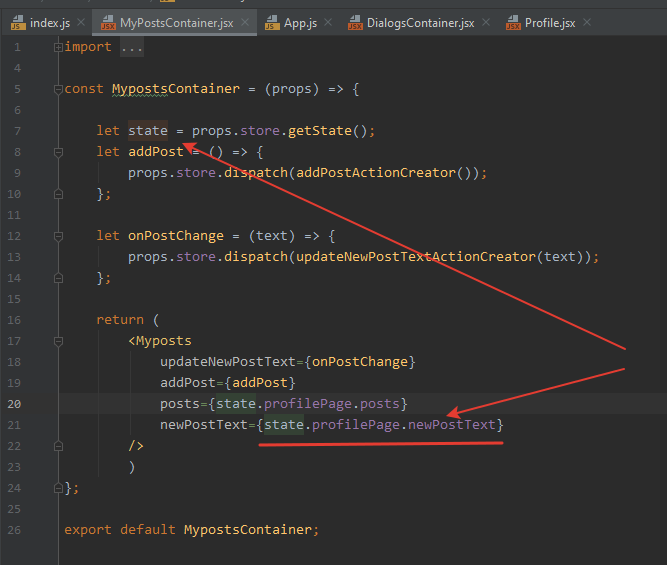
Состояние, созданное именами combineReducers(), сохраняет состояние каждого редьюсера под их ключами, переданные в combineReducers().

Пример:



Вот пример из урока: короче вот есть так у нас: 

К примеру profileReduxer выглядит так: 

И если мне обратиться к состоянию и вытаскивать из этого редьюсера буду делать так: 

Вы можете управлять именами ключей состояний, используя разные ключи для редюсеров в переданном объекте. Например, вы можете называть combineReducers({ todos: myTodosReducer, counter: myCounterReducer }) для формы состояния как {todos, counter}.

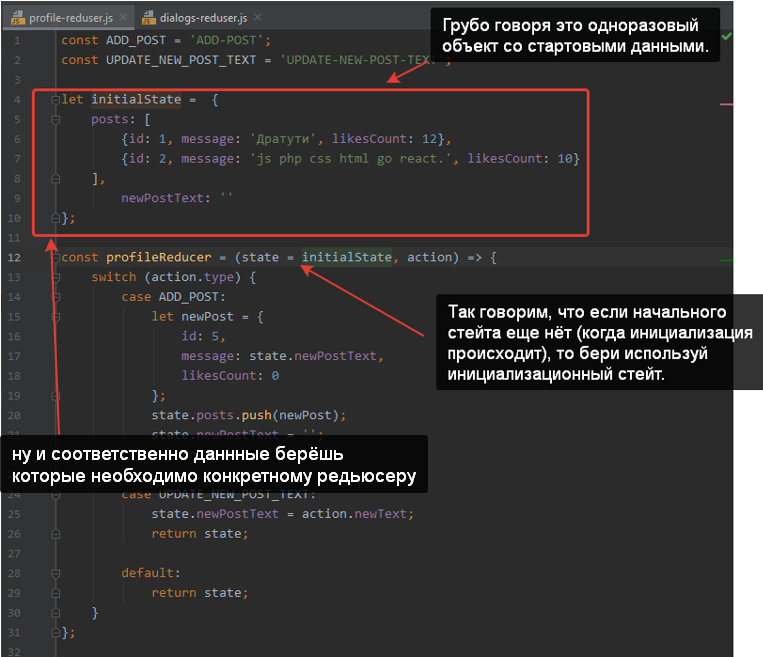
Популярное соглашение - назвать редюсеры после разделения состояний, которыми они управляют, поэтому вы можете использовать сокращенную обозначение свойства ES6: combReducers ({counter, todos}). Это эквивалентно написанию combReducers ({counter: counter, todos: todos}). Возвращает

( Function ): редюсер, который вызывает каждый редюсер внутри объекта reducers и создает объект состояния с той же формой. 

1. Делаешь импорт в index.js:

import store from "./redux/redux-store";

1. Нам надо каждый редьюсер проиницилизировать начальным значением:



1. Так ну и короче redux имеет те же методы, которые мы делали в нашем самописном store. То есть у него есть \_callSubscriber, getState и тп.

43. container component (контейнерная компонента)

Короче у нас (это у нас в этом курсе) все компоненты – функциональные (пока что по крайней мере).

Компоненты React часто делятся на 2 больших сегмента:презентационные компоненты и компоненты контейнера.

презентационные компоненты в основном связаны с созданием некоторой разметки для вывода. Они не управляют никаким состоянием, кроме состояния, связанного с презентацией (они тупые – принимают пропсы и возвращают какие-то данные). Нужно прокидывать в презентационные компоненты как можно меньше данных, чтобы компоненты оставались чистыми, презентационными, чтобы они ничего не знали про store. Чтобы в компоненте не было ничего лишнего. В идеале комонент должен получит данные и колбеки. Короче нехуй всё подряд пихать в презентационный компонент.

Компоненты контейнера в основном связаны с «внутренними» операциями. Они могут обрабатывать состояние различных подкомпонентов. Они могут заключать в себе несколько презентационных компонентов. Они могут взаимодействовать сRedux. Чтобы упростить различие, мы можем сказать презентационные компоненты связаны с внешним видом, Компоненты контейнера заботятся о том, чтобы все работало. Ей разрешено быть не тупой. Её задача отрисовать презентационный компонент и удолетворить его нужды (дать ей необходимые данные). Можно сказать задача её = это быть обёрткой для през комп. В контейнерную компоненту можно прокидывать прям весь store, главное чтобы эти «знания» были защищены от презентац комп.

18 мая – утро – начинаю с 43 урока с середины. (начался официальный 9-ый день обучения).

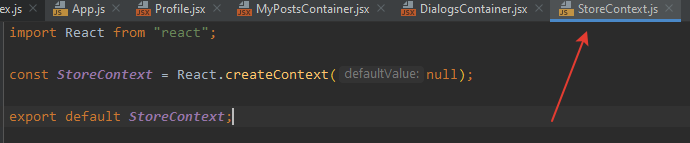
Короче там с переносом методов из презентациооной компоненты не всё так просто. Лучше это смотреть в проекте прям на примере файлов MyPostsContainer и MyPosts (как правильно убрать диспач из MyPosts и засунуть в контайнер и передать параметры и тп). Там например презент компонент получает если у себя весь store это тоже не правильно, нужно передать только то, что ему необходимо и не больше. Типо когда мы на примере MyPostsContainer

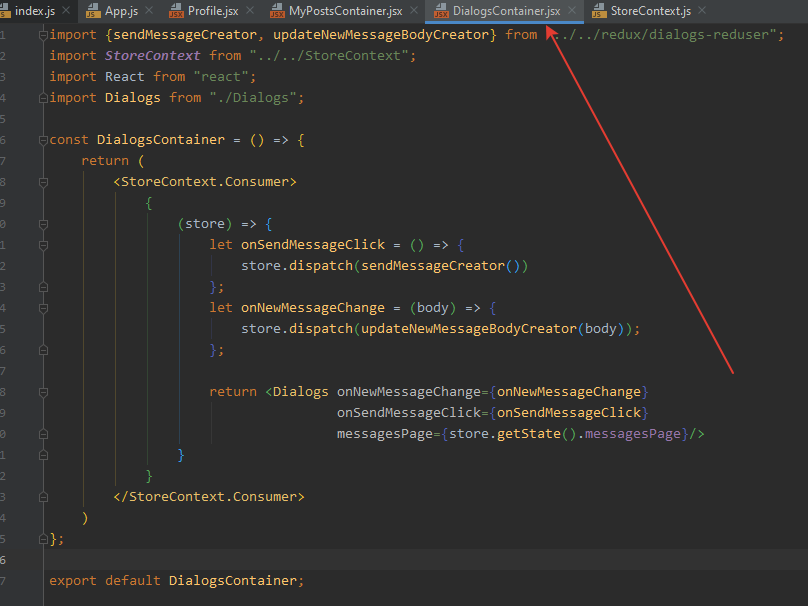
44. Context API (познакомиться и забыть)

Это уже фишечка redux. Установка: npm install redux

В типичном React-приложении данные передаются сверху вниз (от родителя к дочернему компоненту) с помощью пропсов. Однако, подобный способ использования может быть чересчур громоздким для некоторых типов пропсов (например, выбранный язык, UI-тема), которые необходимо передавать во многие компоненты в приложении. Контекст предоставляет способ делиться такими данными между компонентами без необходимости явно передавать пропсы через каждый уровень дерева.

В данном уроке дима убирал пропсы у компонентов (всех кроме презентационных). Короче суть такая что можно вот сделать через контекст – передать его контейнерной компоненте, но презентационная должна всё равно о контектсте ничего не знать.

Вкратце делал он так: 

Потом убрал все пропсы у компонент (всех, кроме презентационнных). И конт компонент выглядит примерно так: 

Но суть короче что это просто чтобы понять как там рабоатет внутри, сами мы конткест не будем делать, за нас это дело будет проворачивать реакт. Как расскажут поже).

45 - react-redux (connect, mapStateToProps, mapDispatchToProps)

О чём урок: Суть этого урока – передать контекст, только уже посредством библиотеки, а не самим писать велосипед как в прошлом уроке.

connect, mapStateToProps, mapDispatchToProps - всё вышеперечисленное это фишечки react-redux.

React-Redux — это официальный уровень привязки пользовательского интерфейса React для Redux . Он позволяет вашим компонентам React считывать данные из хранилища Redux и отправлять действия в хранилище для обновления состояния. Другими словами это спец прослойка между UI и BLL чтобы UI не возился с BLL.

Установка npm install react-redux

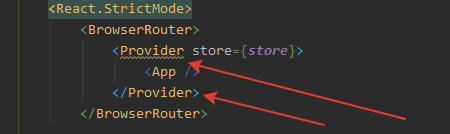
Реализация:

1. Нам надо прокидывать store через контекст. Для этого нужен Provider (Компонент <Provider> есть в react-redux и делает Redux store доступным для любых вложенных компонентов, которым требуется доступ к хранилищу Redux. Поскольку любой компонент React в приложении React Redux может быть подключен к хранилищу, большинство приложений будут отображать <Provider> на верхнем уровне все дерево компонентов приложения внутри него. Затем хуки (мы пока не учим) и connectAPI могут получить доступ к предоставленному экземпляру хранилища через механизм контекста React.)

(Надо импортировать в индексном файле):

 (у меня автоматом подтянулось import Provider from "react-redux/es/components/Provider"; - оставлю так пока)

1. обернуть наше App в Provider и передать ему store (сделали импорт из redux-store):

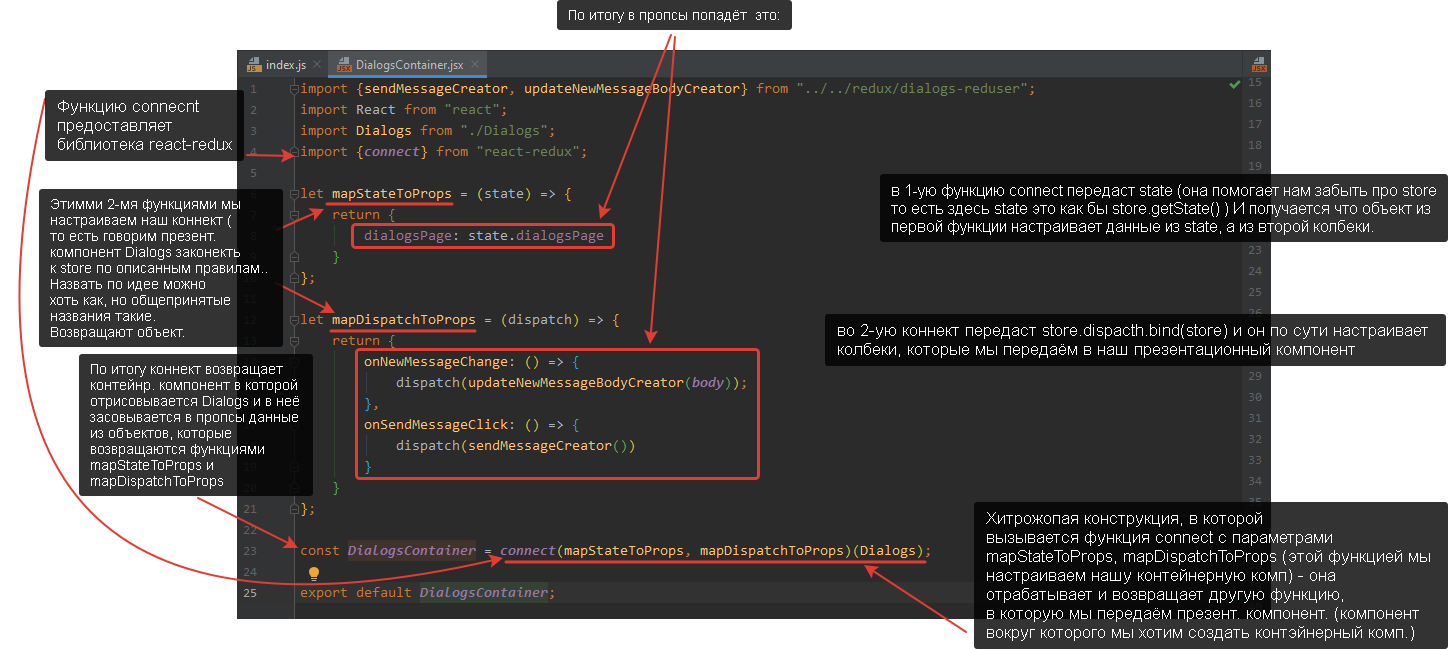


1. Дальше смотри у нас есть презентационный компонент и мы контейнерный компоннет создаём как бы только для того чтобы законектить презент компонент к данным из Store и прокинуть колбеки в презент. компонент.

Как контйнерной компоненте получить store? Создаём контэйнерную компоненту с помощью react-redux и её функции connect.

Connect – Создаёт контейнерный компонент и нужен для того, чтобы меньше возиться с прокидыванием.

Как работает Connect: коннект создаёт контейнерную компоненту, внутри этой контейнерной компоненты коннект рендерит презентационный компонент. И внутрь презентационной компоненты в качестве пропсов передаёт те свойства, которые сидят в mapStateToProps и mapDispatchToProps. Функция connect(), предоставляемая пакетом react-redux, может принимать до четырёх аргументов, каждый из которых является необязательным. После вызова функции connect() возвращается компонент высшего порядка, который можно использовать для оборачивания любого компонента React. Подробнее:



46. копия глубокая vs поверхностная (deep copy vs shallow copy)

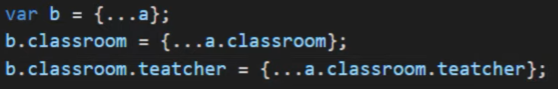
Короче объекты передаются по ссылке.

Два объекта никогда не равны, даже если в них одинаковые ключи и значения.

Есть примитивы и объекты. Вот короче примитивы передаются по значению. Массив если что тоже является объектом.

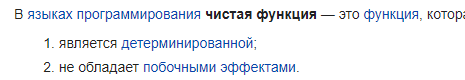
Есть синтаксис чтобы сделать копию объекта: let b = {...a};

Но в объекте могут быть свойства как примитивы, так и вложенные объекты: 

Дак вот при поверхностном копировании объекта вложенные объекты всё равно передадутся по ссылке, их типо надо копировать отдельно. 

47 - Делаем копию состояния в ProfileReducer (88 позырил)

Redux зависит от чистых функций. Чистота функции – это требование к некоторым частям системы в Реакте. Она требуется к компонентам, редьюсерам и селекторам.

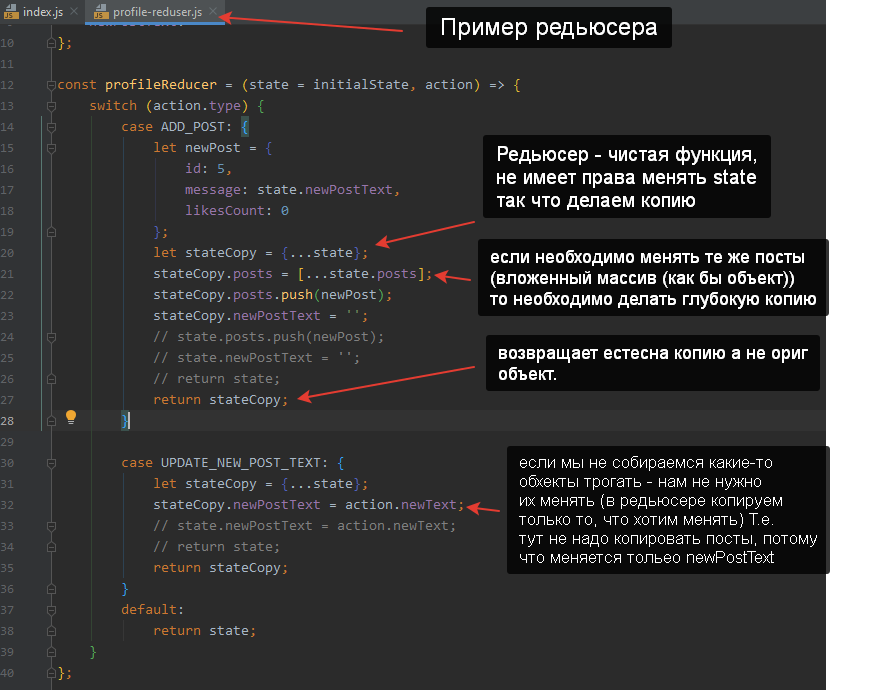
Чистая функция - 

На фундаментальном уровне, любая функция, которая не изменяет входные данные, не зависит от внешнего состояния (базы данных, DOM или глобальной переменной) и возвращает один и тот же результат для одинаковых входных данных является чистой функцией.

Характеристики чистых функций подробнее:

1. Имьютабельность (immutability –”неизменность” - не изменяет входные данные) – чистая функция не должна мутировать (менять) то, что в нее пришло. Например, пришли пропсы или другие параметры, то функция не должна их мутировать, иначе это будет side effect (“побочный эффект”). Проще говоря, функция должна пришедшие параметры обработать и вернуть что-то, не меняя самого “оригинала” пришедших параметров. К примеру, некоторые методы массивов меняют оригинал (можно сказать мутируюище методы) (splice, reverse и др.), а другие (slice) – создают новый массив. То значит первые мы не можем применять к оригиналу. Или например Редьюсеры – это тоже чистые функции. На примере них видно, что если нам нужно что-то все-таки изменить, то мы делаем для этого копию объекта ведь Redux принимает переданное в него состояние (state) и отдаёт его каждому reducer в цикле. Если есть какие-либо изменения он ожидает получить новый объект из reducer функции (также он рассчитывает получить старый объект обратно, если изменений не было), дальше Redux проверяет, изменилось ли старое состояние путём обычного сравнения, а если мы мутировали свойство старого объекта вместо создания нового, внутри нашего reducer, то старое и новое состояние будут ссылаться на один и тот же объект. Поэтому Redux думает, что ничего не изменилось! И ничего тебе не отрендерит.
2. Обязательно имеет return ( чтобы что-то возвращать). В реакт это обычно jsx.
3. Не должны иметь никаких side effects. К “побочным эффектам” относится то же изменение значений в глобальном мире, асинхронные запросы – типо потому что в глобальный мир обращение (это можно делать в методах жизненного цикла или внутри функции useEffect). По этой причине нельзя делать ajax-запросы в редьюсерах, но можно делать в Thunk-redux (в санках (они не чистые функции)).
4. Идемпотентность (Idempotence) – свойство возвращать на выходе всегда тот же самый результат при получении одних и тех же данных. Это свойство касается также и запросов (GET-запросы – идемпонетнны, а POST – нет).
5. Функция называется детерминированной тогда, когда для одних и тех же входных параметров она возвращает один и тот же результат. Например, функция, считающая количество символов, детерминированная

По простому: если функции два раза дать на обработку одно и то же значение, она всегда выдаст один и тот же результат и в программе не изменит ничего, что не относится непосредственно к этой функции. То есть у чистой функции предсказуемый результат и нет побочных эффектов. Или Чистая функция — это детерминированная функция, которая не производит побочных эффектов. Такая функция зависит только от своих входных аргументов и всегда ведёт себя предсказуемо.

Короче мы не имеем права менять существующий state, мы должны работая со стейтом делать его копию в редьюсерах. 

48. Делаем копию состояния в DialogReducer

Делали обновления state имьютабельным способом.





Эта ошибка устраняется вылезает на map и устраняется так:

let messageElements = props.messagesPage.messages.map*(*message => *<*Message message=*{*message.message*}* key=*{*message.id*}/>)*

Вот код после рефакторинга:

const dialogsReducer = *(*state = initialState, action*)* => *{* switch *(*action.type*) {* case UPDATE\_NEW\_MESSAGE\_BODY:  
 return *{* ...state,  
 newMessageBody: action.body  
 *}*;  
 case SEND\_MESSAGE:  
 let body = state.newMessageBody;  
 return *{* ...state,  
 newMessageBody: '',  
 messages: *[*...state.messages, *{*id:6, message: body*}]*,  
 *}*;  
 default:  
 return state;  
 *}  
}*;

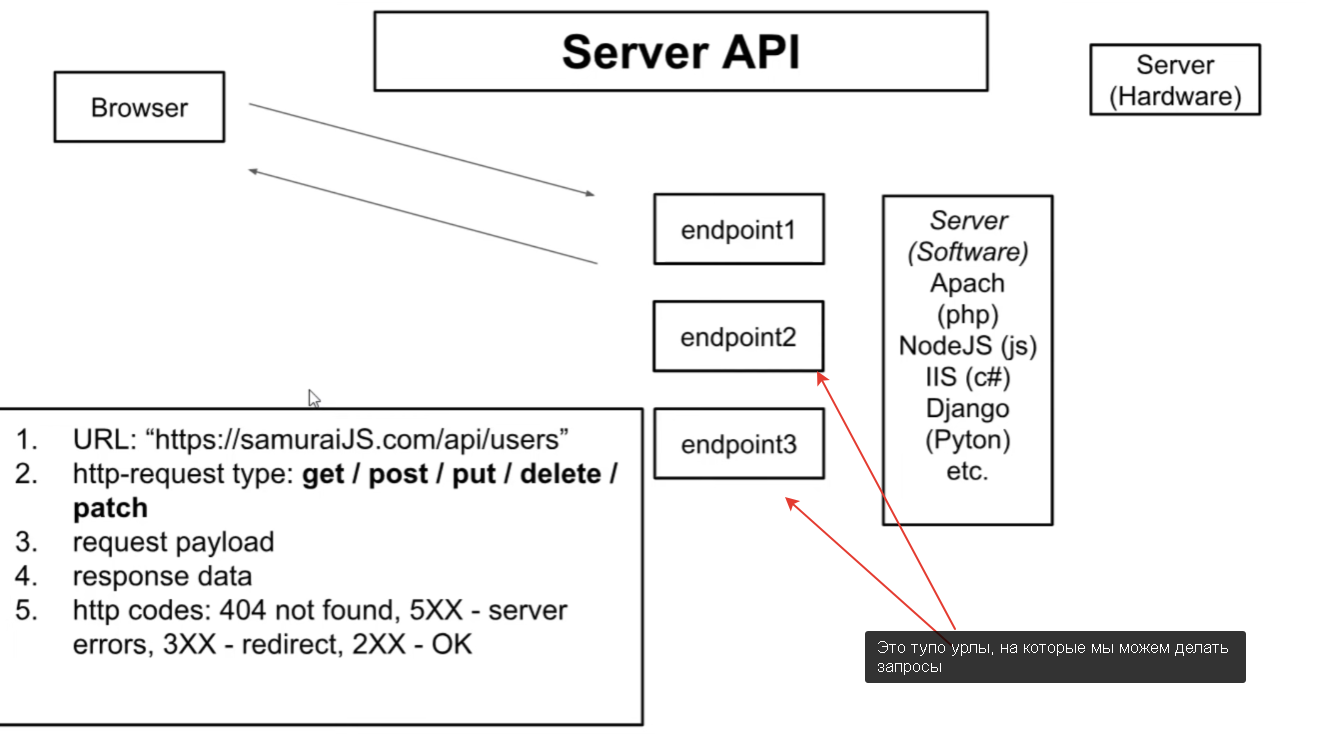
49. страница пользователей

50 - REST API (краткая теория)

API - программный интерфейс приложения, или интерфейс программирования приложений) — специальный протокол для взаимодействия компьютерных программ, который позволяет использовать функции одного приложения внутри другого. это совокупность инструментов и функций в виде интерфейса для создания новых приложений, благодаря которому одна программа будет взаимодействовать с другой. Это позволяет разработчикам расширять функциональность своего продукта и связывать его с другими. Простым языком это контракт, который предоставляет программа. «Ко мне можно обращаться так и так, я обязуюсь делать то и это.

API - это общий термин, который означает "программный интерфейс приложения". Rest API - это конкретный API, под названием REST, описывающий протокол взаимодействия с веб-сервисоv, это короче как набор правил по взаимодействию с сервером.

REST API - это способ взаимодействия сайтов и веб-приложений с сервером. Его также называют RESTful. Representational State Transfer (REST) в переводе — это передача состояния представления. Если проще, то REST API — это когда серверное приложение дает доступ к своим данным клиентскому приложению по определенному URL.



51 - Users API

Тут сервачок димин https://social-network.samuraijs.com/

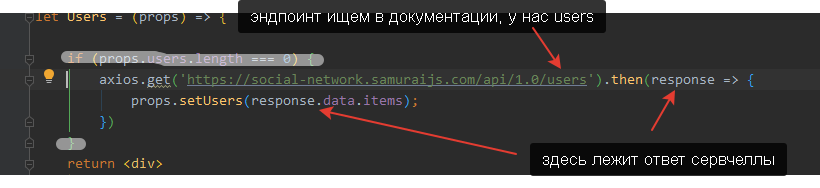
Юзать будем axios. Axios – это библиотека с открытым исходным кодом, позволяющая делать HTTP-запросы. Она предоставляет методы .get(), .post() и .delete().

Установка Axios: Npm install axios

Любая серверная апишка имеет базовый url, например: 

И вот к этому базовому урлу мы уже цепляем приписки (к примеру /users).

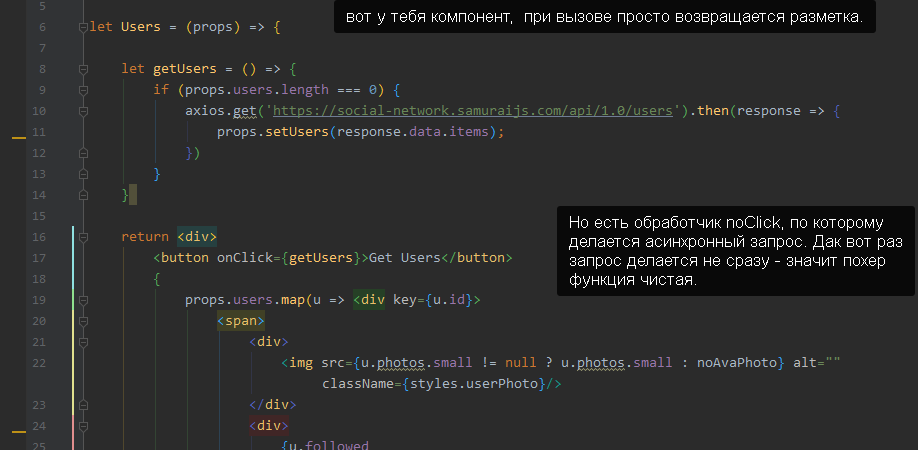
Реализация get запроса на сервер:

1. Там где используем делаем импорт 
2. 

Изи.

52 - Side Effect

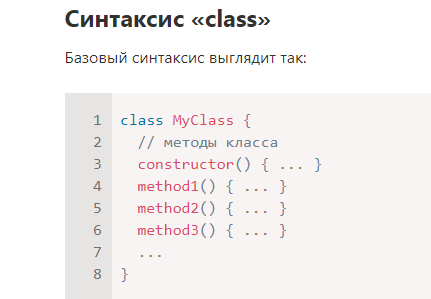
Короче если функция делает какие-то запросы на сервер – это не чистая функция, но здесь момент, она вот вызвалась вернула разметку и при этом ни куда не послаал запрос – да она чистая, но внутри неё может сидеть логика с запросом – просто при вызове функции эта логика не используется. Пример:

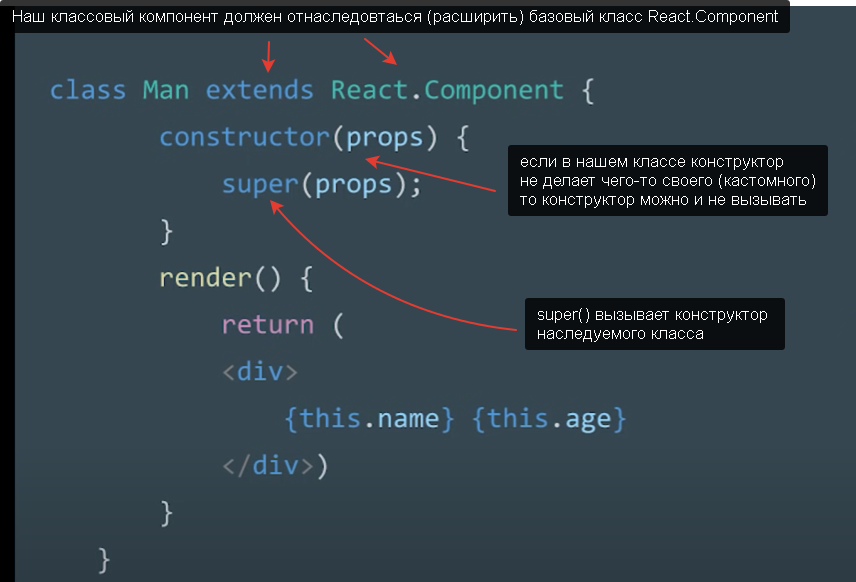


53 - классовый компонент (extends React.Component}

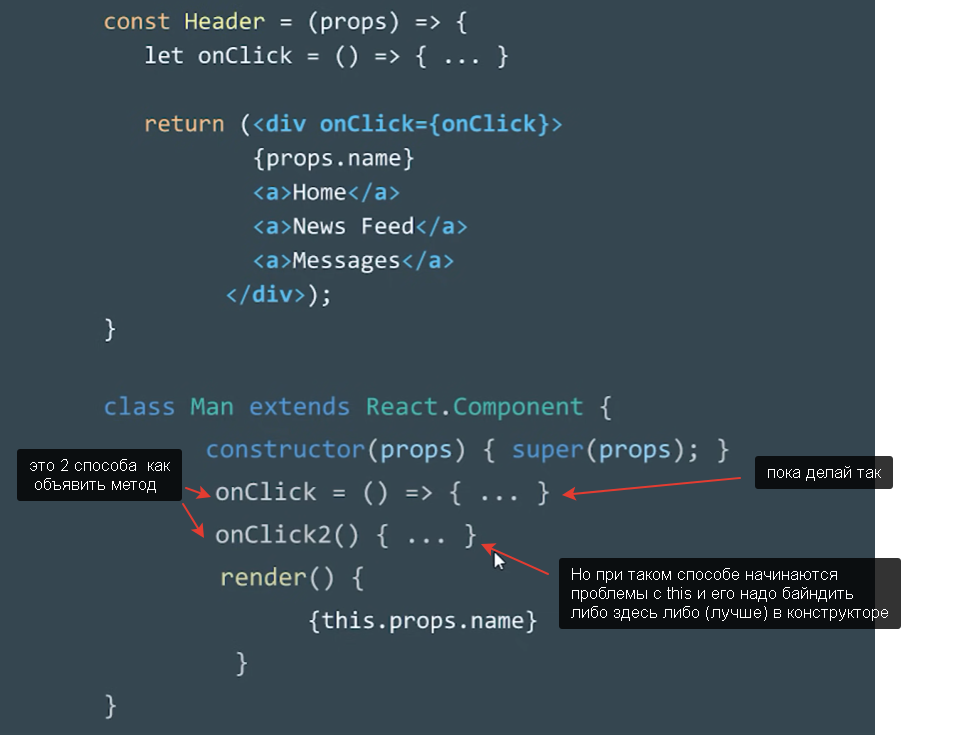
React позволяет определять компоненты как классы или функции.

*В объектно-ориентированном программировании*класс*– это расширяемый шаблон кода для создания объектов, который устанавливает в них начальные значения (свойства) и реализацию поведения (методы).*



Затем используйте вызов new MyClass() для создания нового объекта (экземплара класа, инстанца) со всеми перечисленными методами. При этом автоматически вызывается метод constructor(), в нём мы можем инициализировать объект. 

Единственный обязательный метод в подклассе React.Component — [render()](https://ru.reactjs.org/docs/react-component.html" \l "render) (он не делает перерисовки он как бы возвращает просто разметку и реакт уже потом сам думает чё с ней сделать). Все остальные методы являются необязательными.



54 - жизненный цикл, life cycle methods, componentDidMount

В приложениях со множеством компонентов очень важно освобождать используемые системные ресурсы, когда компоненты удаляются.

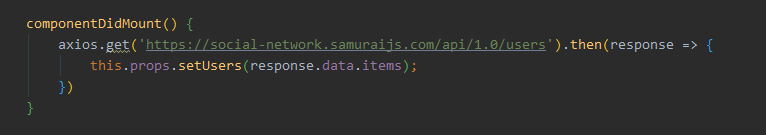
Первоначальный рендеринг компонента в DOM называется «монтирование» (mounting).

componentDidMount – Метод componentDidMount() запускается после того, как компонент отрендерился в DOM, именно здесь мы делаем запросы на сервак и вообще все side эффеты.

Каждый компонент имеет несколько «методов жизненного цикла». Переопределение такого метода позволяет выполнять код на конкретном этапе этого процесса.

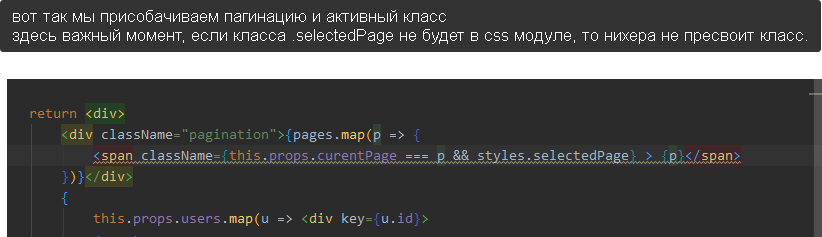
Каждый раз когда DOM-узел, созданный компонентом, удаляется, происходит «размонтирование» (unmounting).



В компоненте Users поменяли запрос к серваку в метод ComponentDidMount 

55 - pagination, постраничный вывод пользователей

Всё, что меняется, должно быть в state – потому что компонента не может по идеологии перерендериться без изменения стейта (flux).



Небольшое отступление, а то я хуёво понимаю уже. Короче как рабоатет:

Есть index.js – в нём

Установки:

Create react app workProject

Npm install redux

npm install react-redux

npm install axios