ROADMAP



Введение в JSX

ЭЛЕМЕНТЫ (ОНИ НЕ ИЗМЕНЯЕМЫ!!!)

Создав однажды элемент, вы не сможете изменить его дочерние элементы или атрибуты. Элемент похож на один кадр в фильме: он представляет собой пользовательский интерфейс в определённый момент времени.

**Элементы — это то, что из чего компоненты «сделаны»**

Самый простой пример кода для React:

ReactDOM.render(

<h1>Привет, мир!</h1>,

document.getElementById('root')

);

Рассмотрим объявление такой переменной.

const ***element*** = <h1>Hello world</h1>

Такой код называется JSX, и это расширение синтаксиса JavaScript. Он не обязателен для JS, Но оч рекомендуем.

В приведённом ниже примере мы объявляем переменную с именем name, а затем используем ее внутри JSX, обернув ее в фигурные скобки:

const ***name*** = "Josh Perez"  
const ***element*** = <h1>Hello, {***name***}</h1>  
  
ReactDOM.render(  
 ***element***,  
 ***document***.getElementById('root')  
);

В фигурных скобках JSX вы можете поместить любое корректное [выражение JavaScript](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Expressions_and_Operators#Expressions).

Т.е. мы создали переменную name, const, использовали данные из переменной, чтобы отрисовать element

Может сделать так:

function formatName(user) {  
 return user.firstName + " " + user.lastName;  
}  
  
const ***user*** = {  
 firstName: "Harper",  
 lastName: "Perez"  
}  
  
const ***element*** = (  
 <h1>Hello, {formatName(***user***)}!</h1>  
);  
  
ReactDOM.render(  
 ***element***,  
 ***document***.getElementById('root')  
)

После компиляции выражения JSX становятся обычными вызовами функций JavaScript и вычисляются в объекты JavaScript.

Это означает, что вы можете использовать JSX внутри операторов if иfor, присваивать его переменным, принимать его в качесиве аргументов и возвращать из функций:

**Установка атрибутов с помощью JSX**

Вы можете использовать кавычки для указания строковых литералов в качестве атрибутов:

Вы также можете использовать **фигурные скобки для вставки JavaScript**-выражения в атрибут:

const ***avatars*** = {  
 avatarUrl: "hhtps://mail.ru"  
}  
  
const ***element*** = <img src={***user***.avatarUrl}></img>

Имена свойств - camelCase

Теги JSX могут содержать **дочерние элементы:**

const ***element*** = (  
 <div>  
 <h1>Hello!</h1>  
 <p>Nice to meet you!</p>  
 </div>  
)

**Babel компилирует JSX в вызовы React.createElement().**

ОТРИСОВКА ЭЛЕМЕНТОВ

Чтобы отрисовать элементы, нужно поместить их в корневой узел ReactDOM, например

переменная НЕ ВСПЛЫВАЕТ!!!

const ***element*** = (  
 <div>  
 <h1>Hello!</h1>  
 <p>Nice to meet you!</p>  
 </div>  
)  
  
ReactDOM.render(  
 ***element***,  
 ***document***.getElementById('root')  
)

Компоненты не изменяемы, поэтому единственный способ обновить содержимое корневого узла root - это вызвать reactDOM.render() определенное количество раз. Например:

function tick() {  
 const element = (  
 <div>  
 <h1>Привет, мир!</h1>  
 <h2>Сейчас {new ***Date***().toLocaleTimeString()}.</h2>  
 </div>  
 );  
 ReactDOM.render(element, ***document***.getElementById('root'));  
}  
  
setInterval(tick, 1000);

DOM React сравнивает элемент и его дочерние элементы с предыдущими и применяет только обновления DOM, необходимые для преобразования DOM в желаемое состояние.

Несмотря на то, что мы создаём элемент, описывающий всё дерево пользовательского интерфейса на каждом тике, только текстовый узел, содержимое которого изменилось, обновляется в DOM React.

COMPONENTS

**Компоненты** позволяют разделить пользовательский интерфейс на независимые, повторно используемые части и работать с каждой из частей отдельно.

**Концептуально компоненты похожи на функции JavaScript.** Они принимают произвольные входные данные (**называемые «props» или свойствами**) и возвращают React-элементы, описывающие, что должно появиться на экране.

***Функциональные компоненты***

Самый простой способ определить компонент — написать JavaScript-функцию:

function Welcome(props) {

return <h1>Привет, {props.name}</h1>;

}

такие компоненты мы называем «функциональными», потому что они являются буквально **функциями JavaScript.**

***Классовые компоненты***

class Welcome extends React.Component {

render() {

return <h1>Привет, {this.props.name}</h1>;

}

}

Два вышеуказанных компонента эквивалентны с точки зрения React.

ОТРИСОВКА КОМПОНЕНТОВ

Когда React видит элемент, представляющий **пользовательский компонент**, он передаёт JSX-**атрибуты** этому компоненту **в виде единственного объекта**. Мы называем этот объект **«props».**

Например, этот код отображает «Привет, Сара» на странице:

function Welcome(props) {  
 return <h1>Hello, {props.name}</h1>  
}  
  
const ***element*** = <Welcome name="Sara"/> //пользовательский компонент передает атрибуты этого компонента в виде объекта  
  
ReactDOM.render(***element***, ***document***.getElementById('root'))

Давайте посмотрим, что происходит в этом примере:

1. Мы вызываем ReactDOM.render() с элементом <Welcome name="Сара" />.
2. React вызывает компонент Welcome с объектом {name: 'Sara'} как props.
3. Наш компонент Welcome возвращает элемент <h1>Hello, Sara</h1> в качестве результата.
4. React DOM эффективно обновляет DOM, чтобы соответствовать <h1>Hello, Sara</h1>.

**Примечание: Всегда именуйте компоненты с заглавной буквы.**

Компоненты могут ссылаться на другие компоненты в своём выводе.

Порядок передачи данных в компонент:

1. Например, мы создали объект с данными:

const ***comment*** = {  
 date: new ***Date***(),  
 text: "Hope, you like React",  
 author: {  
 name: "Hello Kitty",  
 avatarUrl: 'https://placekitten.com/g/64/64'  
 }  
}

Нам нужно использовать эти данные для их представления в функциональном компоненте.

1. Вызов рендера ReactDOM.render(). Ниже вызов без данных

ReactDOM.render(<Comment />, ***document***.getElementById('root'))

При вызове рендера мы передаем в отрисовку компонент, чтобы передать в этот компонент данные из объекта comment нам нужно сделать следующее.

1. Используя атрибуты компоненты <Comment/>, мы КАК БЫ СОЗДАЕМ НОВЫЙ ОБЪЕКТ “PROPS” из атрибутов. Атрибуты и значения атрибутов, например name = {object.name} будут ключами-значениями передаваемого в компонент данных.

Т.е. у нас есть объект:

const ***comment*** = {  
 date: new ***Date***(),  
 text: "Hope, you like React",  
 author: {  
 name: "Hello Kitty",  
 avatarUrl: 'https://placekitten.com/g/64/64'  
 }  
}

на его основе мы создаем новый объект из ключей-значений:

ReactDOM.render(<Comment   
 date={***comment***.date}   
 text={***comment***.text}   
 author={***comment***.author}/>, ***document***.getElementById('root'))

(Атрибуты это будут ключ-значения нового объекта)ю Мы не видим создание этого объекта.

const ***props*** = {  
 date: ***comment***.date,  
 text: ***comment***.text,  
 author: ***comment***.author  
}

Такой объект props затем передается в компонент для отрисовки, т.е.

значение **props.date** ссылается на значение **comment.date**

значение **props.author.avatarUrl** ссылается на значение **comment. author.avatarUrl**

**и т.д.**

function Comment(props) {  
 return (  
 <div className="Comment">  
 <div className="UserInfo">  
 <div className="Avatar">  
 <img src={props.author.avatarUrl} alt={props.author.name}/>  
 <div className="UserInfo-name">  
 {props.author.name}  
 </div>  
 </div>  
 <div className="Comment-text">  
 {props.text}  
 </div>  
 </div>  
 <div className="Comment-date">  
 {formatDate(props.date)}  
 </div>  
 </div>  
 );  
}

ИЗВЛЕЧЕНИЕ КОМПОНЕНТОВ

Можно разделать компоненты на более мелкие компоненты

Например, у нас есть такой компонент

function Comment(props) {  
 return (  
 <div className="Comment">  
 <div className="UserInfo">  
 <div className="Avatar">  
 <img src={props.author.avatarUrl} alt={props.author.name}/>  
 <div className="UserInfo-name">  
 {props.author.name}  
 </div>  
 </div>  
 <div className="Comment-text">  
 {props.text}  
 </div>  
 </div>  
 <div className="Comment-date">  
 {formatDate(props.date)}  
 </div>  
 </div>  
 );  
}

Мы можем разделить его на части,

извлечем Avatar и другие компоненты

КЛАССОВЫЕ КОМПОНЕНТЫ

Компоненты, определённые как классы, имеют некоторые дополнительные возможности. **Локальное состояние** — это как раз одно из них: эта возможность доступна только классам.

Преобразуем функциональный компонент:

function Clock(props) {  
 return (  
 <div>  
 <h1>Hello, world</h1>  
 <h2>Now is {props.date.toLocaleTimeString()}</h2>  
 </div>  
 )  
}  
  
function tick() {  
 ReactDOM.render(  
 <Clock date={new ***Date***()}/>,  
 ***document***.getElementById('root')  
 )  
}  
  
setInterval(tick, 1000)

В классовый компонент.

***Преобразование функции в класс:***

Преобразовать функциональный компонент, такой как Clock, в классовый компонент можно за пять шагов:

1. Создать ES6-класс с тем же самым именем, который расширяет React.Component.
2. Добавить к нему пустой метод render().
3. Перенести тело функции в метод render().
4. Заменить props на this.props в теле render().
5. Удалить оставшиеся пустое объявление функции.

// БЫЛО

function Clock(props) {  
 return (  
 <div>  
 <h1>Привет, мир!</h1>  
 <h2>Сейчас {props.date.toLocaleTimeString()}.</h2>  
 </div>  
 );  
}

//СТАЛО

class Clock extends React.Component {  
 render() {  
 return (  
 <div>  
 <h1>Hello, world</h1>  
 <h2>Now is {this.props.date.toLocaleTimeString()}</h2>  
 </div>  
 );  
 }  
}

**Clock** теперь определён как класс, а не функция.

Теперь добавим СОСТОЯНИЕ к классу CLOCK

1. Заменить this.props.date на this.state.date в методе render():

class Clock extends React.Component {  
 render() {  
 return (  
 <div>  
 <h1>Hello, world</h1>  
 <h2>Now is {this.state.date.toLocaleTimeString()}</h2>  
 </div>  
 );  
 }  
}

1. Добавить конструктор класса, который устанавливает начальное состояние в this.state.

Обратите внимание, что мы передаём props базовому (родительскому) конструктору!

class Clock extends React.Component {  
 constructor(props) {  
 super(props);  
 this.state = {date: new ***Date***()}  
 }  
  
 render() {  
 return (  
 <div>  
 <h1>Hello, world</h1>  
 <h2>Now is {this.state.date.toLocaleTimeString()}</h2>  
 </div>  
 );  
 }  
}

1. Удалить свойство date из элемента <Clock />:

Позже мы добавим код таймера обратно к самому компоненту.

ReactDOM.render(  
 <Clock />,  
 ***document***.getElementById('root')  
 )

Итого получаем(часы статичны, не изменяются) :

class Clock extends React.Component {  
 constructor(props) {  
 super(props);  
 this.state = {date: new ***Date***()}  
 }  
  
 render() {  
 return (  
 <div>  
 <h1>Hello, world</h1>  
 <h2>Now is {this.state.date.toLocaleTimeString()}</h2>  
 </div>  
 );  
 }  
}  
  
ReactDOM.render(  
 <Clock/>,  
 ***document***.getElementById('root')  
)

***Добавление методов жизненного цикла в класс***

В приложениях с множеством используемых компонентов очень важно освобождать ресурсы, занятые при их удалении.

Мы хотим **настроить таймер всякий раз**, когда Clock отрисовывается в DOM в первый раз. Это **называется «монтированием» (установкой) в React.**

Мы также хотим **сбрасывать этот таймер** всякий раз, когда DOM, созданный Clock, удаляется. Это **называется «размонтированием» в React.**

Мы можем объявить специальные методы в классе-компоненте для выполнения кода, когда компонент устанавливается и удаляется:

componentDidMount()

componentWillMount()

Это называется **хуками (методами)** жизненного цикла.

Хук **componentDidMount()** запускается **после** того, как вывод компонента отрисован в DOM. Это хорошее место для установки таймера.

Обратите внимание, что мы сохраняем идентификатор таймера в this.

componentDidMount() {  
 this.timerID = setInterval(() => this.tick(), 1000);  
}

Мы удалим таймер в хуке жизненного цикла **componentWillUnmount():**

componentWillMount() {  
 clearInterval(this.timerID)  
}

Наконец, реализуем метод tick(), который компонент Clock будет запускать каждую секунду.

Он будет использовать this.setState() для планирования обновлений локального состояния компонента:

tick() {  
 this.setState({  
 date: new ***Date***()  
 });  
}

Давайте быстро повторим, что происходит, а также перечислим порядок, в котором вызываются методы:

1. Когда <Clock /> передаётся ReactDOM.render(), React вызывает конструктор компонента Clock. Так как Clock должен отображать текущее время, он инициализирует this.state с объектом, включающим текущее время. Позднее мы обновим это состояние.
2. Затем React вызывает метод render() компонента Clock. Вот как React узнаёт, что должно отображаться на экране. Потом React обновляет DOM, чтобы он соответствовал выводу отрисовки Clock.
3. Когда в DOM вставлен вывод Clock, React вызывает хук жизненного цикла componentDidMount(). Внутри него компонент Clock указывает браузеру настроить таймер для вызова метода tick() компонента один раз в секунду.
4. Каждую секунду браузер вызывает метод tick(). Внутри него компонент Clock планирует обновление пользовательского интерфейса, вызывая setState() с объектом, содержащим текущее время. Благодаря вызову setState() React знает, что состояние изменилось, и снова вызывает метод render(), чтобы узнать, что должно отображаться на экране. На этот раз this.state.date в методе render() будет другим, и поэтому вывод отрисованного компонента будет включать обновлённое время. React обновляет DOM соответствующим образом.
5. Если компонент Clock когда-либо удаляется из DOM, React вызывает хук жизненного цикла componentWillUnmount(), чтобы оставить таймер.

***Правильное использование состояния setState()***

Есть три детали о setState(), про которые нужно знать.

1. Нельзя изменять состояние напрямую

// Неправильно

this.state.comment = 'Привет';

// Правильно

this.setState({comment: 'Привет'});

1. Обновления состояния могу быть асинхронными

React может выполнять несколько вызовов setState() за одно обновление для лучшей производительности.

Поскольку this.props и this.state могут обновляться асинхронно, вы не должны полагаться на их значения для вычисления следующего состояния.

// Неправильно

this.setState({

counter: this.state.counter + this.props.increment,

});

// Правильно

this.setState((state, props) => ({

counter: state.counter + props.increment

}));

// Правильно (без стрелочной функции)

this.setState(function(state, props) {

return {

counter: state.counter + props.increment

};

});

1. Обновления состояния объединяются

Когда вы вызываете setState(), React объединяет объект, который вы предоставляете c текущим состоянием.

## Однонаправленный поток данных

Ни родительский, ни дочерний компоненты не могут знать, является ли какой-либо компонент с состоянием или без него, и им не важно, определен ли он как функция или класс.

Вот почему состояние часто называют локальным или инкапсулированным. Оно недоступно для любого компонента, за исключением того, который владеет и устанавливает его.

10. Default опасность

**Export default <name>** может создать путаницу, т.к. при импорте <name> может называться произвольно. Очень важно правильно указать путь, откуда мы импортируем!

При импорте

**import <name> from ‘./folder/file.ts’** важно правильно указать имя и путь!

Импорт компонент можно делать автоматически:

**ALT + Enter** - автоимпорт компонент в WebStorm. При автоимпорте **проверяй path!!!**

11. Соц сеть, шаг 1, верстка, grid

**Делаем верстку на любой технологии, можно на FLEX, GRID, FLOAT.**

**В дано случае будем делать верстку на GRID**

12. Разбивка на компоненты.

Теперь разбиваем нашу разметку на компоненты.

Для этого создадим отдельную папку, в ней отдельные файлы, название пишем **с большой буквы!**

**Не забываем делать импорт-экспорт.**

13. Git

**Ссылка на репозиторий Димыча (там есть TYPESCRIPT):**

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

**Модули node** не закидываются в репозиторий, автоматом прописывается в GITIGNORE! Но в файл package.json сохраняются все зависимости! Из него node Подтягивает инфо какие модули установить.

Пишем NPM install для установки зависимостей, т.к. при создании проекта их нет

14. CSS Modules

Лучше создавать компоненты, и к ним писать отдельные CSS стили.

В файле компоненты нуно указывать импортируемый стиль CSS, например:

import './Navbar.css'**;**

Но, даже без импорта при совпадении стилей, они будут влиять на классы с таким же названием. Поэтому, если нужно, чтобы CSS стили были прописаны только для конкретной компоненты, в имени файла нужно указывать module, например:

**Navbar.module.css**

При этом все уникальные className, которые генерирует React, содержатся в объекте **classes** в виде:

(чтобы увидеть этот объект, его можно вывести в console.log(classes) )

{

’именование классов, которое я придумал’ : ’именование классов, которые сгенерировал React’,

’именование классов, которое я придумал’ : ’именование классов, которые сгенерировал React’,

…

}

Поэтому, в файле компонента пишем следующее:

- Для импорта объекта стилей:

import classes from './Profile.module.css'**;**

- Для присваивания классов в className вносим следующие значения (classes.posts, например):

export function Profile() {  
 return (  
 <div className="content">  
 <div>  
 <img  
 src="https://i.siteapi.org/no2IMQtVA-Jko-HOxCRaQpEC3FE=/fit-in/900x1000/center/top/filters:format(png)/5045b586a05f134.s.siteapi.org/img/d35c0d1ac7c8dd12e1e4146f14df612b9c31a4a3.jpg"  
 alt="nature"/>  
 </div>  
 <div>Ava + description</div>  
 <div>My posts  
 <div>New post</div>  
 <div className={classes.posts}>  
 <div className={classes.item}>Post1</div>  
 <div className={classes.item}>Post2</div>  
 </div>  
 </div>  
 </div>  
 )  
}

Чтобы дать тегу несколько классов, используем ШАБЛОННЫЕ СТРОКИ:

**className = {`${classes.nav} ${classes.active}`}**

Для аналога стилей CSS:

**.nav . active {**

**…**

**}**

Т.к. при определение классов мы обращаемся к свойствам объекта classes (можно при импорте назвать его по-другому), то мы не можем использовать в именах классов дефис. Можем, если только будем использовать другой синтаксис типа classNmae={classes.[“header-rm”]} вместо {classes.header}. **Поэтому, в именах классов ДЕФИС НЕ ИСПОЛЬЗУЕМ, можно camelCase, можно нижнее подчеркивание\_\_**

15. Структура папок, новые компоненты, улучшаем CSS

Сгруппируем в отдельные папки файл компонента и файл стилей.

Дробим наши полученные компоненты.

В папке исходного компонента создаем папку с именем компонента следующе уровня, вместе с ним и стили css.

Так мы можем использовать одно из главных преимуществ компонент, мы можем их потом дублировать тегами!

16. PROPS

Компонент - это функция, возвращающая разметку JSX

она же

Компонент - это тег

в React функция вызывается с параметром props

export function Header(props) {  
 return (  
 <header className={classes.header}>  
 <img  
 src="https://www.phpro.be/uploads/media/sulu-100x100/00/440-react%404x.png?v=2-0"  
 alt="logo"/>  
 </header>  
 )  
}

React всегда вкидывает объект в функцию (компоненту)

Когда мы указываем компоненту без аттрибутов props, в компоненту приходит пустой оъект.

Компоненту ИЗВНЕ мы настраиваем с помощью атрибутов.

Создадим атрибут компоненту <Header/>

<Header name="Dima K"/>

Тогда React автоматически создаст props объект и отрисовывается будет этот компонент уже с объектом. Т.е. компонент как функция будет иметь доступ к объекту как будто аргумент у функции.

{  
 name: "Dima K"  
}

Как использовать объект props в компоненте (функции) мы прописываем в фигурных скобках {}

export function Header(props) {  
 return (  
 <header className={classes.header}>  
 <img  
 src="https://www.phpro.be/uploads/media/sulu-100x100/00/440-react%404x.png?v=2-0"  
 alt="logo"/>  
 </header>  
 <span>{props.name}</span>  
 )  
}

18. Страница Dialogs

<App/> -это функция, которая возвращает JSX и принимает параметры props.

Один из принципов функционального программирования - функция должна возвращать какое-то значение, основываясь только на входящих данных (предсказуемое поведение)

19.Route, browser-router, маршрутизация

При нажатии на **ссылки вида <a href=”/profile”>** браузер все равно загружает страницу **index.html,** но при этом **менятся ulr.** Если мы хотим, чтобы что-то отрисовывалось в зависимости от url, мы должны использовать **ROUTING.**

Встроенного пакета в Create-React-App нет, поэтому устанавливаем пакет роутинга **через терминал** через команду **npm i react-router-dom -save.**

При использовании yarn и Typescript дописываем:

**yarn add react-router-dom @types/react-router-dom**

**ВАЖНО!** Папка node-modules в нашем проекте не попадает в удаленный репозиторий, нужно обязательно прописать установку этого роутера в конфиге. Поэтому к команде установки модуля добавляем **-save.** Это сохранит зависимости в конфиг- файле package.json и потом при клоне автоматически установит пакет.

Чтобы использовать Routing нужно сделать обрамление всего кода компонента в тег(функцию) **<BrowserRouter>** (Также нужно выполнить **import {BrowserRouter, Route} from 'react-router-dom'**

При использовании **<BrowserRouter>** не обязательно отдельно подключать отдельную библиотеку для подключения истории.

**Итого порядок создания ссылок в приложении React.**

* Добавить модуль router(см выше)
* В ссылке тега <NavLink to=”/Music”> пишем имя компонента Music
* В компоненте отрисовки этих страниц, весь код оборачиваем тегом <BrowserRouter>…</BrowserRouter>
* Отображение компонент пишем в теге вида <Route path='/music' component={Music}/>

Получается примерно следующее:

function App() {  
 return (  
 <BrowserRouter>  
 <div className="app-wrapper">  
 <Header/>  
 <Navbar/>  
 {/\*<Profile/>\*/}  
 <div className="app-wrapper-content">  
 <Route path='/dialogs' component={Dialogs}/>  
 <Route path='/profile' component={Profile}/>  
 <Route path='/news' component={News}/>  
 <Route path='/music' component={Music}/>  
 <Route path='/Settings' component={Settings}/>  
 </div>  
 </div>  
 </BrowserRouter>  
 )**;**}

20. NavLink

При нажатии на **ссылки вида <a href=”/profile”>** браузер перезагружает всю страницу целиком, а это противоречит концепции SPA, поэтому вместо ссылок в тегах <a> используем ссылки в тегах(они же компоненты) **<NavLink>.** NavLink нужно импортировать!

Вместо атрибута href=’’ нужно использовать атрибут **to=’’.** Все атрибуты компонент превращаются в PROPS. Итого получаем ссылки такого вида:

import React from 'react'**;**import classes from './Navbar.module.css'**;**import {NavLink} from 'react-router-dom'**;**export function Navbar() {  
 return (  
 <nav className={classes.nav}>  
 <div className={classes.item}>  
 <NavLink to="/profile">Profile</NavLink></div>  
 <div className={classes.item}>  
 <NavLink to="/dialogs">Messages</NavLink></div>  
 <div className={classes.item}>  
 <NavLink to="/News">News</NavLink>  
 </div>  
 <div className={classes.item}>  
 <NavLink to="/Music">Music</NavLink></div>  
 <div className={classes.item}>  
 <NavLink to="/Settings">Settings</NavLink></div>  
 </nav>  
 )  
}

При этом, если посмотреть в devtools, компоненты NavLink будут отображаться как теги <a> с сcылками href.

При нажатии на ссылку, определенную как NavLink, он добавляет ссылке **класс active** (это моно увидеть в DevTools)

Но классы CSS мы должны брать из объекта CLASSESб поэтому просто прописать стили классу active работать не будет. Чтобы заработал класс active, нужно добавить к Navlink такой атрибут:

activeClassName={classes.activeLink}

где .activeLink - это имя класса для состояния active.

Итого наша компонента выглядит так:

import React from 'react'**;**import classes from './Navbar.module.css'**;**import {NavLink} from 'react-router-dom'**;**export function Navbar() {  
 return (  
 <nav className={classes.nav}>  
 <div className={classes.item}>  
 <NavLink to="/profile"  
 activeClassName={classes.activeLink}>Profile</NavLink>  
 </div>  
 <div className={classes.item}>  
 <NavLink to="/dialogs"  
 activeClassName={classes.activeLink}>Messages</NavLink>  
 </div>  
 <div className={classes.item}>  
 <NavLink to="/News"  
 activeClassName={classes.activeLink}>News</NavLink>  
 </div>  
 <div className={classes.item}>  
 <NavLink to="/Music"  
 activeClassName={classes.activeLink}>Music</NavLink>  
 </div>  
 <div className={classes.item}>  
 <NavLink to="/Settings"  
 activeClassName={classes.activeLink}>Settings</NavLink>  
 </div>  
 </nav>  
 )  
}

а модуль CSS следующим образом:

.nav {  
 grid-area: nav**;** background-color: darkkhaki**;**}  
  
.item a {  
 color: whitesmoke**;**}  
  
.item a.activeLink {  
 color: gold**;**}

22.Route exact

Если в библиотечную компоненту **<Route>** добавить **exact**, то он будет отслеживать путь по точному совпадению.

<Route exact path='/dialogs' component={Dialogs}/>

При смене ulr НЕ обязательно нужно делать новый ROUT, т.е. мы можем и нам нужно при клике подгружать не новую компоненту, а просто содержимое (текст)

24. UI - BLL, выносим данные в сторону

BLL - business logic layer (REDUX)

UI - user interface (REACT)

25. Отображаем компоненты с посощью метода массива MAP

Имея исходный массив данных, мы на основании него можем отрисовывать массив компонентов.

**const postsData = [  
 {id: 1, message: "Hi! How are you?"},  
 {id: 2, message: "It is my first post"},  
 ]**  
  
 const postsElements = postsData.map((item)=> {  
 return (<Post message={item.message}/>)  
 })  
  
 return (  
 <div className={classes.postsBlock}>  
 <h3>My posts</h3>  
 <div>  
 <div>  
 <textarea>Input your thoughts here...</textarea>  
 </div>  
 <div>  
 <button>Add post</button>  
 <button>Remove</button>  
 </div>  
 </div>  
 <div className={classes.posts}>  
 **{postsElements}**  
 </div>  
 </div>  
 )  
}

При изменении исходного массива данных будет автоматически отображаться нужное количество компонент с данными из массива

26. Структура проекта, еще раз про ui-bll

- index.ts не является компонентой проекта.

- из intex.ts вызываем App.

- ВСЕ содержимое App оборачивается обязательно BrowserRouter (или HashRouter )

function App() {  
 return (  
 <BrowserRouter>  
 <div className="app-wrapper">  
 <Header/>  
 <Navbar/>  
 {/\*<Profile/>\*/}  
 <div className="app-wrapper-content">  
 <Route path='/dialogs' component={Dialogs}/>  
 <Route path='/profile' component={Profile}/>  
 <Route path='/news' component={News}/>  
 <Route path='/music' component={Music}/>  
 <Route path='/Settings' component={Settings}/>  
 </div>  
 </div>  
 </BrowserRouter>  
 );  
}

- из App далее идет ветвление всех остальных компонент

- Все компоненты должны находиться в отдельных файлах.

**BLL (Business Logic Layer) redux**

BLL это некая абстракция, которая отвечает за хранение данных. Из нее PROPS приходят в компоненту. На сегодня BLL это REDUX.

Будем переносить все данные в BLL (index.tsx)

**UI (user interface) react**

Данные в реакт приходят из BLL

27. Components vs Render, прокидываем PROPS через Render

Если мы перенесем все данные в index.tsx, то компиляция будет невозможна, т.к. компоненты не смогут передать данные из index.tsx из-за ROUTE. Решить эту проблему можно двумя путями:

1. Используем атрибут **render.** Router ожидает получить или component или render, одно из двух

function App() {  
 return (  
 <BrowserRouter>  
 <div className="app-wrapper">  
 <Header/>  
 <Navbar/>  
 {/\*<Profile/>\*/}  
 <div className="app-wrapper-content">

<Route path='/dialogs' **component**={Dialogs}/>  
//или  
  
 <Route path='/dialogs' **render**={Dialogs}/>  
  
 </div>  
 </div>  
 </BrowserRouter>  
 );  
}

Render принимает не функцию(компоненту), а стрелочную функцию, которая вызовет нужную компоненту

28. Выносим данные в index.tsx

Если мы перенесем все данные в index, нужно создать типизацию, а также прокинуть все данные через все props.

Если мы используем метод MAP, то его нужно писать внутри компоненты

29. Упаковываем данные в STATE

Хранить все данные будем в файле STATE.TS.

Из этого файла будем делать импорт данных в INDEX.JSX, чтобы система была контролируемая

31. onClick, ref, Virtual DOM

В Реакте мы не можем напрямую обращаться к DOM, т.е. у нас есть Virtual DOM. Например, м не можем брать данные из document.getElementById и прочее.

Для того, чтобы вытянуть значения из TextArea будем использовать ref.

import React from 'react'

import classes from './MyPosts.module.css'

import { Post } from './Posts/Post'

export *type* MyPostsPropsType = {

  posts: *Array*<PostElementType>

}

export *type* PostElementType = {

  id: *number*

  message: *string*

}

export *function* MyPosts (*props*: MyPostsPropsType) {

*let* postsElements = *props*.posts.map(*item* *=>* {

    return <*Post* message={*item*.message} />

  })

*let* newPostElement = React.createRef() //создаем переменную

  return (

    <div className={classes.postsBlock}>

      <h3>My posts</h3>

      <div>

        <div> //вписываем ref={}

          <textarea ref={newPostElement}>Input your thoughts here...</textarea>

        </div>

        <div>

          <button>Add post</button>

          <button>Remove</button>

        </div>

      </div>

      <div className={classes.posts}>{postsElements}</div>

    </div>

  )

}

31. Прокидываем callback через PROPS

Когда мы нажимаем на textarea, чтобы создать пост, нам нужно добавить сначала эти данные в исходный массив, который хранится в redux. Там же должна храниться функция, которая добавляет данные к исходному массиву

Перейдем в sate.js и создадим функцию добавления постов:

export *let* addPost = (*postMessage*: *string*) *=>* {

*let* newPost = {

    id: 5,

    message: *postMessage*

    // likesCount: 0

  }

  state.postsData.push(newPost)

}

Проверим, работает ли функция, вызвав ее в index.ts (нужно ее импортнуть)

addPost("WoW Samurai");

Далее нам нужно прокинуть функцию в ПРОПС index.tsx

ReactDOM.render(

  <*React.StrictMode*>

    {/\*<App dialogsData={dialogsData} messagesData={messagesData} postsData={postsData}/>\*/}

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

  </*React.StrictMode*>,

  document.getElementById('root')

);

и так далее до компоненты MyPosts.tsx. В ней по клику и вызываем функцию.

*let* addPost = () *=>* {

    //debugger

*let* text = newPostElement.current?.value //? значит, что в этом поле моежт быть Null

    if (text) {

      // check if text exists (no undefined). Without this snippet type error is caused

*props*.addPost(text)

    }

  }

  return (

    <div className={classes.postsBlock}>

      <h3>My posts</h3>

      <div>

        <div>

          <textarea ref={newPostElement}></textarea>

        </div>

        <div>

          <button onClick={addPost}>Add post</button>

          <button>Remove</button>

        </div>

      </div>

      <div className={classes.posts}>{postsElements}</div>

    </div>

  )

}

33. Добавление поста на стену (зачатки flux концепции)

FLUX - это архитектура, REDUX - это ее реализация (реализаций много)

Ниже на схеме представлен однонаправленный поток данных. Компоненты(React) могут только отправлять actions, т.е. запрос на изменение данных, могут даже собирать эти данные, могут вызывать функции, пришедшие из-вне. Но менять сове состояние сами не могут, только после команды от STATE, когда он изменился.



НЕЛЬЗЯ ИЗМЕНЯТЬ UI ДО ИЗМЕНЕНИЯ STATE!

Помним - **главный у нас BLL (Business Logic Layer**)**. UI мы можем "убить" в любой момент и отрисовать заново!!!** Даже если в state изменилась какая-то супер маленькая деталь, плевать - берём и отрисовываем ВСЁ заново. Ой, скажете вы, это же не эффективно??) Отчасти вы правы. Но мы на старте не думаем про оптимизацию! Давайте думать про концепцию и архитектуру, а потом будем думать про то, как всё "ускорить". К тому же мы используем React, а он далеко не дурак, чтобы всё перерисовывать, если это не нужно и даже если мы ему типа приказали) FLUX концепция, она немного другая, безусловно. Там есть в цепочке этой немного другие звенья. Но суть ТАКАЯ ЖЕ! Мы будем эту картину формировать поэтапно, постепенно, чтобы у вас было понимание того, что происходит!!! Летим!!!!!!!

34. FLUX круговорот, STATE MANAGEMENT

Где хранить стейт?

- Если приложение совсем маленькое, можно использовать **локальный стейт каждой компоненты**

- Можно консолидировать данные в **определенных родительских компонентах**

- Можно весь стейт **хранить в одном файле**

**Все зависит от задач и архитектуры приложения.**

Существуют определенные парадигмы по хранению данных в React. Среди них:

**Redux** - парадигма для функционального программирования

**MobX** - парадигма для ООП



**Redux** это библиотека, которая реализовывает **Flux** архитектуру.

НЕЛЬЗЯ ИЗМЕНЯТЬ UI ДО ИЗМЕНЕНИЯ STATE! КАЖДЫЙ СИВОЛ В INPUT!

35. Callback, subscribe, observer

Избегаем циклические зависимости, поэтому rerenderEntireTree() переносим в callback функцию.

35. Про ООП поверхностно

**Инкапсуляция** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *encapsulation*, от [лат.](https://ru.wikipedia.org/wiki/%D0%9B%D0%B0%D1%82%D0%B8%D0%BD%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *in capsula*) — в [информатике](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0) размещение в одном компоненте данных и методов, которые с ними работают. Также может означать **скрытие** внутренней реализации от других компонентов. Например, доступ к скрытой переменной может предоставляться не напрямую, а с помощью методов для чтения ([геттер](https://ru.wikipedia.org/wiki/%D0%93%D0%B5%D1%82%D1%82%D0%B5%D1%80_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5))) и изменения ([сеттер](https://ru.wikipedia.org/wiki/Setter)) её значения.

*let* page = {

  title: 'SamuraiJS', // является интерфейсом

  \_content: '', //не является интерейсом (нельзя к нему обращаться и изменять самому)

  setContent (*content*) { //setter

    this.\_content = this.content

  },

  getContent () { //getter

    return this.\_content

  },

  render: *function*() {

    document.write(this.\_content)

  }

}

**к свойству с нижним подчеркиванием нельзя обращаться \_content: '',**

Это приватное свойство (икапсулированное) можно изменять ИЗВНЕ только методами СЕТТЕРАМИ, для получения свойства ИЗВНЕ нужно пользоваться ГЕТТЕРАМИ

В ООП данные и логика сидят в одном объекте. С ним и нужно взаимодействовать.

37. Bind

**Если объект передает свой метод кому-то другому (в колл бек например), если мы хотим, чтобы сохранился владелец метода, его надо байндить. Иначе он будет вызываться от имени другого объекта.**

 <*Profile*

                postsData={state.profilePage.posts}

                addPost={*props*.store.addPost.bind(*props*.store)}

                newPostText={state.profilePage.newPostText}

                updatePostText={*props*.store.updatePostText.bind(*props*.store)}

              />

38. dispatch()

**Типизация:**

**dispatch: (action: any) => void**

**action, который приходит в dispatch, является объектом, причем каждый раз разным. Каждый тип объекта action нужно прописывать отдельно. Из этого составить тип action.**

**type AddPostActionType = {**

**type: ‘ADD\_POST’**

**postText: string**

**}**

**type ChangeNewTextActionType = {**

**type: ‘CHANGE-NEW-TEXT’**

**newText: string**

**}**

**получаем тип action:**

**type ActionsTypes= AddPostActionType | ChangeNewTextActionType**

**dispatch: (action: ActionType) => void**

**Зачем нужен dispatch()?**

Чтобы не плодить огромное количество методов объекта Стор, мы их закинем в один - **dispatch**



38. Action creator,

action type

**dispatch(action)**

**action - это объект! В нем обязательно есть свойство type.**

**Принцип single responsibility помогает избежать ошибок**

На данном этапе action создаем вручную и отправляем объекты из UI.

Но нужно делегировать создание action в **actionCreator() в BLL. Вызов будет идти через callback**

**Было:**



**Стало**



ACTION TYPE надо вынести в отдельные переменные

export *const* ADD\_POST: 'ADD\_POST' = 'ADD\_POST'

export *const* UPDATE\_POST\_TEXT: 'UPDATE\_POST\_TEXT' = 'UPDATE\_POST\_TEXT'

41. Reducer()

Reducer() - чистая функция, которая принимает state и action, возвращает измененный/не измененный state.

42. REDUX

Store - это ООП объект, который содержит State и его методы.

**Redux** — библиотека для [JavaScript](https://ru.wikipedia.org/wiki/JavaScript" \o "JavaScript)[[2]](https://ru.wikipedia.org/wiki/Redux#cite_note-2) с открытым исходным кодом, предназначенная для управления состоянием приложения.

https://redux.js.org/

Инсталляция Redux в проект:

yarn add redux

Redux сам создает STORE (до этого мы создали сами, чтобы понимать как он работает):

let store = createStore(); //создаем store

Объединяем reducers:

let reducers = combineReducers() // Объединяем reducers. Внутри передаем объект со свойствами редьюсеров

*let* reducers = combineReducers({

  profileReducer,

  dialogsReducer,

  sidebarReducer

})

Закомбайненые редьюсеры отдаем в стор:

*let* reducers = combineReducers({

  profileReducer,

  dialogsReducer,

  sidebarReducer

})

export *let* store = createStore(reducers)

Редаксу нужно для первой отрисовки (инициализции) передать в каждый редьюсер изначальное состоние, т.к. на данном этапе оно udefined. Делается это путем использования «параметров по умолчанию» Например, есть функция, в которой зададим параметр по умолчанию для b:

*let* sum = (*a*, *b* = 1) *=>* {

  return *a* + *b*

}

для этого создаем сначала initial state в каждом reducer

43. Презентационные VS контейнерные компоненты

Сколько пропсов прокидывать в компоненты? Как можно меньше! Чтобы компонента как можно меньше знала про STORE. В случае функций прокидываем колл-беки от родителя к дочерним компонентам



Контейнерная компонента служит для того, чтобы обслуживать презентационные.

В контейнерную компоненту можно передать вест STORE

Вам будете гораздо проще повторно использовать компоненты, если вы **разделите их на две категории**. Я называю их Контейнер и Презентационный компоненты\*. Но также слышал Толстый и Худой, Умный и Глупый, Имеющий состояние и Чистый, соответсвенно. Они не совсем идентичны, но в основе лежит одна идея.



Ниже мы обернули компонент MyPosts в контейнер. Теперь она не имеет напрямую связь со стором, а только через колл беки, которые ей передает контейнер:

export *function* MyPostsContainer (*props*: MyPostsPropsType) {

  //---BLL

*let* onAddPost = (*newText*: *string*) *=>* {

*props*.dispatch(addPostAC(*newText*))

  }

*let* onPostChange = (*newText*: *string*) *=>* {

*let* action = updatePostTextAC(*newText*)

*props*.dispatch(action)

  }

  //---UI

  return (

    <*MyPosts*

      updateNewPostText={onPostChange}

      addPost={onAddPost}

      newPostText={*props*.newPostText}

      postsData={*props*.postsData}

    />

  )

}

export *function* MyPosts (*props*: MyPostsPropsType) {

  //---BLL

*let* postsElements = *props*.postsData.map((*item*, *i*) *=>* {

    return <*Post* key={*i*} message={*item*.message} />

  })

*let* newPostElement = React.createRef<*HTMLTextAreaElement*>()

*let* addPost = () *=>* {

*let* newText = *props*.newPostText

    if (newText) {

*props*.addPost(newText)

    }

  }

  //---UI

  return (

    <div className={classes.postsBlock}>

      <h3>My posts</h3>

      <div>

        <div>

          <textarea

            // TODO вынести функцию из JSX (e: ChangeEvent<HTMLTextAreaElement> )

            onChange={*e* *=>* {

*let* newText = *e*.currentTarget.value

*props*.updateNewPostText(newText)

            }}

            ref={newPostElement}

            value={*props*.newPostText}

          />

        </div>

        <div>

          <button onClick={addPost}>Add post</button>

          <button>Remove</button>

        </div>

      </div>

      <div className={classes.posts}>{postsElements}</div>

    </div>

  )

}

Основная идея - разделить работу с данными и отображение контента.

44. Context API

<https://ru.reactjs.org/docs/context.html>

Context API позволяет передать информацию не по цепочке из компонента в компонент, а сразу от родителя к любому потомку

Контекст позволяет передавать данные через дерево компонентов без необходимости передавать пропсы на промежуточных уровнях.

**Когда использовать контекст?**

Контекст разработан для передачи данных, которые можно назвать **«глобальными» для всего дерева React-компонентов** (например, текущий аутентифицированный пользователь, UI-тема или выбранный язык).

Сначала создаем контекст

**const MyContext = React.createContext(**defaultValue**)**

Обернуть компоненты тегом:

**<MyContext.Provider** value={/\* some value\*/}>, в нашем случае будем передавать value это store

Затем можно обратиться к контексту с помощью такого синтаксиса:

**<MyContext.Consumer>**

{value => /\* render something based on the context value\*/}

**</MyContext.Consumer>**

1. **Создаем файл storeContext.ts**
2. **В файле index.tsx обрамляем App в </StoreContext.Provider** value={store}>. **Всем компонентам внутри App будет доступен стор**

export *let* renderTree = (*store*: *any*) *=>* {

  ReactDOM.render(

    <*React.StrictMode*>

      <*StoreContext.Provider* value={*store*}>

        <*App* store={*store*} />

      </*StoreContext.Provider*>

    </*React.StrictMode*>,

    document.getElementById('root')

  )

}

1. **К контексту можно подключать только КОНТЕЙНЕРНЫЕ КОМПОНЕНТЫ, нельзя подключать к презентационным!**
2. **Обернем нашу компоненту DiaogsContainer.tsx в <StoreContext.Consumer>, добавим колл-бек функцию Перенесем функции компоненты в тот же колл-бек:**

**Похожий функционал можно сделать с помощью библиотеки React Redux. () не путать с простым Redu!!!)**

**https://react-redux.js.org/**

45. Connect, mapStateToProps, mapDispatchToProps

Будем использовать дополнительную библиотеку React-Redux. Она будет как прослойка между компонентами и сторомю Не путать с библиотекой просто Redux!!!

**yarn add react-redux --save**

Пакет react-redux предоставляет привязки React для контейнера состояния Redux, чрезвычайно упрощая подключение React-приложения к хранилищу Redux. Это позволяет разделять компоненты React-приложения, основываясь на их связи с хранилищем. А именно, речь идёт о следующих видах компонентов:

1. Презентационные компоненты. Они отвечают лишь за внешний вид приложения и не осведомлены о состоянии Redux. Они получают данные через свойства и могут вызывать коллбэки, которые также передаются им через свойства.
2. Компоненты-контейнеры. Они ответственны за работу внутренних механизмов приложения и взаимодействуют с состоянием Redux. Их часто создают с использованием react-redux, они могут осуществлять диспетчеризацию действий Redux. Кроме того, они подписываются на изменения состояния.

Пакет react-redux обладает очень простым интерфейсом. В частности, самое интересное в этом интерфейсе сводится к следующему:

1. <Provider store> — позволяет создавать обёртку для React-приложения и делать состояние Redux доступным для всех компонентов-контейнеров в его иерархии.
2. connect([mapStateToProps], [mapDispatchToProps], [mergeProps], [options]) — позволяет создавать компоненты высшего порядка. Это нужно для создания компонентов-контейнеров на основе базовых компонентов React.

*let* f1 = () *=>* {

  return {

    a: 1,

    b: 2

  }

}

*let* f2 = () *=>* {

  return {

    c: 3

  }

}

*const* superDialogsContainer = connect(f1, f2)(Dialogs)

**Этот код означает: connect создаст контейнерную компоненту, отрисует презентационную компоненту Dialogs, и засунет в нее через аттрибуты a =1, b = 2, c= 3. Значит, в объекте props компоненты Dialogs будет сидеть props.a, props.b, props.c**

**Connect передает не STORE, a STATE! Забываем про STORE!**

На основании этого передадим данные в компоненту Dialogs. В примере ниже первая функция будет передавать State, назовем ее mapStateToProps, вторая функция будет передавать колл-беки в презентационную компоненту. Назовем ее mapDispatchToProps

*let* mapStateToProps = (*state*: RootStateType) *=>* {

  return {

    // отправляем данные из СТЕЙТА

    dialogsPage: *state*.dialogsPage

  }

}

*let* mapDispatchToProps = (dispatch: (*action*: ActionType) *=>* *void*) *=>* {

  // колл-беки, которые будем отправлять в презентационую компоненту

  return {

    UpdateNewMessageBody: (*newText*: *string*) *=>* {dispatch(UpdateNewMessageBodyAC(*newText*))},

    SendMessage: (*newText*: *string*) *=>* {dispatch(SendMessageAC(*newText*))}

  }

}

*const* superDialogsContainer = connect(mapStateToProps,mapDispatchToProps)(Dialogs)

Итого, опишем, что мы здесь сделали:

*let* mapStateToProps = (*state*: RootStateType) *=>* {

  return {

    // отправляем данные из СТЕЙТА

    dialogsPage: *state*.dialogsPage

  }

}

*let* mapDispatchToProps = (dispatch: (*action*: ActionType) *=>* *void*) *=>* {

  // колл-беки, которые будем отправлять в презентационую компоненту

  return {

    UpdateNewMessageBody: (*newText*: *string*) *=>* {

      dispatch(UpdateNewMessageBodyAC(*newText*))

    },

    SendMessage: (*newText*: *string*) *=>* {

      dispatch(SendMessageAC(*newText*))

    }

  }

}

export *const* DialogsContainer = connect(mapStateToProps,mapDispatchToProps)(Dialogs)

Создали две функции mapStateToProps  и mapDispatchToProps, которыми настроили connect (законнектили компонент Dialogs по правилам mapStateToProps  и mapDispatchToProps )

46. Deep copy, shallow copy

Эту тему уже проходили

47. Делаем копию состояния в ProfileReducer

**Идемпоте́нтность** ([лат.](https://ru.wikipedia.org/wiki/%D0%9B%D0%B0%D1%82%D0%B8%D0%BD%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *idem* — тот же самый + *potens* — способный) — свойство объекта или операции при повторном применении операции к объекту [давать тот же результат](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%B2%D0%B0%D1%80%D0%B8%D0%B0%D0%BD%D1%82_(%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)), что и при первом.

Идемпотентным называют такой метод API, повторный вызов которого не меняет состояние. Здесь есть тонкий момент: результат идемпотентного вызова может меняться. Например, при повторном вызове идемпотентного API создания заказа — заказ не будет создаваться еще раз, но API может ответить как 200, так и 400. При обоих кодах ответа API будет идемпотентно с точки зрения состояния сервера (заказ один, с ним ничего не происходит), а с точки зрения клиента поведение существенно разное.

В connect есть свой локальный subscribe (отвечает за изменения). Как он работает? Каждый раз, когда происходят локальные изменения, запускается функция mapStateToProps() и формируется новый объект. Новый объект сравнивается со старым (внутренности). Объект не поменялся, т.к. мы изменили существующий стейт. Реакту не с чем сравнить его, чтобы понять, нужно ли перерисовывать UI. Поэтому нужно обязательно **ДЕЛАТЬ КОПИЮ ОБЪЕКТА стейта, перед тем, как с ним работать! Следовательно, в редьюсерах нужно создать новый объект с глубокой копией и с ней возаимодействовать:**

export *const* profileReducer = (

*state*: InitialStateType = initialState,

*action*: ActionType

): InitialStateType *=>* {

  switch (*action*.type) {

    case ADD\_POST: {

*let* newPost = {

        id: 5,

        message: *action*.postMessage,

        likesCount: 0

      }

*let* stateCopy = { ...*state* }

      stateCopy.posts = [...stateCopy.posts]

      stateCopy.posts.unshift(newPost)

      stateCopy.newPostText = ''

      return stateCopy

    }

    case UPDATE\_POST\_TEXT: {

*let* stateCopy = { ...*state* }

      stateCopy.newPostText = *action*.newText

      return stateCopy

    }

    default:

      return *state*

  }

}

В редьюсере кроме поверхностной копии, глубоко копируем только то, что собираемся менять. Иначе это будут лишние данные и трата лишних ресурсов!

47. Делаем копию состояния в DialogsReducer

Мы должны делать копию только того свойства стейта, что меняется. От action зависит, какую ветку объекта нужно делать глубокую копию. Перепишем DialogsReducer + сделаем рефакторинг

export *const* dialogsReducer = (

*state*: InitialStateType = initialState,

*action*: ActionType

): InitialStateType *=>* {

  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*

  }

}

Зарефакторим также прошлый редьюсер profileReducer аналогичным способом

*let* mapStateToProps = (*state*: AppStateType): MapStatePropsType *=>* {

  return {

    newPostText: *state*.profilePage.newPostText,

    postsData: *state*.profilePage.posts

  }

}

*let* mapDispatchToProps = (dispatch: (*action*: ActionType) *=>* *void*) *=>* {

  return {

    updateNewPostText: (*newText*: *string*) *=>* {

      dispatch(updatePostTextAC(*newText*))

    },

    addPost: (*newText*: *string*) *=>* {

      dispatch(addPostAC(*newText*))

    }

  }

}

export *const* MyPostsContainer = connect(

  mapStateToProps,

  mapDispatchToProps

)(MyPosts)

49. Страница пользователей Users

Создадим компонент from scratch. С чего начать разработку?

***Совет: почитать книгу Domain Driven Design (Проблемно-ориентированное программирование Эрик Джю Эванс)***

Опытные разработчики начинают с BLL, потом UI.

Но нам нужно знать примерно визуальное представление, чтобы начать разработку BLL

1. **Начинаем с reducera (это чистая функция, которая принимает старый стейт, аргументы и выводит новый стетйт). Создаем InitialState и прочее. Все в файле usersReducer: acionTypes, ActionCreators, InitialState, UsersReducer**
2. **Создаем usersConatiner. В нем используем connect()**
3. **Создаем компоненту Users**

50. Страница пользователей Users

**API - Application Programing Interface (интерфейс взаимодействия с программой)**

На интересует интерфейс взаимодействия с программным сервером. Это могут быть Node JS, Apache, NGiNX и т.д.

У каждой программы-сервера есть end-points.

end-points это URL-адреса, на которые мы можем делать запросы

**Что нужно определить для end-point (это и есть SERVER API):**

**1) URL** [**https://....**](https://....)

**2) Тип запроса http-request type: GET(хотим что-то получить от сервера) / POST (хотим отправить что-то на сервер)**

**3) request payload. Что именно мы хотим отправить на сервер?**

**4) Response data. Что мы хотим получить с сервера?**

**5) http коды: 404 not found, 5xx server errors, 3xx redirect, 2xx OK**



Что такое REST API?

По сути это набор правил, которым должна соответствовать программа. Раньше хватало только два типа запроса на сервер: GET и POST. Сейчас их несколько, это CRUD: GET POST PUT DELETE (CREATE READ UPDATE DELETE). Запросы отправляются только на один утв-point для каждой сущности. Итого это выглядит так:



В нашем проекте мы будем использовать библиотеку axios.



**.get** - отправляем запрос на сервер (возвращает промис)

**.then** - когда придет ответ от сервера выполняем команду. При это выполнение скрипта в браузере не замораживается, т.е. выполняется АСИНХРОННО. Другими словами, код сработает следующим образом:

1. Отправится запрос на сервер
2. Выведется console.log(‘after request’)
3. Выведется результат then console.log(data)

AJAX

**AJAX: Asynchronous Javascript and XML** **– технология обращения к серверу без перезагрузки страницы. Как делаются AJAX запросы?**

**Устаревшие методы: XMLHttpRequest, jQuery**

**Сейчас можно использовать fetch API, axios**

**Создадим файл, подключим jQuery, создадим AJAX запрос.**

**Для простейшего AJAX запроса нужен будет только URL**

$.ajax('https://repetitora.net/api/JS/Images')

**В итоге в devtools dj вкладе мы будем работать в этом окне, пользуемся вкладками!**

****

Чтобы обработать запрос от сервера нужен колл-бек. Это функция, которую мы отдам кому-то, чтобы кто-то другой ее запустил когда настанет время это сделать.

Например, **setTimeout(function() {}, 1000).** Ей передаем колл-бек, который она запустит через секунду.

Чтобы передать функцию-коллбек AJAX, нужно создать в запросе объект с функцией:

$.ajax('https://repetitora.net/api/JS/Images', {

  success: *function* () {

    console.log("I've got response")

  }

})

**Колл-беки очень часто используются, когда речь идет про АСИНХРОННЫЙ КОД**

**Когда библиотека Jquery запускает метод ajax и получает ответ, в функцию-колл-бек можно передать объект data от сервера:**

$.ajax('https://repetitora.net/api/JS/Images', {

  success: *function* (*data*) {

    console.log(*data*)

  }

})

**Если работаем из браузера, то во вкладе Sources можно использовать точки останова, чтобы дебажить и смотреть данные!**

**Итак, напишем код, который будет подтягивать картинки с сервера по запросу:**

$.ajax('https://repetitora.net/api/JS/Images', {

  success: *function* (*data*) {

*data*.forEach(*element* *=>* {

*const* img = document.createElement('img')

      img.src = *element*.thumbnail

      document.querySelector('body').appendChild(img)

    })

  }

})

Есть бесплатные сервер API по запросу в нете: publick get api for testing

**ТИПЫ ЗАПРОСОВ (REQUEST METHOD)**

Все AJAX запросы являются HTTP запросами. Вообще можно любым типом запроса выполнить любую задачу. Но есть договоренность использовать характерные запросы.

**GET** - используем, когда хотим что-то взять с сервера. Когда мы пишем адрес в адресной строке браузеры мы по сути делаем GET запрос на конкретную страницу.

Технически при выполнении GET запроса мы должны также отправлять инфо на сервер (для более точного понимания сервером, что мы хотим от него получить, например). Делается это с помощью **GET-параметра ключ-значение. Их мы будем запихивать в строку запроса.** Например:

https://repetitora.net/api/JS/Images**?page=2**

**?page=2 - это GET-параметр. Если их несколько, используем &:**

<https://repetitora.net/api/JS/Images?page=2&otherParam=3&name=james&age=32>

**В качестве параметра, технически можно передать даже файл (перед этим его нужно закодировать)**

**Какие параметры мы можем передавать на сервер?** Это решает серверный разработчик, какие параметры прописаны в серверной программе.

Параметры могут быть **обязательными и опциональными**



Допустим, есть некий URL:

https://repetitora.net/api/JS/Images

Возьмем из таблицы get-параметры, которые мы можем передать на сервер и добавим их к строке:

https://repetitora.net/api/JS/Images?page=1&count=20

Сервер выдаст 20 картинок

**UI - DAL**

Желательно разделить код программы на **UI** - это User Interface и **DAL** - Data Access Layer

В нашем случае ГШ это будут команды, касающиеся только интерфейса

*const* resultBlock = document.querySelector('#result')

*const* pageNumber = document.querySelector('#page-number')

*const* clickMeButton = document.querySelector('#click-me')

clickMeButton.addEventListener('click', () *=>* {

  getImages()

})

ajax запрос это будет DAL, этот код работает только с сервером и не знает ни о разметке ни о логике. Принцип single responsibility. Поэтому код ниже неправильный, из него нужно убрать все, что касается UI.

*function* getImages () {

  $.ajax(

    `https://repetitora.net/api/JS/Images?page=${pageNumber.value}&count=2`,

    {

      success: *function* (*data*) {

*data*.forEach(*element* *=>* {

*const* img = document.createElement('img')

          img.src = *element*.thumbnail

          document.querySelector('#img\_container').appendChild(img)

        })

      }

    }

  )

}

После рефакторинга получим следующее (мы разнесли по разным файлам UI и DAL):

*///////////UI*

*const* resultBlock = document.querySelector('#result')

*const* pageNumber = document.querySelector('#page-number')

*const* clickMeButton = document.querySelector('#click-me')

clickMeButton.addEventListener('click', () *=>* {

  getImages(pageNumber.value, onDataReceived)

})

*function* onDataReceived (*data*) {

*data*.forEach(*element* *=>* {

*const* img = document.createElement('img')

    img.src = *element*.thumbnail

    document.querySelector('#img\_container').appendChild(img)

  })

}

*///// DAL*

*function* getImages (*pageNumber*, *successCallback*) {

  $.ajax(`https://repetitora.net/api/JS/Images?page=${*pageNumber*}&count=2`, {

//callback сработает, когда придет ответ от сервера

    success: *successCallback*

  })

}

Иногда в программах может появиться очень много колл-беков, поэтому на помощь приходят ПРОМИСЫ.

PROMISE, AXIOS

Передача напрямую колл-беков нарушает принцип разделения UI и DAL, поэтому сделаем свой промис, убираем колл-беки

метод $.ajax() на самом деле возвращает ПРОМИС.

**ПРОМИС** - это обещание, что когда функция асинхронная закончится, к результату операции можно достучаться. **Это жирый чувак, который следит за асинхронной операцией.**

//

// UI

//

*const* resultBlock = document.querySelector('#result')

*const* pageNumberEl = document.querySelector('#page-number')

*const* clickMeButton = document.querySelector('#click-me')

clickMeButton.addEventListener('click', () *=>* {

*const* promise = getImages(pageNumberEl.value)

  promise.then(onDataReceived)

})

*function* onDataReceived (*data*) {

*data*.forEach(*element* *=>* {

*const* img = document.createElement('img')

    img.src = *element*.thumbnail

    document.querySelector('#img\_container').appendChild(img)

  })

}

//

// DAL

//

*function* getImages (*pageNumber*) {

*const* promise = $.ajax(

    `https://repetitora.net/api/JS/Images?page=${*pageNumber*}&count=2`

  )

  return promise

}

У промиса есть метод **THEN**. Он дает возможность выполнить функцию, когда промис зарезолвится. **Каждый THEN возвращает ПРОМИС, поэтому можно делать цепочку THEN! promise.then(…).then(…).then(…)**

**Подключим AXIOS g html. Будем использовать вместо Jquery.**

За счет того, что мы четко разделили два layerm нам достаточно поменять только функцию getImages:

*function* getImages (*pageNumber*) {

*const* promise = axios.get(

    `https://repetitora.net/api/JS/Images?page=${*pageNumber*}&count=2`

  )

  return promise

}

ВНИМАНИЕ! У каждой библиотеки свой интерфейс респонза!!! Но в любом случа возвращается объект!

TODOLIST-API, GET, POST

Что добавить инфо на сервер будем использовать метод **POST** для добавления тасок

*function* createTask (*title*) {

*const* promise = axios.post(

    `https://repetitora.net/api/JS/Tasks?widgetId=125&title=${*title*}`

  )

  return promise.then(*response* *=>* {

    return *response*.data

  })

}

вызов функции сделаем прямо в файле

createTask('LearnJSd').then(*data* *=>* {

  console.log(*data*)

})

REQUEST PAYLOAD, POST, отправка данных на сервер

В примере ранее мы передавали данные прямо в строке методом POST. Но у браузера есть ограничения по длине, поэтому лучше использовать **payloader**

У POST запроса есть еще тело запроса **(body).** Либа axios вторым параметром принимает JSON объект с данными. В DevTools их можно увидеть в **Request Payload**

*function* createTask (*title*) {

*const* promise = axios.post(`https://repetitora.net/api/JS/Tasks`, {

    widgetId: 127,

    title: *title*

  })

  return promise.then(*response* *=>* {

    return *response*.data

  })

}

ИДЕМПОТЕНТНОСТЬ, rest api, put, delete

В примере ранее мы передавали данные прямо в строке.

Идемпотентность GET запроса заключается в том, что сколько бы раз мы не отправили GET запрос, мы всегда получим один и тот же ответ. Другими словами, GET запрос ничего не меняет на сервере.

POST запрос НЕ идемпотентный.

**Принцип REST API предусматривает:**

- одинаковые URL для всех типов запросов

- запросы GET POST PUT DELETE

**PUT** используется для обновления (изменение, например поменять title, age) данных.

У метода DELETE как и у GET запроса нет body, n.t. мы не можем передавать объект payloader

*function* deleteTask (*id*) {

*const* promise = axios.delete(

    `https://repetitora.net/api/JS/Tasks?widgetId=128&taskId=${*id*}`

  )

  return promise.then(*response* *=>* {

    return *response*.data

  })

}

deleteTask('23d2f0ce-5c47-4011-af91-6952f1e9f4cd')

Также запишем метод PUT (в нем есть возможность отправить PayLoad)

*function* updateTask (*title*, *taskId*) {

*const* promise = axios.put(`https://repetitora.net/api/JS/Tasks`, {

    widgetId: 128,

    title: *title*,

    taskId: *taskId*

  })

  return promise.then(*response* *=>* {

    return *response*.data

  })

}

updateTask('New title', '4348ea01-2aed-4525-9b4b-21b5519f04b2')

CORS, options, content-type, accept-type

Перед тем как отправить «опасные» запросы на сервер, браузер отправляет метод **OPTIONS,** чтобы спросить у сервера, ты знаешь, что к тебе хочет обратиться кто-то, кому разрешаешь к тебе обращаться? В ответ приходит данные, которые можно увидеть в HEADERS закладке, в том числе объект {status: success}

Cross-Origin Resource Sharing ([CORS](https://developer.mozilla.org/ru/docs/Glossary/CORS)) — механизм, использующий дополнительные [HTTP](https://developer.mozilla.org/ru/docs/Glossary/HTTP)-заголовки, чтобы дать возможность [агенту пользователя](https://developer.mozilla.org/ru/docs/Glossary/User_agent) получать разрешения на доступ к выбранным ресурсам с сервера на источнике (домене), отличном от того, что сайт использует в данный момент. Говорят, что агент пользователя делает запрос с другого источника **(cross-origin HTTP request),** если источник текущего документа отличается от запрашиваемого ресурса доменом, протоколом или портом.

51. USER API, запрос на сервер

Любая серверная API имеет базовый URL. В нашем случае:

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

К этому URL мы будем цеплять отдельные приписки. Для получения списка (**endpoint user`a**) пользователей нам нужно добавить **/users**

**Всю информацию о запросах нужно смотеть в документации к API. Например, для /users**

****

Пропишем наш запрос на получение данных о пользователях:

export *const* Users = (*props*: UsersPropsType) *=>* {

  // BLL

  if (*props*.users.length === 0) {

    axios

      .get('https://social-network.samuraijs.com/api/1.0/users')

      .then(*response* *=>* {

*props*.setUsers(*response*.data.items)

      })

  }

из полученных данных отрисуем Юзеров, у кого нет картинки - сделаем заглушку:

import usePhoto from '../../assets/images/user\_ico.jpg'

  <img

                  src={*u*.photos.small !== null ? *u*.photos.small : usePhoto}

                  className={styles.userPhoto}

                />