***Lesson 32***

В файле **state.js** пишем функцию **addPost** и экспортируем ее в другие документы

export let addPost = (postMessage) => {

    let newPost = {

        id: 3,

        message: postMessage,

        likesCount: 0

    };

    state.profilePage.posts.push(newPost);

};

В index.js данную функцию импортируем

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

Далее прокидываем ее через **props** дальше

ReactDOM.render(<App state={state} addPost={addPost} />, document.getElementById('root'));

И далее прописываем тем компонентам, которым это нужно

addPost={props.addPost}

а уже в конечном пункте **Post**

  let addPost = () => {

        let text = newPostElement.current.value;

        props.addPost(text);

    };

***Lesson 33***

Т.к. отрисовка у нас происходит только один раз, новый пост мы не видим. Значит нам надо отрисовывать, “дерево” компонент, каждый раз как изменяются данные. В **index.js** объявляем функцию перерисовки и запускаем ее.

let rerenderEntireTree = ()=>{

ReactDOM.render(<App state={state} addPost={addPost} />, document.getElementById('root'));

};

rerenderEntireTree();

Компонента должна отрисовываться только тогда когда изменились данные в **state.** Создаем новый файл **render.js**

APP

Import state

import

Render.js

State.jsоы

import

Index.js

rerenderEntireTree

В **render.js** передаем **state** как параметр в функции **rerenderEntireTree**

export let rerenderEntireTree = (state) => {

    ReactDOM.render(<App state={state} addPost={addPost} />, document.getElementById('root'));

};

Так же **state** добавляем функцию в файле **state.js** и **index.js**

export let addPost = (postMessage) => {

    let newPost = {

        id: 3,

        message: postMessage,

        likesCount: 0

    };

    state.profilePage.posts.push(newPost);

    rerenderEntireTree(state);

};

export default state;

Теперь посты добавляются у нас в компоненте **Post**

Так можно обнулить **textarea** после ввода текста

  let addPost = () => {

        let text = newPostElement.current.value;

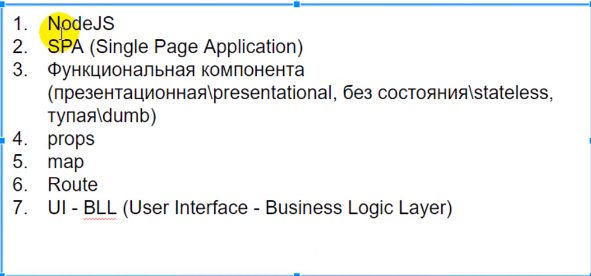
        props.addPost(text);

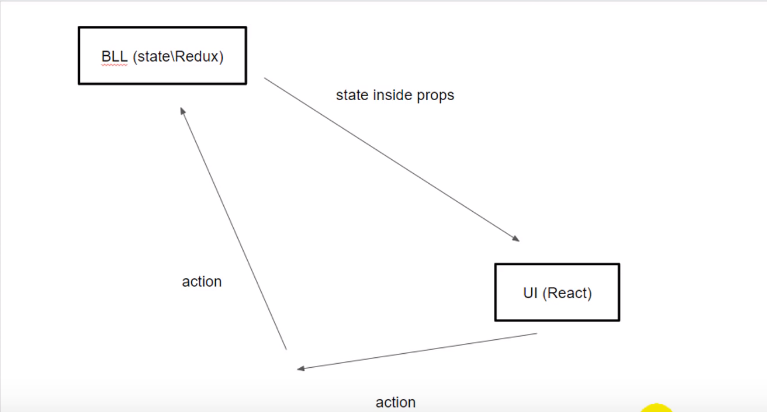
        newPostElement.current.value = '';

    }

**Ligthshot –** программа для быстрых скриншотов

**ROADMAP**





FLAX->REDUX

***Lesson 34***

**Redux –** это библиотека, которая организовывает FLUX архитектуру.

Для того, чтобы можно было вводить текст в **textarea**, нужно изменять **state**, т.к. UI должен перерисовываться только после изменения BLL. Для этого вводим обработчик изменений **onChange.**

 let onPostChange = () => {

    }

И прописываем атрибут

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

Далее указываем, что значение равно чему-то из **props**

<textarea ref={newPostElement} onPostChange={onPostChange} value={props.newPostText} />

Нам надо данное значение прокинуть наверх во все компоненты начиная с **state.js**

Добавляем в массив **ProfilePage**, новое св-во **newPostText** с каким-то значением**.**

В APP немного заменим название **state** на **profilePage**, чтобы не путаться и прокидываем **newPostText** к нужной компоненте **Post**

В **APP**

<Route path='/profile' render={() => <Profile profilePage={props.state.profilePage} addPost={props.addPost} />} />

В **Profile**

<Posts posts={props.profilePage.posts} addPost={props.addPost} newPostText={props.profilePage.newPostText} />

Далее нам нужно изменять **state** при введении данных, изменить **state** может сам **state**. Создаем там функцию **updateNewPostText,** где **newText –** то что к нам пришло из **textarea** и присваиваем это значение **newPostText,** после перерисовываем все.

export let updateNewPostText = (newText) => {

    state.profilePage.newPostText = newText;

    rerenderEntireTree(state);

};

Теперь эту функцию нам надо прокинуть вниз через **props.** В **render.js** прописываем

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

export let rerenderEntireTree = (state) => {

    ReactDOM.render(<App state={state} addPost={addPost} updateNewPostText={updateNewPostText}/>, document.getElementById('root'));

};

В APP

<Route path='/profile' render={() => <Profile profilePage={props.state.profilePage} addPost={props.addPost} updateNewPostText={props.updateNewPostText} />} />

в **Profile**

<Posts posts={props.profilePage.posts} addPost={props.addPost} newPostText={props.profilePage.newPostText} updateNewPostText={props.updateNewPostText} />

В Post вызываем эту функцию и передаем ей то значение, которое хочет ввести пользователь

let onPostChange = () => {

        let text = newPostElement.current.value;

        props.updateNewPostText(text);

    }

Теперь рассмотрим как мы зануляем данное значение. До этого мы его обнуляли принудительно, но обнулить мы его можем только если он придет в props, значит убираем от туда обнуление оно и так туда приходит.

 let addPost = () => {

        let text = newPostElement.current.value;

        props.addPost(text);

    }

И мы его можем просто обнулить в **state**, передав пустую строку

let addPost = () => {

        let text = newPostElement.current.value;

        props.addPost(text);

        props.updateNewPostText(‘’);

    }

Далее можем еще улучшить наше решение, то что мы набираем лежит и в BLL(state) и UI

Проверить это можно создав в **state** объект **window.state = state** и проверяем в консоли

Мы можем ничего не отправлять в **state** через **addPost**

let addPost = () => {

        props.addPost();

        props.updateNewPostText('');

    }

Оно и так там видит в свойстве **newPostText** и если раньше он приходил к нам как параметр, теперь мы его смотрим в **state.profilePage.newPostText**

export let addPost = () => {

    let newPost = {

        id: 3,

        message: state.profilePage.newPostText,

        likesCount: 0

    };

    state.profilePage.posts.push(newPost);

    rerenderEntireTree(state);

};

Можно улучшить код если обнулять значение не в **addPost**, а в самом **state**

export let addPost = () => {

    let newPost = {

        id: 3,

        message: state.profilePage.newPostText,

        likesCount: 0

    };

    state.profilePage.posts.push(newPost);

    state.profilePage.newPostText = '';

    rerenderEntireTree(state);

};

***Lesson 35***

Проводим рефакторинг кода. Переносим данные из **render** в **index.js** и удаляем **render.js**

В **state.js** вводим функцию, а в **index.js** ее вызываем

export const subscribe = (observer) => {

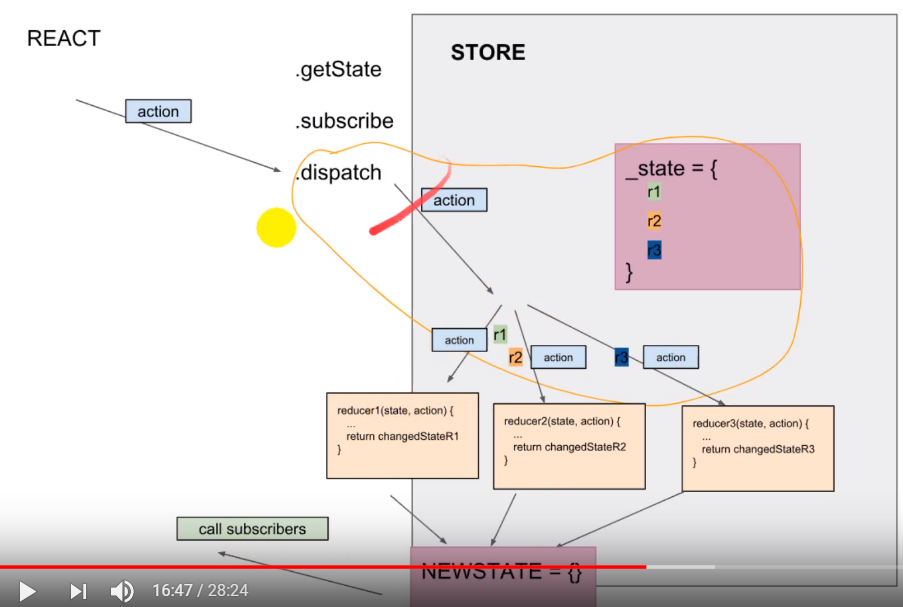
    rerenderEntireTree = observer;

}

subscribe(rerenderEntireTree);

***Lesson 40***

**Reducer** – это функция, которая принимает **state** и **action**, если нужно применяет этот **action** к **state** и возвращает **new state** или не измененный **state** если **action** не подошел.



В  **Reducer-**ах поменяли if else на switch case и в каждом варианте return state при изменении, иначе остальные case не сработают.