**Props** (Пропсы) – это объект, параметры для компоненты, которые передаются в тэге (приходят всегда)

**Компонента** – это (желательно чистая) функция которая принимает props и возвращает jsx разметку **ИЛИ** это класс который наследуется от React.Component и у которого как минимум есть метод render

**Роутинг** – отслеживание адреса (npm install react-router-dom --save)

import { BrowserRouter, Route } from 'react-router-dom';

<BrowserRouter>

<Route exact path = '/dialogs' component={Dialogs} /> (exact значит ищет ТОЧНОЕ совпадение)

</BrowserRouter>

<a href = “”></a> - это просто ссылка после которой обновляется страница

<NavLink to=”” activeClassName=”(этот класс добавляется к тэгу когда тот активен)”></NavLink> - ссылка после прохождения по которой страница не обновляется

preventDefault – отмена действия по умолчанию (это не код, просто запомнить слово)

Если данные для отрисовки компонентов находятся в index.js то нужно прокидывать их через пропсы каждой вышестоящей компоненты(index -> app -> dialogs -> dialogsItem)

Выносим все данные в state.js

**Колбэк** – функция которую мы не вызываем, а только отдаем, а потом кто то другой ее вызывает(главное мы в ручную её не вызываем)

**Redux** – state menegment (способ хранения бизнес логики проекта)

**Инкапсулирование** – сокрытие деталей от внешнего воздействия (ограничение внешнего воздействия, запрет прямого воздействия)

Какойто метод.bind(/\*объект с которым хотим связаться\*/) – чтобы this был полным объектом

**Reduser** – это чистая функция которая принимает state и action, если нужно применяет этот action к этому state и возвращает этот state, либо возвращает state который был до этого (не измененный) (функции преобразователи; делят dispatch)

**Диспатч** (dispatch) – универсальная функция для взаимодействия со store.

**Redux** – предоставляет возможность создать store

Npm install redux –save

**Контейнерная компонента** – компонента которая создается поверх основной для выведения из нее обращения к store (как понял чтобы основную компоненту можно было потом использовать повторно)

**Контекст** – позволяет передавать данные по дереву без прокидывания через пропсы

Родитель оборачивается:

<NameContext.Provider value={данные}>

</NameContext.Provider>

Принимающий (всегда ниже по дереву компонентов) оборачивается:

<NameContext.Consumer> {

(данные) (

Здесь вся jsx разметка

и где надо уже обращаемся не через пропсы, а напрямую к нужным данным

было:

{props.state.profilePage} // если в пропсах был передан уже сам state

стало:

{store.getState().profilePage}

)}

</NameContext.Consumer>

С помощью контекста данные можно передавать только контейнерным компонентам. Презентационным компонентам передаем данные по старинке через пропсы

Бибилиотека react-redux позволяет из React работать не напрямую с Redux, а через react-redux как через посредника который скрывает не нужные моменты

Получается схема:

React -> react-redux -> Redux

**Connect** (функция библиотеки react-redux) – создает контейнерную компоненту, внутри которой она рендерит призентационную компоненту и внутрь презентационной компоненты в качестве пропсов передает свойства которы передаются в первых скобках. Функция connect вызовет первую фунцкцию и передаст в неё state и вызовет вторую функцию и передаст в неё store.dispatch.bind(store)

**Идемпоте́нтность** (детерминорованность)– при повторном применении операции результат один и тот же

**Урок 47, 48 пояснение**: мы в редьюсорах раньше изменяли state напрямую, а так делать нельзя изза понятия «чистых функций». Теперь мы создаем новую копию state при выполнении каждого action и заменяем только те свойства которые и планировали изменить при таком action. Большая часть state остается не тронутой и просто копируется ссылка на него оператором расширения.

followAC = (userID) => ({ type: FOLLOW, userID }) – здесь у ключа userID значение не задаётся явно, значением этого ключа становится значение параметра имя которого должно совпадать с именем «висящего» ключа. Использование такой конструкции вне функции (простая инициализация объекта) запрещено (будет ошибка). Конструкция равносильна «, userID: userID })»

**Область видимости переменной** — это та часть программы, в которой к этой переменной можно обратиться

**DAL (data access layer)** – уровень доступа к данным. Тот программный код который работает с сервером

**Single Responsibility** – принцип единственной ответственности

**Promise** – обещание, что когда-то асинхронная операция закончится и через промис можно будет достучаться к результату

**CRUD** (Create Read Update Delete)

|| возвращает первый true или последний false

&& возвращает первый false или последний true

Существует только 3 вида преобразования : строковое, числовое, булевое. При преобразовании операторами происходит неявное преобразование типа

Методы **call** и **apply** позволяют задать контекст для выполняемой функции. Первый параметр обеих функций определяет контекст выполнения функции (то, чему будет равен **this**)

**Side Effect** – лишние действия для чистой функции которые она не должна выполнять. Это то, что неприемлимо для чистой функции. Что нарушает ее чистоту

Если мы не напишем **extends React.Component** то объект созданный с помощью класса \_\_ не будет обладать нужными умениями чтобы React мог с ним взаимодействовать

Жизненый цикл – методы которые говорят компоненте что она должна обновиться/удалиться/создаться

**PreLoader** – крутилка загрузки

Нужно быть готовым к тому, что изначально файлы не приходят (они приходят через несколько секунд за которые страница уже прорисуется)

Создание переиспользуемого кода

<Route path='/profile/:userId?' render={() => <ProfileContainer />}/> - после двоеточия добавляется имя свойства которое будет отображено в пропсах(а именно **props.match.params**) переданных компоненте от контейнерной компоненты созданной с помощью **withRouter**. Знак двоеточия говорит о том, что это имя свойства, а вопросительный знак говорит о том, что это свойство опциональное

Логин пароль к сайту с API

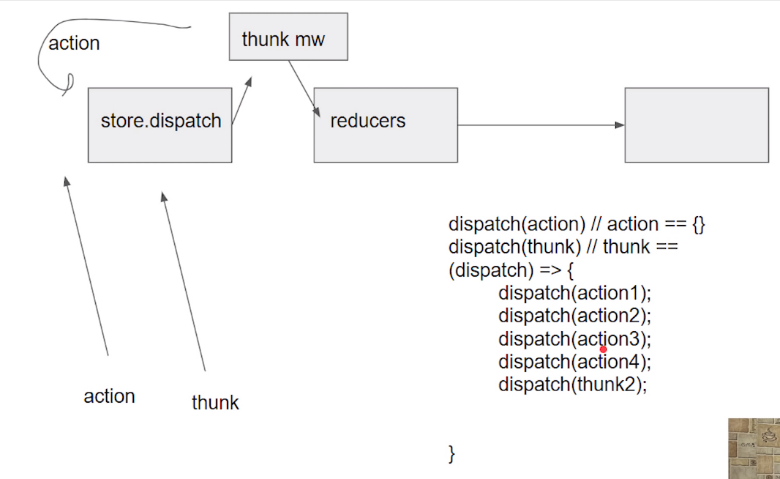
qwerty123QWE, [641027B@gmail.com](mailto:641027B@gmail.com), qwerty123

Rest API – договоренность работы с сервером

Request Method: OPTIONS - нужен для «разузнать обстановку» на сервере на который отправляется запрос. (принимает ли он вообще такой тип запросов, такие свойства)

**Thunk** (Санка) - это функция, которая делает асинхронную операцию и диспатчит обычные action`ы. Санку можно тоже задиспатчить и store поймет что с ней делать (он ее сначала запустит, а только потом пустит в редьюсеры). Санка это функция которая принимает метод dispatch. Санка может возвращать промис(но это исключение(например чтобы проверить завершился ли диспатч), лучше делать – «задиспатчил и забыл»). Каждый then возвращает совой promise

Для того чтобы redux мог принимать в редьюcеры такие санки, мы должны интегировать в него “thunk middleware”



Thunk mw будет запускать диспатч до тех пор пока в диспатче не останутся только action`ы

Единственным к кому должен обращаться UI это BLL (store)

**HOC** (High Order Component) – компонента которая принимает компоненту и возвращает контейнерную компоненту (в момент вызова hoc)

Withrouter – это тоже hoc

Connect – функция которая возвращает hoc

**compose**(

connect(mapStateToProps, mapDispatchToProps),

withAuthRedirect

)(Dialogs);

Логика: возьми Dialogs закинь в функцию withAuthRedirect получи результат и этот результат закинь в connect (получается что идут сверху вних по уровням вложенности(чем выше функция по списку, тем ближе она к export))

Локальными данными обычно делают данные которые не являются бизнес данными(не глобальные/не так важны в целом/временно нужны только отдельному компоненту)

this.forceUpdate() – заставляет react перересовать компонент принудительно

npm i redux-form

Добавить в reduxStore.js

Input меняем на **Field** (тоже является контейнерной компонентой которая рисует другую компоненту)

**reduxForm**({

form: "login" // что то типа идентификатора для формы (не относится не к html не к полю в store)

})(LoginForm)

handleSubmit прокидывается reduxForm

Ошибки валидации лежат в meta.error

**stopSubmit** это action самого redux-from (нужно импортировать). Первым аргументов указывается название формы, вторым объект в который можно положить как ошибки к отдельным инпутам(как валидация у Field) так и ошибку ко всей форме. Этот action нужно задиспатчить (можно из санки)