# Lesson 14 (css)

1. active существует по умолчанию, его не нужно создавать. Классы, импортированные из Dialogs.module.css, применяются в {} и с меткой, такие классы локальны и имеют уникальные индетефикаторы. Использовать два класса можно двумя способами:

let c1 = 'item'  
let c2 = 'active'  
//' item active'  
let classes = {c1+' '+c2};  
let classesnew = {`${c1} ${c2}`};

<div className={s.dialog + ' ' + s.active}>Dimych</div>

# Lesson 25 (map)

let dialogsData = [  
 {id: 1, name: 'Dimych'},  
 {id: 2, name: 'Andrey'},  
 {id: 3, name: 'Anya'},  
 {id: 4, name: 'Sveta'},  
 {id: 5, name: 'Viktor'},  
 {id: 6, name: 'Valera'},  
 ]

<DialogItem name={dialogsData[0].name} id={dialogsData[0].id}/>,  
<DialogItem name={dialogsData[1].name} id={dialogsData[1].id}/>,  
<DialogItem name={dialogsData[2].name} id={dialogsData[2].id}/>,

1. Можно положить в массив, и вызвать массив:

let dialogsElements = [  
 <DialogItem name={dialogsData[0].name} id={dialogsData[0].id}/>,  
 <DialogItem name={dialogsData[1].name} id={dialogsData[1].id}/>,  
 <DialogItem name={dialogsData[2].name} id={dialogsData[2].id}/>,]

{dialogsElements}

Метод массива map – проходит по каждому элементу массива, и создает новый массив, согласно условию ( в данном случае – если имя мужское – возвращает 1, если нет – 0 )

let oldArray = ['dima','olga','andrey']  
  
let newArray = oldArray.map (name => isMaleName(name) ? 1:0)  
  
получаем : [1,0,1]

1. Вместо dialogsElements иcпользуем map, в итоге создается столько dialogsData, сколько у нас сейчас в диалог дата (при добавлении нового обьекта , он сразу отрисуется)

let dialogsElements = dialogsData  
 .map (dialog => <DialogItem name={dialog.name} id={dialog.id}/> )

# Lesson 26

BLL – bussines logic layer - данные, библеотека(redux) хранятся в BLL

UI – user interface – то,где мы работаем (react)

UI отвечается за отрисовку приходящих данных ( взаимодействие с пользователем), поэтому отправляем массивы данных(такие, как dialogs) в index.tsx

let dialogs = [  
 {id: 1, name: 'Dimych'},  
 {id: 2, name: 'Andrey'},  
 {id: 3, name: 'Anya'},  
 {id: 4, name: 'Sveta'},  
 {id: 5, name: 'Viktor'},  
 {id: 6, name: 'Valera'},  
 ]

# Lesson 27-28

Было

<Route path= '/profile' component={Profile}/>

Стало, что позволяет прокинуть пропсы

<Route path='/profile' render={() => <Profile/>}/>

Вынос данных вверх,было :

function Profile() {  
 let posts = [  
 {id: 1, messages: 'Hi,how are you?', likesCount: 11, img: 'https://upload.wikimedia.org/wikipedia/commons/thumb/0/07/Spb\_metro\_logo.svg/1280px-Spb\_metro\_logo.svg.png'},  
 {id: 2, messages: 'its my first posts', likesCount: 41, img: 'https://upload.wikimedia.org/wikipedia/commons/thumb/e/e5/NASA\_logo.svg/200px-NASA\_logo.svg.png'},  
 ]  
  
 return (  
 <div>  
 <ProfileInfo/>  
 <MyPosts posts={posts} />  
 </div>

Стало:

function Profile(props:PostsType) {  
 return (  
 <div>  
 <ProfileInfo/>  
 <MyPosts posts={props.posts} />  
 </div>

В компоненту приходят данные, и она передает их дальше (используем props.)

Типизируем таким образом: три массива, у каждого свой массив с типизацией.

type AppType = {  
 posts:Array<PostType>  
 messages:Array<MessageType>  
 dialogs:Array<DialogType>  
}

В итоге вынесли все данные в index.tsx

***Объекты*** – это ассоциативные массивы с рядом дополнительных возможностей.

Они хранят свойства (пары ключ-значение), где:

* Ключи свойств должны быть строками или символами (обычно строками).
* Значения могут быть любого типа.

# Lesson 29

1. Создали файл state.tsx, перенесли данные в стайт.
2. Типизировали

# Lesson 32

1. Создали в стейт функцию, которая добавляет сообщения запушивая его в массив, прокинули ее через пропс

export let addPost = (postMessage:string) => {  
 let newPost:PostsType = {  
 id:new ***Date***().getTime(),//уникальный ид  
 messages:postMessage,  
 likesCount: 0,  
 img:’’  
 ***state***.profilePage.posts.push(newPost)

# Lesson 33

1. Создали функцию ререндер, обернули ей все содержимое индекс и вынесли в отдельный файл, в индексе вызываем функцию со стейтом в параметре

# Lesson 34

Flux круговорот – при вводе значение в инпут, это значение сначала попадает в стейт и только потом отприсовывается.

1. При изменении инпута, в updateNewPostText записывается текущее состояние инпута, далее при нажатии кнопки функция аддПост отрисовывает сообщение, записанное в newPostText

let addPost = () => { props.addPost()}  
const onChangeHandler = (event: ChangeEvent<HTMLTextAreaElement>) => {  
 props.updateNewPostText(event.currentTarget.value)  
}

<textarea  
 onChange={onChangeHandler}  
 value={props.newPostText}  
 ref={newPostElement}> </textarea>  
</div>  
<div>  
 <button onClick={addPost}> Add post</button>

export let addPost = () => {  
 let newPost:PostsType = {  
 id:new ***Date***().getTime(),  
 messages:***state***.profilePage.newPostText,  
 likesCount: 0,  
 img: 'https://   
 }  
 ***state***.profilePage.posts.push(newPost)  
 ***state***.profilePage.newPostText = ''  
 renderTree(***state***)  
}  
export let updateNewPostText = (newText:string) => {  
 ***state***.profilePage.newPostText = newText  
 renderTree(***state***)  
}

# Lesson 34

1. renderTreeee = observer вместо содержимого рендерТри теперь будет обсерв

let renderTreeee = () => {  
 ***console***.log('123')  
}  
export const subscribe = (observer:any) => {  
 renderTreeee = observer  
}

1. рендерТри не изменится, перменная работает только внутри функции

export const subscribe = (observer:any) => {  
 let renderTreeee = observer  
}

# Lesson 39

1. Аналогичные записи(мы возвращаем обьект, и чтобы фигурные скобки не воспринялись как тело функции, нужно обернуть их в круглые)

export const addPostActionCreator = () => ({type: ADD\_POST})

export const addPostActionCreator = () => {  
 return {type: ADD\_POST}  
}

# Lesson 41

1.Было

if (action.type === ADD\_POST) {  
 let newPost:PostsType = {  
 id:new ***Date***().getTime(),  
 messages:state.newPostText,  
 likesCount: 0,  
 }  
 state.posts.push(newPost)  
 state.newPostText = ''  
  
 } else if(action.type === UPDATE\_NEW\_POST){  
 state.newPostText = action.newText  
}

2.Стало(аналог иф елс)

switch (action.type) {  
 case ADD\_POST:  
 let newPost: PostsType = {  
 id: new ***Date***().getTime(),  
 messages: state.newPostText,  
 likesCount: 0,  
 }  
 state.posts.push(newPost)  
 state.newPostText = '';  
 return state  
  
 case UPDATE\_NEW\_POST:  
 state.newPostText = action.newText;  
 return state  
 default:  
 return state  
}

1. Reducer.ы – это функции, которые получают свою часть стейта и экшн, и возвращает измененый стейт (вынесли функции из диспатча, для деструкторизации)
2. Автоматическая типизация(в функции нужно дописать as const):

type AddPostActionType = ReturnType<typeof addPostCreator>

export const addPostCreator = () => {  
 return {  
 type: ADD\_POST  
 } as const  
}

# Lesson 42

1. Устанавливаем редакс – yarn add redux --save
2. Redux – библиотека, с встроенными фугкциями(диспатч,сабскрайб итд), используется для хранения стейта и методов
3. Создаем стор(используя старый стор)
4. let ***store*** = ***createStore***(reducer)
5. Каждому разделу стейта присваеваем свои методы
6. let reducer = combineReducers({  
    sidebar: sidebarReducer,  
    dialogsPage: dialogsReducer,  
    profilePage: profileReducer  
   })

# Lesson 45

1. Добавляем реакт-редакс – библиотека, которая скрывает детали взаимодействия реакта и редакса(такие как стореКонтекст итд)

yarn add react-redux –save

npm install @types/react-redux

2.Создаем контейнерную компоненту(в которой лежат функции и передаем стейт ), оригинальная компонента становится еще «тупее» и универсальнее. Было

type DialogType = {  
 store:StoreType  
}  
  
function DialogsContainer(props:DialogType) {  
  
 let state = props.store.getState().dialogsPage  
  
 let onChange = (text:string) => {  
 let action = updateNewMessageCreator(text);  
 props.store.dispatch(action)  
 }  
  
  
 let onClick = () => {  
 props.store.dispatch(sendMessageCreator())  
 }  
  
 return (  
 <Dialogs  
 updateNewMessageBody={onChange}  
 sendMessage={onClick}  
 dialogsPage={state}  
 />  
 )  
}

3.стало

type MapStateToPropsType = {  
 dialogsPage: DialogsPageType  
}  
type MapDispatchToPropsType = {  
 updateNewMessageBody:(text:string) => void  
 sendMessage:() => void  
}  
const mapStateToProps = (state:ReduxStateType):MapStateToPropsType => {  
 return {  
 dialogsPage: state.dialogsPage  
 }  
}  
const mapDispatchToProps = (dispatch:Dispatch):MapDispatchToPropsType => {  
 return {  
 updateNewMessageBody: (text:string) => {  
 let action = updateNewMessageCreator(text);  
 dispatch(action)  
 },  
 sendMessage: () => {  
 dispatch(sendMessageCreator())  
 }  
 }  
}  
  
  
const ***DialogsContainer*** = ***connect***(mapStateToProps,mapDispatchToProps)(Dialogs);  
  
export default ***DialogsContainer***

3.1 Сдесь мы используем инструменты реакт-редакс. ***DialogsContainer*** «соединяет» диалог и отправляемые в него функции\стейт.

3.2 (dispatch:Dispatch) тут типизация имортируется из библиотеки редакс

4. Два варианта типизации

4.1

export type MessagesType = {  
 id: number  
 messages: string  
}  
  
export type DialogsType = {  
 id: number  
 name: string  
}  
export type DispatchType = UpdateNewMessageBodyActionType | SendMessageActionType  
  
type UpdateNewMessageBodyActionType = ReturnType<typeof updateNewMessageCreator>  
type SendMessageActionType = ReturnType<typeof sendMessageCreator>  
  
  
let initialState = {  
 dialogs: [  
 {id: 1, name: 'Dimych'},  
 {id: 2, name: 'Andrey'},  
 {id: 3, name: 'Anya'},  
 {id: 4, name: 'Sveta'},  
 {id: 5, name: 'Viktor'},  
 {id: 6, name: 'Valera'},  
 ] as DialogsType[],  
 messages: [  
 {id: 1, messages: 'Hi'},  
 {id: 2, messages: 'How are you?'},  
 {id: 3, messages: 'Good'},  
 ] as MessagesType[],  
 newMessageBody:''  
}  
export type InitialStateType = typeof initialState  
  
export const dialogsReducer = (state: InitialStateType = initialState, action: DispatchType):InitialStateType => {

4.2  
export type PostsType ={  
 id: number  
 messages: string  
 likesCount: number  
}  
export type ProfilePageType ={  
 posts: PostsType[]  
 newPostText: string  
}  
export type DispatchType = AddPostActionType | UpdatePostActionType  
type AddPostActionType = ReturnType<typeof addPostCreator>  
type UpdatePostActionType = ReturnType<typeof updateNewPostCreator>  
  
  
let initialState = {  
 posts: [  
 {  
 id: 1,  
 messages: 'Hi,how are you?',  
 likesCount: 11,  
 },  
 {  
 id: 2,  
 messages: 'its my first posts',  
 likesCount: 41,  
 },  
  
 ],  
 newPostText:''  
}  
  
export const profileReducer = (state: ProfilePageType = initialState, action: DispatchType):ProfilePageType => {

# Lesson 46

let numbers = {  
 a: 3,  
 b:{  
 c:3  
 }  
}

1.поверхностная копия let copy = {...numbers}

2.глубокая копия let deepCopy = {  
 ...numbers, b:{...numbers.b}  
}

# Lesson 47

* 1. Если раньше при изменении стейта происходила перерисовка всего дерева, то теперь рендеринг будет локальный.

Убираем:

renderTree()  
  
***store***.subscribe(renderTree)

2.Чтобы произошел рендер нужно чтобы обьект изменился, но меняться может только копия обьекта. Изменения оригинального обьекта недопустимы.

2.1 Вместо

state.posts.push(newPost)  
state.newPostText = '';  
return state

Создаем копию в profileReducer

let stateCopy = {...state}  
stateCopy.posts = {...state.posts}  
stateCopy.posts.push(newPost)  
stateCopy.newPostText =''  
  
return stateCopy

# Lesson 48

1.вместо пуш, добавляем элемент в массив таким способом:

let stateCopy = {...state,  
 newMessageBody: '',  
 messages:[...state.messages,{id: new ***Date***().getTime(), messages: body}]}

2.было:

stateCopy.messages.push({id: new ***Date***().getTime(), messages: body});  
stateCopy.newMessageBody = ''