# React

## УСТАНОВКА YARN

Инсталлим node.js с сайта, после этого через npm инсталлим yarn:

npm i yarn -g

yarn -v

## КОМАНДЫ YARN

yarn install --production // заинсталлит только dependencies, без devDependencies

yarn install // заинсталлит все зависимости, production и development

## УСТАНОВКА WEBPCK (ЛОКАЛЬНО)

Через yarn:

yarn add webpack webpack-dev-server extract-text-webpack-plugin

yarn add babel-core babel-loader babel-preset-env babel-preset-react babel-preset-stage-2 babel-polyfill

yarn add css-loader style-loader sass-loader node-sass normalize.css

После этого нужно в **package.json** добавить скрипт,

build:dev - для девелопмнет версси, единожды запускает webpack, прогоняет все файлы через себя и babel, выливает в bundle.js (не минифицированый)

build:prod - для продакшн версси, единожды запускает webpack, прогоняет все файлы через себя и babel, выливает в bundle.js, styles.css, и фалы source-maps

dev-server - для девелопмент версии, запускает веб-сервер, все файлы держит в оперативке

"scripts": {

"build:dev": "webpack",

"build:prod": "webpack -p --env production",

"dev-server": "webpack-dev-server",

"test": "jest --config=jest.config.json"

}

"babel": {

"presets": [

"env",

"react",

"stage-2"

]

}

## WEBPACK.CONFIG.JS

Все настройки по всем установленным модулям webpack берет из файла **webpack.config.js**.

Необходимо создать файл в корне проекта, в него пишем:

const path = require('path');

const ExtractTextPlugin = require('extract-text-webpack-plugin');

module.exports = (environment) => {

const isProduction = environment === 'production';

const CSSExtract = new ExtractTextPlugin('styles.css');

return {

entry: ['babel-polyfill','./src/app.js'],

output: {

path: path.join(\_\_dirname, 'public', 'dist'),

filename: "bundle.js"

},

module: {

rules: [{

loader: 'babel-loader',

test: /\.js$/,

exclude: /node-modules/

}, {

test: /\.s?css$/,

use: CSSExtract.extract({

use: [

{

loader: 'css-loader',

options: {sourceMap: true}

},

{

loader: 'sass-loader',

options: {sourceMap: true}

}

]

})

}]

},

plugins: [CSSExtract],

devtool: isProduction ? 'source-map' : 'inline-source-map',

devServer: {

contentBase: path.join(\_\_dirname, 'public'),

historyApiFallback: true,

publicPath: '/dist/'

}

}

};

## ЗАПУСК

После этого можно запустить webpack, который возьмет файл src/app.js (из webpack.config.js) и выльет его в папку public/dist/bundle.js. Это версия для девелопмента

yarn run build:dev

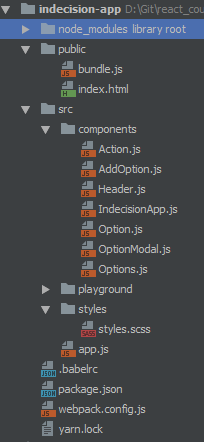
Для девелопмента, что бы запустить веб-сервер, пишем,

yarn run dev-server

## УСТАНОВКА REACT

yarn add react react-dom

## ФАЙЛОВАЯ СТРУКТУРА REACT

Каждый компонент в отдельном js-файле

**app.js** выглядит так:

import React from 'react';

import ReactDOM from 'react-dom';

import IndecisionApp from './components/IndecisionApp';

import 'normalize.css/normalize.css';

import './styles/styles.scss';

ReactDOM.render(

<IndecisionApp/>, document.getElementById('app')

);

**IndecisionApp.js** так:

import React from 'react';

import Header from './Header';

import Action from './Action';

import Options from './Options';

import AddOption from './AddOption';

export default class IndecisionApp extends React.Component {

constructor(props) {

super(props);

...

## СИНТАКСИС REACT ШАБЛОНОВ

Шаблоны React'a принимают только утверждения (statements) либо тернарные операторы, но не выражения (expressions)

true, false, null, undefined - не будут отображены

const user = {name: 'Alex', age: 17, location: 'Kiev'};

function getLocation(location) {

if (location) return <p>Location: {location}</p>;

}

// покажет имя или Анонимус, если имя undefined

// проверит, есть ли возраст и больше ли он 18, если true - отобразит строку с возрастом

// отобразит строку с локацией или вообще ничего не покажет

const template2 = (

<div>

<h1>{user.name ? user.name : 'Anonymous'}!</h1>

{(user.age && user.age >= 18) && <p>Age: {user.age}</p>}

{getLocation(user.location)}

</div>

);

const userRoot = document.getElementById('user');

ReactDOM.render(template2, userRoot);

JSX так же поддерживает шаблоны внутри шаблонов. Можно обрабатывать массивы, но не обьекты:

<ul>

{

app.options.map((option, i, arr) => {

return <li key={i}>{option}</li>

})

}

</ul>

## РЕНДЕР

Некоторые свойства HTML нельзя писать напрямую в шаблоны JSX. Например class, вместо него используется className.

<https://reactjs.org/docs/dom-elements.html> - внизу можно посмотреть список неизмененных свойств

[https://reactjs.org/docs/events.html#form-events](https://reactjs.org/docs/events.html" \l "form-events) - HTML эвенты

ReactDOM.render() работает крайне эффективно за счет виртуального DOM-дерева, потому даже если рендерить после небольшого изменения целую страничку сразу этим методом, Реакт отследит изменения и внесет их только в те части на стороне клиента, где они нужны

## КОМПОНЕНТЫ

Реакт имеет метод React.Component что является обычным родительским классом. Синтаксис:

class Header extends React.Component{

render(){

return <p>This if Header</p>

}

}

Все компоненты создаются с Прописной буквы. Если без прописной, Реакт пытается отрендерить элемент как строчный, вместо передачи класса

Можно отрендерить компонент внутри компонента:

class IndecistionApp extends React.Component {

render(){

return {

<div>

<Header/> {}

<Action/> {}

</div>

}

}

}

ReactDOM.render(<IndecistionApp/>, document.getElementById('app'));

Цикл внутри компонента с привязкой this к обработчику события. Когда функция вешается на элемент html как обработчик события, this теряется

class Options extends React.Component {

constructor(props) {

super(props);

this.handleRemoveAll = this.handleRemoveAll.bind(this);

}

handleRemoveAll() {

this.props.options.length = 0;

}

render() {

return (

<div>

<button onClick={this.handleRemoveAll}>Remove all</button>

<ol>

{

this.props.options.map((option, i, arr) => {

<Option key={i} optionText={option}/>

})

}

</ol>

</div>

);

}

}

class Option extends React.Component {

render() {

return <li key={this.props.index}>{this.props.optionText}</li>

}

}

ReactDOM.render(<Options />, document.getElementById('app'));

## СОСТОЯНИЕ КОМПОНЕНТА (COMPONENT STATE)

Реакт автоматически отслеживает изменения состояния, что бы рендерить их сразу в UI. Состояния - обычные пары ключ\значение, которые пишутся как свойства обьекта в компонент. Что бы создать состояние, нужно в конструкторе создать обьект state, а в нем задать свои свойства. Доступ к свойствам осуществляется через наследуемый метод setState:

class Counter extends React.Component {

constructor(props){

super(props);

this.handleAddOne = this.handleAddOne.bind(this);

this.state = {

count: 0

}

}

handleAddOne(){

this.setState((prevState) => {

return {

count: prevState.count + 1

}

})

}

render(){

return (

<div>

<h1>Count: {this.state.count}</h1>

<button onClick={this.handleAddOne}>+1</button>

</div>

)

}

}

ReactDOM.render(<Counter/>, document.getElementById('app'));

**Передача функций**

Что бы вложенные компоненты могли обрабатывать и изменять состояния всего приложения или других компонентов, в родительском компоненте создаются функции, которые передаются детям как обработчики:

class IndecistionApp extends React.Component {

constructor(props) {

super(props);

this.state = {

options: ['Item 1', 'Item 2', 'Item 3', 'Item 4',]

};

this.handlePick = this.handlePick.bind(this);

}

handlePick() {

const randomNum = Math.floor(Math.random() \* this.state.options.length);

const option = this.state.options[randomNum];

alert(option);

}

render() {

return (

<div>

<h1>Super app</h1>

<Action handlePick={this.handlePick} />

</div>

)

}

}

class Action extends React.Component {

render() {

return <button onClick={this.props.handlePick}>What should I do?</button>

}

}

ReactDOM.render(<IndecistionApp/>, document.getElementById('app'));

**Обработка ошибок и передача информации в функции**

Ошибка вызывается за счет того, что при успехе изменяется this.state.options, а возвращается ничего. При ошибке возвращается строка, которая является true, соответственно элемент отображается

class IndecistionApp extends React.Component {

constructor(props) {

super(props);

this.state = {

options: []

};

this.handleAddOption = this.handleAddOption.bind(this);

}

handleAddOption(option) {

if(!option){

return 'Enter valid value';

} else if(this.state.options.indexOf(option) > -1){

return 'This option already exists'

}

this.setState((prevState) => {

return {

options: prevState.options.concat(option)

}

});

}

render() {

const title = 'Indecision';

return (

<div>

<h1>Indecision</h1>

<AddOption handleAddOption={this.handleAddOption} />

</div>

)

}

}

class AddOption extends React.Component {

constructor(props){

super(props);

this.handleAddOptionHelper = this.handleAddOptionHelper.bind(this);

this.state = {

error: undefined

}

}

handleAddOptionHelper(e) {

e.preventDefault();

const option = e.target.elements.option.value.trim();

const error = this.props.handleAddOption(option);

this.setState(() => {

return { error }

})

}

render() {

return (

<div>

{this.state.error && <p>{this.state.error}</p>}

<form onSubmit={this.handleAddOptionHelper}>

<input type="text" name="option"/>

<button>Add option</button>

</form>

</div>

)

}

}

ReactDOM.render(<IndecistionApp/>, document.getElementById('app'));

## STATELESS FUNCTIONAL COMPONENT

В качестве компонента используется обыкновенная функция. Плюсы - нет лишнего кода, он быстрее и легче тестируется. Минусы - нет объекта state, т.е. нельзя менять состояние

defaultProps - передача параметров по умолчанию, если явно не указан какой-то из них

const User = (props) => {

return (

<div>

<p>name: {props.name}</p>

<p>age: {props.age}</p>

</div>

)

};

// свойство по умолчанию, если не указано в свойствах при рендере

User.defaultProps = {

name: 'Anonymous'

};

ReactDOM.render(<User name="Alex" age={26}/>, document.getElementById('app'));

## ПРОБРОС СВОЙСТВ В CHILD COMPONENTS

const Option = (props) => {

return (

<div data-option-index={props.optionIndex}>

{props.optionText}

<button

onClick={(e) => {

props.handleDeleteOptionSingle(props.optionIndex)

}}

>

remove

</button>

</div>

)

};

Если надо передать какую-то функцию детям, которые например динамически генерируются на каком-то уровне прослойке, функцию можно передавать как обычное свойство от компонента к компоненту, а на самом нижнем уровне вызвать ее анонимной функцию. Если вызвать ее напрямую, то будет передано не то, что нам нужно ( в данном случае prop.optionIndex), а event - e

## COMPONENTS LIFECYCLE METHODS

Вызвать можно только из class-based components, из stateless не выйдет

componentDidMount() - че то типа DOMContentLoaded, вызывается, когда компонент построил дерево

componentDidUpdate(prevProps, prevState) - вызвается, когда компонент обновляет props или state

componentWillUnmount() - вызывается перед удалением компонена

<https://reactjs.org/docs/react-component.html> - остальные функции тут

## ПЕРЕДАЧА ШАБЛОНА ЧЕРЕЗ СВОЙСТВА

**Подход 1.**

const Layout = (props) => {

return (

<div>

<p>Header</p>

{props.content}

<p>Footer</p>

</div>

)

};

const template = (

<div>

<h1>Page title</h1>

<p>This is my page</p>

</div>

);

ReactDOM.render(<Layout content={template}/>, document.getElementById('app'));

**Подход 2. Через props.children**

const Layout = (props) => {

return (

<div>

<p>Header</p>

{props.children}

<p>Footer</p>

</div>

)

};

ReactDOM.render(<Layout><p>This is inline</p></Layout>, document.getElementById('app'));

## THIRD-PARTY LIB IN REACT (REACT-MODULE)

Например, ставим библиотеку для модальных окон.

yarn add react-modal

После попытки запуска у меня посыпались ошибки компиляции через babel, пришлось доставить пресет и модуль:

yarn add babel-plugin-add-module-exports

yarn add babel-preset-stage-2

В .babelrc соотвественно дописываем:

{

"presets": ["env", "react", "stage-2"],

"plugins": ["transform-class-properties", "add-module-exports"]

}

Создаем модуль. Делаем отдельный файл в Components, назовем его **OptionModule**

**import React from 'react';**

**import Modal from 'react-modal';**

**const OptionModal = (props) => (**

**<Modal**

**isOpen={!!props.selectedOption}**

**onRequestClose={props.handleClearSelectedOption}**

**contentLabel="Selected Option"**

**>**

**<h3>{props.selectedOption}</h3>**

**{props.selectedOption && <p>test</p>}**

**<button onClick={props.handleClearSelectedOption}>Okay</button>**

**</Modal>**

**);**

**export default OptionModal;**

Теперь его можно спокойно импортировать и добавить в родительский модуль, как любой другой

В родительском модуле создаем две функции, которые открывают и закрывают окно модуля:

handleClearSelectedOption = () => {

this.setState(() => ({selectedOption: undefined}))

}

## СТИЛИ В REACT

Заинсталлим normalize

yarn add normalize.css

В **app.js** теперь необходимо импортировать стили

import 'normalize.css/normalize.css';

import './styles/styles.scss';

## ФАЙЛОВАЯ СТРУКТУРА СТИЛЕЙ

1. Разбиваем на компоненты так же. Создаем папку **styles**, в ее корне создаем точку входа - файл styles.scss, куда будут импортироваться все составляющие.
2. В папке styles создаем папку **base**, в ней создаем файл **\_base.scss** - будет хранить глобальные переменные, такие как размер шрифта
3. Теперь можно импортировать \_base.scss в **styles.scss**. При импорте SASS, важно опускать префикс нижнего подчеркивания и расширение scss:

@import "./base/base";

## REACT ROUTER

<https://reacttraining.com/react-router/web/guides/philosophy> - web-туториал

yarn add react-router

Пример **app.js** c базовым роутером

import React from 'react';

import ReactDOM from 'react-dom';

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

import 'normalize.css/normalize.css';

import './styles/styles.scss';

const Header = () => (

<header>

<h1>Expencify</h1>

<NavLink to="/" activeClassName="is-active" exact={true}>Main</NavLink>

<NavLink to="/create" activeClassName="is-active">Create Page</NavLink>

</header>

);

const ExpenseDashboardPage = () => (<div>This is from dashboard component</div>);

const AddExpensePage = () => (<div>This is add expense component</div>);

const NotFoundPage = () => (<div>404 - <Link to="/">Go home</Link></div>);

const routes = (

<BrowserRouter>

<div>

<Header/>

<Switch>

<Route path="/" component={ExpenseDashboardPage} exact={true}/>

<Route path="/create" component={AddExpensePage}/>

<Route path="/edit/:id" component={EditExpensePage}/>

<Route component={NotFoundPage}/>

</Switch>

</div>

</BrowserRouter>

);

**BrowserRouter** запускает роунинг. Внутри него может находится только 1 элемент!

**Header** обычный компонент без роутинга, потому он будет грузится на каждой странице

**Switch** будет последовательно проверять path каждого компонента, если path идентичен запрашиваемому - грузит страничку

**Route** - модули со страничками

**Link** - по скольку навигация через href="/" все еще ведет к полной перезагрузке страницы, Реакт переписывает обработчик событий в Link, что бы рендер был только на стороне клиента

**exact={true}** говорит считывать конкретно указанный путь, а не добавлять компонент ко всем похожим путям, которые начинаются как этот

По умолчанию, браузер пытается отрендерить страницы запросами на сервер. Т.е. запрос /create приведет к 404. Для того, что бы включить рендер с JS, необходимо сконфигурировать сервер так, что бы все неизвестные запросы переадресовывались на index.js. В **webpack.config.js** нужно добавить:

devServer: {

contentBase: path.join(\_\_dirname, 'public'),

historyApiFallback: true

}

Пример выноса компонента в отдельную директорию и его экспорт:

import React from 'react';

const EditExpensePage = (props) => {

return (<div>Editing the expense with id of {props.match.params.id}</div>)

};

export default EditExpensePage;

# REDUX

<http://redux.js.org/>

Библиотека создана что бы обойти "проброс" свойств и состояний в Реакте от родительского компонента всем последующим, что бы передать его последнему. Потому Redux сделали глобальный обьект (лол) Redux Store, в котором хранятся все значения со всего приложения (лол)

yarn add redux

Для первоначального создания state в Redux используется метод createStore. В метод обязательно надо передать функцию, которая вернет обьект, можно пустой. Это и есть инициализация базового state

import {createStore} from 'redux';

// для ясности вернем обьект state по умолчанию

const store = createStore((state = {count: 0}) => {

return state

});

console.log(store.getState()); // вернет весь обьект state

## REDUX ACTIONS

**Actions** описывают факт того, что что-то произошло, но не описывают как состояние приложения поменялось

Изменять состояния можно через метод dispatch, в котором можно задавать произвольные значения в свойcтве type. Конвенция говорит, что надо писать все слова в верхнем регистре, а разделять их нижним подчеркиванием

**Важно!** Не надо менять само состояние state.count! Это может привести к ошибкам.

**state.concat**

Потому вместо state.push надо использовать state.concat, тк первый вариант ИЗМЕНЯЕТ первоначальный массив, а второй - возвращает новый массив, но старый не изменяет

**...spread**

**...** оператор так же не изменяет первоначальный массив:

const names = ['Alex', 'Marina'];

console.log([...names, 'Mike']); // Alex, Marina, Mike

console.log(names); // Alex, Marina

Пример базовых actions

**import {createStore} from 'redux';**

**const store = createStore((state = {count: 0}, action) => {**

**switch (action.type){**

**case 'INCREMENT':**

**const incrementBy =**

**typeof action.incrementBy === 'number' ? action.incrementBy : 1;**

**return { count: state.count + incrementBy};**

**case 'DECREMENT':**

**return { count: state.count - 1};**

**case 'RESET':**

**return { count: 0};**

**default:**

**return state;**

**}**

**});**

**const unsubscribe = store.subscribe(() => console.log(****store.getState()));**

**store.dispatch({type: 'INCREMENT', incrementBy: 5});**

unsubscribe();

**dispatch** передает значения, обрабатывать их нужно самому. *type* - обязательное значение, остальные произвольны

**subscribe** вызывается каждый раз, когда state изменяется. Что бы отписаться, нужно присвоить функции имя, а потом вывзвать ее без аргументов:

const unsubscribe = store.subscribe(() => console.log(store.getState()));

unsubscribe();

## REDUX REDUCERS

**Reducers** описывают, как поменялось состояние приложения после прихода **actions**

1. Reducers - чистые (pure) функции, т.е. они не оперируют данными извне, изменяют только те данные, которые переданы им внутрь
2. Никогда не изменяются state или action. Не стоит изменять state напрямую! reducer должен вернуть объект.

Пример выше можно переписать так:

const countReducer = ((state = {count: 0}, action) => {

switch (action.type){

case 'INCREMENT':

return { count: state.count + action.incrementBy};

default:

return state;

}

});

const store = createStore(countReducer);

Reducers можно разделять на несколько небольших, что бы не передавать чудовищный обьект везде. Для этого необходимо подключить метод redux - **combineReducers:**

import {createStore, combineReducers} from 'redux';

const expensesReducerDefaultState = [];

const expensesReducer = (state = expensesReducerDefaultState, action) => {

switch (action.type){

default:

return state;

}

};

const filtersReducerDefaultState = {

text: '',

sortBy: 'date',

startDate: undefined,

endDate: undefined

};

const filtersReducer = (state = filtersReducerDefaultState, action) => {

switch (action.type){

default:

return state;

}

};

const store = createStore(combineReducers({

expenses: expensesReducer,

filters: filtersReducer,

}));

console.log(store.getState());

Пример сложной action

const addExpense = (

{

description = '',

note = '',

amount = 0,

createdAt = 0

} = {}) => ({

type: 'ADD\_EXPENSE',

expense: {

id: uuid(),

description,

note,

amount,

createdAt

}

});

// Expenses Reducer

const expensesReducerDefaultState = [];

const expensesReducer = (state = expensesReducerDefaultState, action) => {

switch (action.type) {

case 'ADD\_EXPENSE':

return [...state, action.expense];

default:

return state;

}

};

// Combine Reducers

const store = createStore(combineReducers({

expenses: expensesReducer,

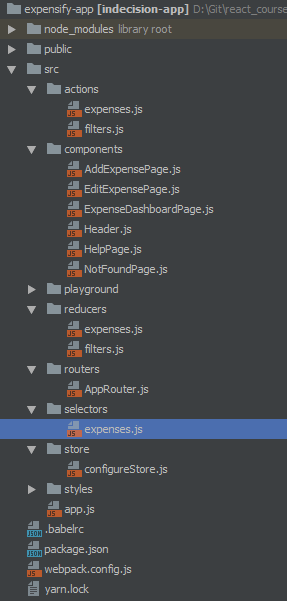
filters: filtersReducer,

}));

const unsubscribe = store.subscribe(() => console.log(store.getState()));

store.dispatch(addExpense({description: 'Rent', amount: 100}));

## СТРУКТУРА REACT ПРИЛОЖЕНИЯ

Разделим все по папкам. Вход app.js будет импортировать:

**Store**, функция-комбайн, которая соединяет reducers и возвращает store. Этот файл так же импортирует reducers

**Reducers**, которые изменяют state приложения

**Actions**, которые описывают для reducers что надо изменить

**Selectors**, возвращает отфильтрованные обьекты (если фильтры переданы)

## HIGHER ORDER COMPONENT

Это встроенный в React механизм рендера компонентов, который рендерит другие компоненты. Обертка для компонентов. Например, показать определенный компонент или нет в зависимости от того, залогинен пользователь или нет

const Info = (props) => (

<div>

<h1>Info</h1>

<p>The info is: {props.info}</p>

</div>

);

const withAdminWarning = (WrappedComponent) => {

return (props) => (

<div>

{props.isAdmin && <p>This is private info. Please dont share</p>}

<WrappedComponent {...props}/>

</div>

)

};

const AdminInfo = withAdminWarning(Info);

ReactDOM.render(<AdminInfo isAdmin={true} info="details"/>, document.getElementById('app'));

# REACT WITH REDUX

Заинсталлим библиотеку для коннекта react и redux <https://github.com/reactjs/react-redux>

yarn add react-redux

Теперь передадим свойства Redux Store в компоненты React. При каждом изменении Redux state, приложение будет реагировать и менять данные в html на лету

1. **app.js** будет включать такой фрагмент:

import {Provider} from 'react-redux'; // подключили библиотеку react-redux

import AppRouter from './routers/AppRouter.js'

import configureStore from './store/configureStore';

const store = configureStore();

// передаем store в Provider как свойство

const jsx = (

<Provider store={store}>

<AppRouter/>

</Provider>

);

ReactDOM.render(jsx, document.getElementById('app'));

1. В **AppRouter.js** страница вызвается так:

<Route path="/" component={ExpenseDashboardPage} exact={true}/>

1. **ExpenseDashboardPage.js** будет выглядеть так:

import React from 'react';

import ExpenseList from './ExpenseList';

const ExpenseDashboardPage = () => (<div><ExpenseList/></div>);

export default ExpenseDashboardPage;

1. **ExpenseList.js** включает такой код:

import React from 'react';

import {connect} from 'react-redux';

const ExpenseList = (props) => (

<div>

<h1>ExpenseList</h1>

{props.filters.text}

{props.expenses.length}

</div>

);

const mapStateToProps = (state) => {

return {

expenses: state.expenses,

filters: state.filters

}

};

// сначала передаем функцию, которая достает данные из Redux Store

// потом указываем, куда эти данные нужно вывести

export default connect(mapStateToProps)(ExpenseList);

## CONTROLLED INPUT - ОБРАБОТКА СОБЫТИЙ В REACT C REDUX

**Пример обработки input,**

который бы пересылал значение в redux store и что-то с ним делал.

Что бы значение в input менялось, необходимо менять состояние redux. Потому

а) достаем filters и передаем их в компонент

б) значение по умолчанию будет то, которое на данный момент установленно в filters.text

в) dispatch как функция отправки в redux store передается через connect по умолчанию, потому свойства dispatch доступно сразу. Передаем через этот метод установленное юзером значение в функцию фильтров, которая сортирует или отображает только отфилтрованные элементы

import React from 'react';

import {connect} from 'react-redux';

import {setTextFilter} from '../actions/filters';

const ExpenseListFilters = (props) => (

<div>

<input type="text" value={props.filters.text} onChange={(e) => {

props.dispatch(setTextFilter(e.target.value));

}}/>

</div>

);

const mapStateToProps = (state) => {

return {

filters: state.filters

};

};

export default connect(mapStateToProps)(ExpenseListFilters);

**Пример обработки клика**

Тут ничего фильтровать не надо, потому просто подключаем connect, который передает функцию dispatch по умолчанию. Остальные значения были вытащены из redux store в app.js и из цикла в другом файле, который передавал сразу весь обьект expenses. В этом примере он сразу деструктуризирован

Цикл из компонента **ExpensesList**, который добавляет в props все свойства из expense

{props.expenses.map((expense, i , arr) => {

return <ExpenseListItem key={expense.id} {...expense}/>

})}

Код самого файла **ExpenseListItem**

import React from 'react';

import {connect} from 'react-redux';

import {removeExpense} from '../actions/expenses';

const ExpenseListItem = ({description, amount, createdAt, dispatch, id}) => (

<div>

<h3>{description}</h3>

<p>{amount} - {createdAt}</p>

<button onClick={(e) => dispatch(removeExpense({id})) }>

Remove

</button>

</div>

);

export default connect()(ExpenseListItem);

## КАЛЕНДАРИК AIRBNB - REACT-DATES

<https://github.com/airbnb/react-dates>

**react-dates** - календарь для работы с датами, передает внутри обьект из библиотеки moment и работает с ним, потому так же нужно заинсталлить ее

На реакте 16 инсталляция немного запутана. Нужно заинсталлить несколько зависимостей. Сначала инсталлим саму библиотеку и ее зависимости. Для сбора даты будем moment.js

yarn add moment react-dates react-addons-shallow-compare

Теперь нужно заинсталлить приколы для babel. Прописывать в .babelrc их не нужно! будут ошибки

yarn add babel-plugin-inline-react-svg babel-plugin-transform-replace-object-assign

yarn add babel-preset-airbnb

В файле самого компонента (**ExpenseForm.js**) нужно импортировать initialize, иначе будет ошибка в стилях:

import React from 'react';

import moment from 'moment';

import {SingleDatePicker} from 'react-dates';

import 'react-dates/initialize';

import 'react-dates/lib/css/\_datepicker.css';

# ТЕСТИРОВАНИЕ REACT + REDUX

## БАЗОВЫЙ ПРИМЕР

Тесты проводятся с использованием Jest <https://facebook.github.io/jest/>

Эта библиотека уже содержит встроенный expect, потому дополнительно его инсталлить не надо

yarn add jest

В **package.json** дописываем:

"scripts": {

"dev-server": "webpack-dev-server",

"test": "jest"

},

По умолчанию тесты запускаются на все файлы, имя которых оканчивается на test.js. Потому создаем отдельную директорию tests, куда добавляем например файл **add.test.js**:

const add = (a, b) => a + b;

test('should add two numbers', () => {

const result = add(3, 4);

expect(result).toBe(7);

})

После этого тесты можно запускать через

yarn test

Запуск в режиме наблюдателя

Можно как дописать в **package.json** ("test": "jest --watch"), так и указать в командной строке. Лучше через командную строку, тк нет необходимости запускать наблюдателя, если приложение тестируется например перед коммитом.

yarn test --watch

На винде была ошибка, в багтрекере подсказали, что можно запускать через

yarn test --watchAll

## ТЕСТИРОВАНИЕ КОМПОНЕНТОВ

Тестирование компонентов реакт происходит благодаря библиотеке **react-test-renderer** и **enzyme**

yarn add react-test-renderer

Библиотека дает два метода тестирования shallow и fullDOM. Отличие в том, что shallow можно использовать только на 1 компоненте, а для цепочки компонентов подходит fullDOM

## ВАРИАНТ 1: REACT-TEST-RENDERER TESTING

Это очень простая библиотека с несколькими методами.

Для того, что бы оттестировать элемент, используются снимки (snapshot) состояний компонента в том виде, в котором они должны быть если все хорошо. Для снимков используется метод библиотеки Jest - toMatchSnapshot. При первом запуске тестов создаются снимки компонентов и помещаются той же директории

import React from 'react';

import ReactShallowRenderer from 'react-test-renderer/shallow';

import Header from '../../components/Header';

test('should render Header correctly', () => {

const renderer = new ReactShallowRenderer();

renderer.render(<Header/>);

expect(renderer.getRenderOutput()).toMatchSnapshot();

console.log(renderer.getRenderOutput());

});

getRenderOutput - отображает объект со свойствами

toMatchSnapshot - создает \ сравнивает снимки компонента в памяти с текущим

ы

Теперь, если изменить компонент будет ошибка. Что бы подтвердить изменения и убрать ошибку, нужно нажать 'u'

## ВАРИАНТ 2: ENZYME TESTING

<http://airbnb.io/enzyme/>

В отличии от react-test-renderer, более сложная библиотека от airbnb для тестирования компонентов

enzyme - сама библиотека

adapter - необходим для включения react v16 фич

raf - request animation frame - для того что бы тестирование было таким, как это видит браузер

enzyme-to-json - выводит в snapshot только нужную инфу по компоненту

yarn add enzyme enzyme-adapter-react-16 raf enzyme-to-json

После этого создадим **в корне директории для тестов** конфигурационный файл **setupTests.js**

import Enzyme from 'enzyme';

import Adapter from 'enzyme-adapter-react-16';

Enzyme.configure({

adapter: new Adapter()

});

После этого необходимо **в директории проекта** создать файл **jest.config.json**, который подгрузит перед началом тестов необходимые файлы:

{

"setupFiles": [

"raf/polyfill",

"<rootDir>/src/tests/setupTests.js"

],

"snapshotSerializers": ["enzyme-to-json/serializer"]

}

Теперь нужно указать в **package.json** для jest, что бы он подгружал этот конфигурационный файл при запуске. Меняем скрипт "test" на следующее:

"scripts": {

"test": "jest --config=jest.config.json"

},

Теперь можно приступать к тестированию

import React from 'react';

import {shallow} from 'enzyme';

import Header from '../../components/Header';

test('should render Header correctly', () => {

const wrapper = shallow(<Header />);

enzyme-to-json expect(wrapper).toMatchSnapshot();

expect(wrapper.find('h1').length).toBe(1);

expect(wrapper.find('h1').text()).toBe('Expensify');

});

Похоже на jQuery. Документация: <http://airbnb.io/enzyme/docs/api/>

## ТЕСТИРОВАНИЕ ДИНАМЕЧЕСКИХ КОМПОНЕНТОВ ЧЕРЕЗ ENZYME

Поскольку в Enzyme не нужно подключать реальный store, для начала те компоненты, которые эспортировались по умолчанию только как connect(mapStateToProps)(component) экспортировать так же как именнованые. Это нужно, что бы можно было их непосредственно импортировать в файл тестов:

export const Component = (props) => (

<div>Test</div>

);

const mapStateToProps = ...;

export default connect(mapStateToProps)(Component);

**Пример тестинга евета**, показ\сокрытие сообщения об ошибке, если форма заполнена неправильно:

test('should render error for invalid from submission', () => {

const wrapper = shallow(<ExpenseForm />);

expect(wrapper).toMatchSnapshot();

wrapper.find('form').simulate('submit', {

preventDefault: () => {}

});

expect(wrapper.state('error').length).toBeGreaterThan(0);

expect(wrapper).toMatchSnapshot();

});

Snapshot'ов может быть сколько угодно, в соответствующем файле в папке snapshots они будут перечислены порядковыми номерами. В данном случае мы:

1. сначала проверям, что в форме не отображается сообщения об ошибке
2. сабмитаем форму без данных
3. проверяем, что сообщения об ошибке появилось

## MOCKING UP THE 3RD-PARTY LIBRARIES

Иногда нужно подменить подключаемые библиотеки для тестов, например moment.js, для того, что бы даты в сравниваемых объектах были одинаковы.

Для этого рядом с файлом, где предполагается использовать подмену, создается файл **\_\_mocks\_\_,** куда складываются все файлы с измененными библиотеками.

const moment = require.requireActual('moment');

export default (timestamp = 0) => {

return moment(timestamp);

}

## ТЕСТИРОВАНИЕ C MOCK

Описание методов обычно начинается с .toHaveBeen...

<https://facebook.github.io/jest/docs/en/expect.html>

Бываю случаи, когда нужно оттестировать, какие данные передаются в функцию, например:

onSubmit = (e) => {

e.preventDefault();

if (!this.state.description || !this.state.amount) {

this.setState(() => ({error: 'Please provide description and amount'}));

} else {

this.setState(() => ({error: ''}));

this.props.onSubmit({

description: this.state.description,

amount: parseFloat(this.state.amount, 10) \* 100,

createdAt: this.state.createdAt.valueOf(),

note: this.state.note

})

}

};

render() {

return (

<form onSubmit={this.onSubmit}>

...

Это нужно, когда функция оработчик вызывается из родительского элемента

<ExpenseForm

onSubmit={(expense) => {

props.dispatch(addExpense(expense));

props.history.push('/');

}}

**Базовый пример**

Jest имеет несколько методов для тестирования как передается спай и с какими аргументами

test('should call onSubmit prop for valid form submission',() => {

const onSubmitSpy = jest.fn();

onSubmitSpy('Mike', 'Jen');

expect(onSubmitSpy).toHaveBeenCalledWith('Mike', 'Jen');

});

# ПОДКЛЮЧЕНИЕ FIREBASE К REDUX

Для начала надо установить redux-thunk, который позволяет передавать в actions функции

yarn add redux-thunk

Сделаем файл подключения к БД **src/firebase/firebase.js**. Настройки можно найти в Rules на сайте Firebase, сюда только подставить значения. Экспортируем 2 переменных - firebase для того, что бы были доступны другие его методы и database для операций с самой базой данных

import \* as firebase from 'firebase';

const config = {

apiKey: "AIzaSyCVJC43rOE3hlgUid3JzzL3-4uZs17ItJs",

authDomain: "expensify-b01a1.firebaseapp.com",

databaseURL: "https://expensify-b01a1.firebaseio.com",

projectId: "expensify-b01a1",

storageBucket: "expensify-b01a1.appspot.com",

messagingSenderId: "183776196657"

};

firebase.initializeApp(config);

const database = firebase.database();

export {firebase, database as default};

Подключим БД к redux store в **src/store/configureStore/js.**

applyMiddleware необходим для подключения middlware

compose - фоллбек если среда не поддерживает основной метод

composeEnhancers(applyMiddleware(thunk)) - нужно для того, что бы в dev tools в браузере нормально отображалась структура redux:

import {createStore, combineReducers, applyMiddleware, compose} from 'redux';

import thunk from 'redux-thunk';

import expensesReducer from '../reducers/expenses';

import filtersReducer from '../reducers/filters';

// for browser dev tools support firebase

const composeEnhancers = window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_COMPOSE\_\_ || compose;

export default () => {

const store = createStore(combineReducers({

expenses: expensesReducer,

filters: filtersReducer,

}),

// for browser dev tools support firebase

composeEnhancers(applyMiddleware(thunk))

// window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_ && window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_()

);

return store;

}

После этого можно переделать например метод **addExpense.** В организационной структуре react-redux не принято выносить логику работы с базой данных в компоненты, потому править будем только actions: **actions/expenses.js**

// Add expense action

export const addExpense = (expense) => ({

type: 'ADD\_EXPENSE',

expense

});

export const startAddExpense = (expenseData = {}) => {

return (dispatch) => {

// destructuring properties from expenseData

// const for every name in obj (const description = ''; const note = '')

const {description = '', note = '', amount = 0, createdAt = 0} = expenseData;

const expense = {description, note, amount, createdAt};

// push into Firebase and get back generated ID of expense

database.ref('expenses').push(expense).then((ref) => {

// dispatch expense into redux store by passing an expense into addExpense

dispatch(addExpense({id: ref.key, ...expense}))

})

};

};

После этого, во всех методах следует заменить addExpense на startAddExpense. По сути он подменяет addExpense, задает дефолтные настройки для обьекта expense, пушит обьект в БД и передает результат уже с сгенерированным ID в dispatch внутри функции addExpense, которая сначала вызывается, возвращая нужный обьект

## ТЕСТИРОВАНИЕ С УЧЕТОМ ИСПОЛЬЗОВАНИЯ БАЗЫ ДАННЫХ

Для того, что бы подменить хранилище redux, используем mock версию. <http://arnaudbenard.com/redux-mock-store> . Заинсталлим:

yarn add redux-mock-store

Проверять результат будем на промиссах, потому нужно что бы промиссы возвращали промисс для последующей цепочки. Для этого нужно добавить return в функцию, которую создали выше, **actions/expenses.js** - > startAddExpense

return database.ref('expenses').push(expense).then((ref) => {

dispatch(addExpense({id: ref.key, ...expense}))

});

После этого создаим test функцию в файле **tests\actions\expenses.test.js**

**done()** необходимо передавать, что бы тест знал, когда проверять expect, иначе будет постоянный success

**createMockStore()** создает фейковое хранилище со всеми методами, как и у обычного, только результат передает в resolve промисса

import configureMockStore from 'redux-mock-store';

import thunk from 'redux-thunk';

import {startAddExpense, addExpense} from '../../actions/expenses';

import expenses from '../fixtures/expenses';

import database from '../../firebase/firebase';

test('should add expense with defaults to database and store', (done) => {

const store = createMockStore();

const defaultExpense = {description: '', note: '', amount: 0, createdAt: 0};

store.dispatch(startAddExpense({})).then(() => {

const actions = store.getActions();

expect(actions[0]).toEqual({

type: 'ADD\_EXPENSE',

expense: {id: expect.any(String), ...defaultExpense}

});

return database.ref(`expenses/${actions[0].expense.id}`).once('value');

}).then((snapshot) => {

expect(snapshot.val()).toEqual(defaultExpense);

done();

})

});

## ПОДКЛЮЧЕНИЕ ТЕСТОВОЙ БД

Для того, что бы подключить тестовую БД вместо настоящей, для начала надо установить общее тестовое окружение (process.env.NODE\_ENV). Так, в package.json можно было бы прописать, что тесты запускаются при этом тестовом окружении, и в зависимости от значения это переменной, подставлять разные конфигурационные настройки, в том числе и в firebase.js. Кросс-ОС способа поставить окружения нет, потому заинсталлим вспомогательный модуль <https://www.npmjs.com/package/cross-env>

yarn add --dev cross-env

После этого изменим тестовый скрипт в **package.json**

"scripts": {

"test": "cross-env NODE\_ENV=test jest --config=jest.config.json"

},

Обычно настроечные файлы с секретной информацией не коммитятся, потому создаются отдельные файлы в корне проекта, куда все это выносится

Создадим в корне проекта 2 файла: .env.development и .env.test

В **.env.development** выносим все настройки из firebase.js в таком формате (без кавычек!)

FIREBASE\_API\_KEY=AIzaSyCVJC43rOE3hlgUid3JzzL3-4uZs17ItJs

FIREBASE\_AUTH\_DOMAIN=expensify-b01a1.firebaseapp.com

FIREBASE\_DATABASE\_URL=https://expensify-b01a1.firebaseio.com

FIREBASE\_PROJECT\_ID=expensify-b01a1

FIREBASE\_STORAGE\_BUCKET=expensify-b01a1.appspot.com

FIREBASE\_MESSAGING\_SENDER\_ID=183776196657

Создаем новый проект в браузере для Firebase, выставляем все права в true, копируем конфиг настройки и в **.env.test** в таком же формате подставляем новые значения

Для того, что бы парсить и читать этот файл есть npm модуль dotenv, инсталлим:

yarn add --dev dotenv

После этого в **webpack.config.**js добавим переменную окружения и будем читать настройки. Так же, нужно передавать эти ключи на строну клиента, если клиент напрямую работает с БД. Всю переменную целиком передавать не надо в целях безопасности, потому передадим только этих 6 ключей. Для этого используем встроенный в webpack метод DefinePlugin. Каждую переменную можно обернуть в JSON.stringify что бы создавалась строка в строке:

const path = require('path');

const ExtractTextPlugin = require('extract-text-webpack-plugin');

const webpack = require('webpack');

process.env.NODE\_ENV = process.env.NODE\_ENV || 'development';

if(process.env.NODE\_ENV === 'test'){

require('dotenv').config({ path: '.env.test' });

} else if(process.env.NODE\_ENV === 'development') {

require('dotenv').config({ path: '.env.development' });

}

module.exports = (environment) => {

const isProduction = environment === 'production';

const CSSExtract = new ExtractTextPlugin('styles.css');

return {

entry: "./src/app.js",

output: {

path: path.join(\_\_dirname, 'public', 'dist'),

filename: "bundle.js"

},

module: {

rules: [{

loader: 'babel-loader',

test: /\.js$/,

exclude: /node-modules/

}, {

test: /\.s?css$/,

use: CSSExtract.extract({

use: [

{

loader: 'css-loader',

options: {sourceMap: true}

},

{

loader: 'sass-loader',

options: {sourceMap: true}

}

]

})

}]

},

plugins: [

CSSExtract,

new webpack.DefinePlugin({

'process.env.FIREBASE\_API\_KEY':

JSON.stringify(process.env.FIREBASE\_API\_KEY),

'process.env.FIREBASE\_AUTH\_DOMAIN':

JSON.stringify(process.env.FIREBASE\_AUTH\_DOMAIN),

'process.env.FIREBASE\_DATABASE\_URL':

JSON.stringify(process.env.FIREBASE\_DATABASE\_URL),

'process.env.FIREBASE\_PROJECT\_ID':

JSON.stringify(process.env.FIREBASE\_PROJECT\_ID),

'process.env.FIREBASE\_STORAGE\_BUCKET':

JSON.stringify(process.env.FIREBASE\_STORAGE\_BUCKET),

'process.env.FIREBASE\_MESSAGING\_SENDER\_ID':

JSON.stringify(process.env.FIREBASE\_MESSAGING\_SENDER\_ID)

})

],

devtool: isProduction ? 'source-map' : 'inline-source-map',

devServer: {

contentBase: path.join(\_\_dirname, 'public'),

historyApiFallback: true,

publicPath: '/dist/'

}

}

};

После этого нужно заменить в конфиге **src/firebase/firebase.js** захардкоженные значения значниями из переменной:

import \* as firebase from 'firebase';

const config = {

apiKey: process.env.FIREBASE\_API\_KEY,

authDomain: process.env.FIREBASE\_AUTH\_DOMAIN,

databaseURL: process.env.FIREBASE\_DATABASE\_URL,

projectId: process.env.FIREBASE\_PROJECT\_ID,

storageBucket: process.env.FIREBASE\_STORAGE\_BUCKET,

messagingSenderId: process.env.FIREBASE\_MESSAGING\_SENDER\_ID

};

firebase.initializeApp(config);

const database = firebase.database();

export {firebase, database as default};

Для тестов надо изменить так же файл созданный ранее **/tests/setupTests.js**

import Enzyme from 'enzyme';

import Adapter from 'enzyme-adapter-react-16';

import DotEnv from 'dotenv';

DotEnv.config({path: '.env.test'});

Enzyme.configure({

adapter: new Adapter()

});

Теперь добавим секретные файлы в **.gitignore** (в ебаном гите это не работает - иди нахуй сучара)))

.idea/

node\_modules/

public/dist

.env.test

.env.development

## ДОБАВЛЕНИЕ ПРОДАКШН КОНФИГА ДЛЯ HEROKU

Все созданные выше переменные должна так же передаваться на heroku, это можно сделать с помощью командной строки

heroku config // посмотреть текущие переменные

heroku config:set KEY=value // установить переменную KEY

heroku config:unset KEY // удалить переменную KEY

Берем все переменные из .env.development и переносим в конфиг heroku:

heroku config:set FIREBASE\_API\_KEY=AIzaSyCVJC43rOE3hlgUid3JzzL3-4uZs17ItJs

# АУТЕНТИФИКАЦИЯ ЧЕРЕЗ FIREBASE

Заходим на продакшн-версию проекта, там заходим во вкладку authentication, выбираем любую соц сеть - сохранить.

Теперь в конфигурационный файл **firebase/firebase.js** добавим функцию:

const googleAuthProvider = new firebase.auth.GoogleAuthProvider();

export {firebase, googleAuthProvider, database as default};

В actiuons создадим файл **actions/auth.js,** где создадим функцию startLogin для логина через firebase:

import {firebase, googleAuthProvider} from '../firebase/firebase';

export const startLogIn = () => {

return () => {

return firebase.auth().signInWithPopup(googleAuthProvider);

};

};

Теперь можно создать компонент, который будет логинить: **components/LoginPage.js**

import React from 'react';

import { connect } from 'react-redux';

import { startLogIn } from "../actions/auth";

export const LoginPage = ({startLogIn}) => (

<div>

<button onClick={startLogIn}>Login</button>

</div>

);

const mapDispatchTpProps = (dispatch) => ({

startLogIn: () => dispatch(startLogIn())

});

export default connect(undefined, mapDispatchTpProps)(LoginPage);

Теперь при клике, будет открываться попап окно с логином через гугел.

Для того, что бы отследить процесс логина\логаута, в можно в корневой файл добавить обработчик эвента, в данном случае в **app.js**

import { firebase } from './firebase/firebase';

firebase.auth().onAuthStateChanged((user) => {

if(user){

console.log('log in')

} else {

console.log('log out')

}

});

## РОУТИНГ ПО ИСТОРИИ ПЕРЕХОДОВ ИЗ ЛЮБОЙ ТОЧКИ ПРИЛОЖЕНИЯ

Обычно объект с историей History API доступен только внутри BrowserRouter. Что бы обойти это ограничение и использовать history API где угодно (например для редиректа на страницу, на которой находился пользователь после его логина) нужно заинсталлить модуль

yarn add history

После этого надо инициализировать history и поменить BrowserRouter на обычный Router в **routers/AppRouter.js**

...

import createHistory from 'history/createBrowserHistory';

export const history = createHistory();

// replaced BrowserRouter to Router and pass history in

const AppRouter = () => (

<Router history={history}>

<div>

<Header/>

<Switch>

<Route path="/" component={LoginPage} exact={true}/>

...

Создадим новые actions в **actions/auth.js**

export const login = (uid) => ({type: 'LOGIN', uid});

export const logout = (uid) => ({type: 'LOGOUT'});

Создадим новый reducer: **reducers/auth.js**

export default (state = {}, action) => {

switch (action.type) {

case 'LOGIN':

return {

uid: action.uid

};

case 'LOGOUT':

return {};

default:

return state;

}

};

Так же нужно добавить только что созданный reducer в **store/configureStore.js:**

import authReducer from '../reducers/auth';

export default () => {

const store = createStore(combineReducers({

expenses: expensesReducer,

filters: filtersReducer,

auth: authReducer

}),

Теперь можно изменить код аутентификации в **app.js.** Тут меняем рендер приложения, что бы оно не рендерилось каждый раз, когда юзер логинится или выходит, а только редиректило:

// import history

import AppRouter, {history} from './routers/AppRouter.js';

const store = configureStore();

const jsx = (

<Provider store={store}>

<AppRouter/>

</Provider>

);

// avoid rendering app every time after user log in

let hasRendered = false;

const renderApp = () => {

if(!hasRendered){

ReactDOM.render(jsx, document.getElementById('app'));

hasRendered = true;

}

};

// waits until database sends info about authentication

ReactDOM.render(<p>Loading...</p>, document.getElementById('app'));

// listen when firebase check if user logged in

firebase.auth().onAuthStateChanged((user) => {

// if user logged in, firebase sends back user object in promise

if (user) {

store.dispatch(login(user.uid));

// get data from Firebase

store.dispatch(startSetExpenses()).then(() => {

// then render page

renderApp();

// if user on start page - redirect him to expenses list

if(history.location.pathname === '/'){

history.push('/dashboard');

}

});

} else {

store.dispatch(logout());

// if user logged out or not logged in - redirect to "/"

renderApp();

history.push('/');

}

});

Dispatch нужно использовать именно тут, а не как обычно в actions, потому что startLogin из actions вызывается только когда юзер клацает по кнопка логин\выход, а нам нужно, что бы инфа про это записывалсь в store уже при первом посещении юзером странички.

## СОЗДАНИЕ ПРИВАТНЫХ СТРАНИЧЕК

Сейчас даже если юзер не авторизировать, он может пройти по ссылке в меню и попасть на страницу (например /create). Что бы это не было, нужно создать отдельный компонент, который будет проверять, залогинен ли юзер и после этого делать редирект. Потому в некоторых случаях надо подменить обычный роутер на измененный. Изменим **routers/AppRouter:**

import PrivateRoute from './PrivateRoute';

export const history = createHistory();

const AppRouter = () => (

<Router history={history}>

<div>

<Header/>

<Switch>

<Route path="/" component={LoginPage} exact={true}/>

<PrivateRoute path="/dashboard" component={ExpenseDashboardPage} />

<PrivateRoute path="/create" component={AddExpensePage}/>

<PrivateRoute path="/edit/:id" component={EditExpensePage}/>

<Route component={NotFoundPage}/>

</Switch>

</div>

</Router>

);

export default AppRouter;

Теперь заделаем **routers/PrivateRouter.js**. <Header/> можно удалить из AppRouter и переместить в PrivateRouter, что бы показывать только если юзер залогинен:

import React from 'react';

import {connect} from 'react-redux';

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

import Header from '../components/Header';

export const PrivateRoute = ({isAuthenticated, component: Component, ...rest}) => (

<Route {...rest} component={(props) => (

isAuthenticated ? (

<div>

<Header />

<Component {...props}/>

</div>

) : (

<Redirect to="/"/>

)

)}

/>

);

const mapStateToProps = (state) => ({

isAuthenticated: !!state.auth.uid

});

export default connect(mapStateToProps)(PrivateRoute);