<https://www.youtube.com/watch?v=ivDjWYcKDZI&t=25s>

Оглавление

[Инициализация npm 1](#_Toc42805660)

[На следующем шаге нужно зарегестрировать роуты, которые будут обрабатывать апи запросы с фронтенд. 6](#_Toc42805661)

[Далее переходим на работу с фронт-ендом. Пишем команду и разворачиваем реакт-апп в папке Клиент 11](#_Toc42805662)

[Запуск фронтенда и бэкенда одновременно 11](#_Toc42805663)

[Библиотека materialize 12](#_Toc42805664)

[Для переключения страниц подключить реакт-роутер-дом в папке реакта (Клиент) 12](#_Toc42805665)

[Готовим шаблоны страниц с материалайз 14](#_Toc42805666)

[Как отправлять определенные запросы на сервер для регистрации и авторизации 15](#_Toc42805667)

[Будем проксировать запросы с клиента на сервер. 19](#_Toc42805668)

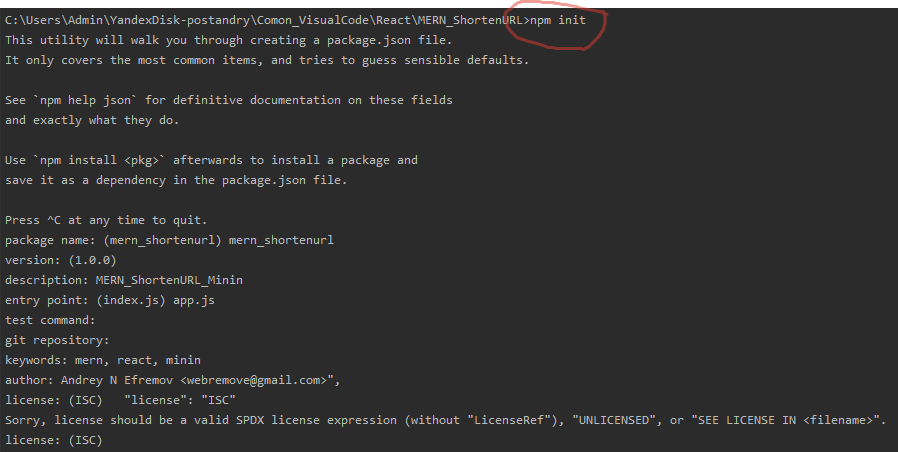
[Обработка ошибок на сервере в аус.роутес.жс при регистрации 21](#_Toc42805669)

[Логинизация пользователя 23](#_Toc42805670)

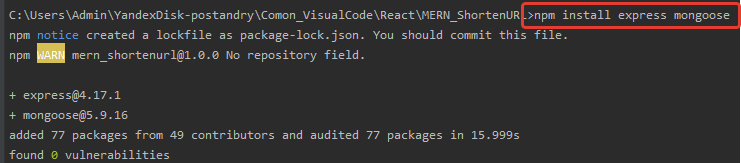
[Система авторизации на фронтэнде 24](#_Toc42805671)

[Добавляем Navbar в залогиненную страницу 27](#_Toc42805672)

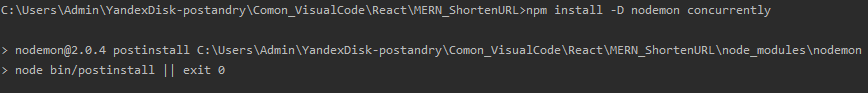
# Инициализация npm



Установка пакетов для соединения с базой и работой с МонгоДиБи



Еще два пакета для разработки (-D чтобы попало в девдепенденсис) нодемон (чтобы работал сервер) и конкаррентли, чтобы одновременно запускать фронтенд и бэкенд.

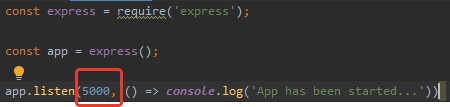


В пэккедж джейсон стартовую страницу меняем на апп, и добавляем сервер, который также стартет с апп.жс.

Нодемон будет запускать бэкенд как скрипт сервер. Нодемон позволяет перезагружать сервер, чтобы не делать этого вручную.

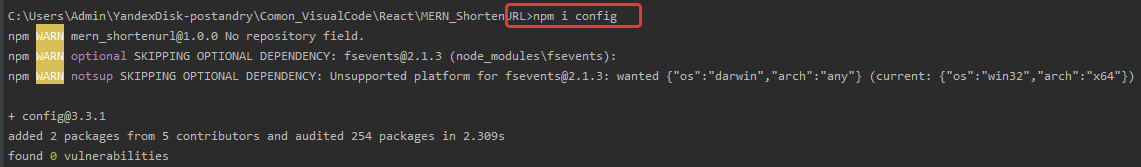


Чтобы не запускать сервер на захардкоденном порту 5000, нужно это вынести в общий конфиг.



Чтобы работать с конфигом ставим конфиг нпм, пакет позволяет взаимодействовать с различными конфигурационными константами.

<https://www.npmjs.com/package/config>



Далее создаем в папке проекта папку Конфиг и в ней файл дефолт.жсон. В нем пока только порт

{  
 "port": 5000  
}

Далее в апп.жс подключаем конфиг и берем методом гет порт и если его нет то хардкодом ставим 5000.

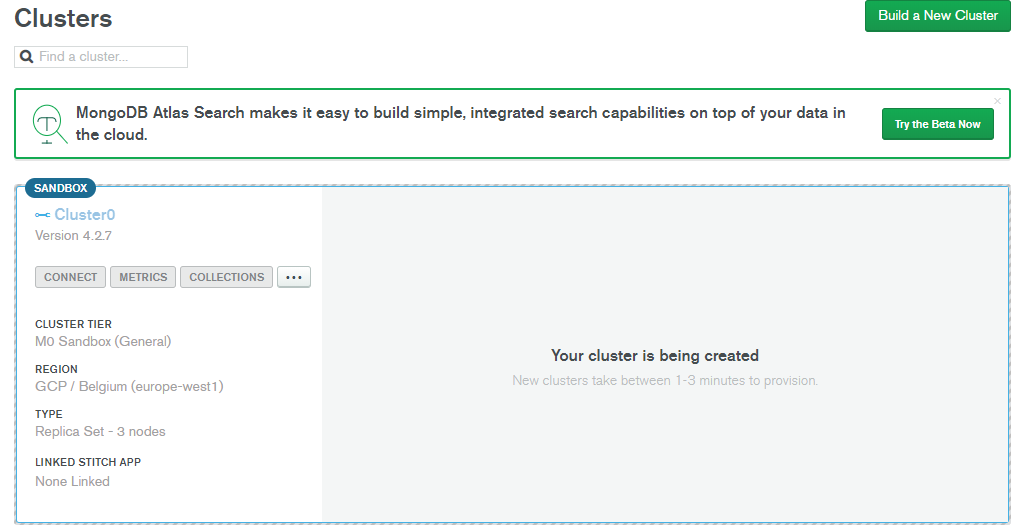


Для подключения к МонгоДБ подключаем пакет монгус. И вызвать метод коннект, чтобы подключиться к базе данных. Делаем через асинк эуэйт обертку. В конфиг добавляем пустое значение монгоУри, которое передаем в коннект первым параметром. Это тот адрес, по которому будем добавлять базу данных.

Вторым параметром идет набор опций (пока пустые). Сервер запускаем после эуэйта.

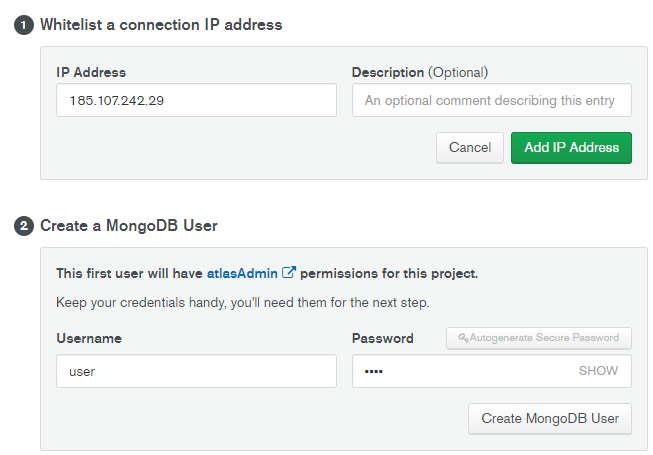
async function start() {  
 try {  
 await mongoose.connect(config.get('mongoUri'), {  
//пока пустые опции  
 })  
 app.listen(PORT, () => ***console***.log(`App has been started on port ${PORT}...`));  
 } catch (e) {  
 ***console***.log('Server Error', e.message)  
 process.exit(1)  
 }  
}  
  
start()

Создаём базу данных на ресурсе МонгоДБ. Сначала новый проект, потом Кластер (провайдеры и сервера можно выбрать любые, лучше ближние, остальное по умолчанию).

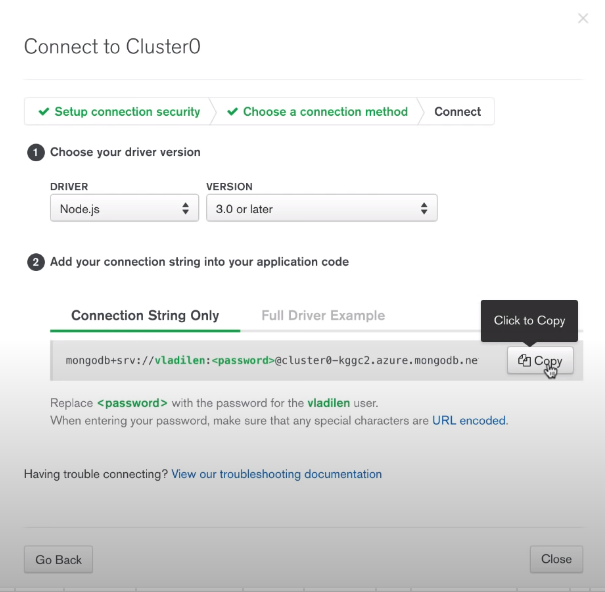


Настроим МонгоКоннект, добавим параметры, чтобы коннект успешно работал.

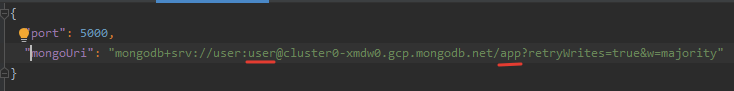
try {  
 await mongoose.connect(config.get('mongoUri'), {  
 useNewUrlParser: true,  
 useUnifiedTopology: true,  
 useCreateIndex: true  
 })

Далее на сайте МонгоДБ в кластере нажимаем коннект, добавляем свой айпи в список доступа, юзера и пароль для доступа. 

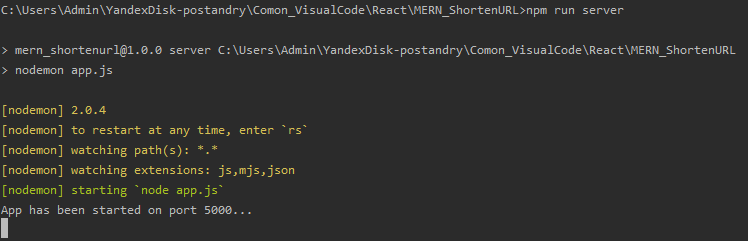
Потом опцию коннект йо аппликейшн. Копируем строку коннекта к базе. И вставляет в папке Конфиг в файл дефолт.жсон в строку монгоУри.



Вместо слова пассворд вписать пароль после двоеточия имени пользователя (у меня юзер логин и юзер пароль). И вместо названия тест написать апп после слэша адреса. Обратить внимание, что в двойных кавычках, на одинарные система ругается и сервак не стартует.



На этом настройка завершена и при запуске сервера в терминале он стартует без ошибок



На следующем шаге нужно зарегестрировать роуты, которые будут обрабатывать апи запросы с фронтенд.

Используем команду апп.юз. Когда регаем роут, то добавляем строчку, которая будет префиксом для будущего пути. Первым сделаем модуль авторизации. Сначала пишем строку путь, потом вторым параметром роут.

const app = express();  
  
app.use('/api/auth', require('./Routes/auth.routes'))  
  
const PORT = config.get('port') || 5000;

Но чтобы его создать нужно создать папку роутес в проекте. А в папке файл аус.роутес.жс.

В файл подключаем из экспресса Роутер. Создаем его и экспортируем. Роутер является по сути миддлвэйр.

const {Router} = require('express');  
const router = Router();  
  
module.exports = router

В аус.роутер.жс добавляем пост. Запросы (формируем эндпойнты для дальнейшей работы).

// /api/auth/register  
router.post('/register', async (req, res) => {  
   
})  
  
// /api/auth/login  
router.post('/login', async (req, res) => {  
  
})

Асинхронные запросы в аус.роутере обрабатываем с помощью трай-кэтч (чтобы перехватить ошибки в асинхронном запросе – это гуд практис). Пока только одну общую ошибку делаем – 500 (серверная ошибка) и выдаем соощение (блок кэтч).

router.post('/register', async (req, res) => {  
 try {  
   
 } catch (e) {  
 res.status(500).json({message: 'Something went wrong, try again'})  
 }  
});

В блоке трай получаем в параметре реквест имэйл и пароль.

router.post('/register', async (req, res) => {  
 try {  
 const {email, password} = req.body  
 } catch (e) {  
 res.status(500).json({message: 'Something went wrong, try again'})  
 }  
});

Дальше нужна логика регистрации нового пользователя и добавление проверок. Нужно создать сущность работы с пользователями. Создаем папку Модель и в нем файл Юзер.жс, модели обычно называют с большой буквы. В нем из монгуса забираем поля Схема и функция Модель (сначала пишем конст требует монгус, а потом в фигурные скобки вэбшторм подтягивает автоматически при начале ввода эту схему и модель). В схеме имэйл (с параметром уникальный), пароль. Экспортируем эту схему как модель пользователя.

const {Schema, model} = require('mongoose')  
  
const schema = new Schema({  
 email: {type: ***String***, required: true, unique: true},  
 password: {type: ***String***, required: true}  
})  
  
***module***.exports = model('User', schema);

Добавляем Линкс в модель и Тайпс из монгуса, чтобы у каждого пользователя был свой список урлов. Добавляем отсылку реф: Линк,, к какой коллекции мы привязываемся, но пока не задействована, нужно будет создать позднее.

const {Schema, model, Types} = require('mongoose');  
  
const schema = new Schema({  
 email: {type: ***String***, required: true, unique: true},  
 password: {type: ***String***, required: true}  
 links: [{type: Types.ObjectId, ref: 'Link'}]  
});  
  
***module***.exports = model('User', schema);

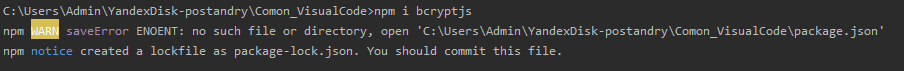
Модель подключаем в аус.роутес.жс.

const {Router} = require('express');  
const User = require('../Models/User');  
const router = Router();

Дальше делаем здесь же в аус.роутес.жс логику регистрации. Проверяем есть ли пользователь в базе. Если есть информация в кандидате – отправляем сообщение что пользователь существует. Чтобы скрипт не шел дальше пишем перед ним return (на этом остановится и даст ошибку, не будет выполняться дальше).

const candidate = await User.findOne({email: email}); //делаем проверку есть ли уже пользователь перед регистрацией, ждем инфы от Юзер есть ли имэйл  
if (candidate) {  
 return res.status(400).json({message: 'The user is existed'})  
}

Хэшируем пароль для безопасности. С помощью библиотеки bcrytp.js



Добавляем в аус.роутерс.жс

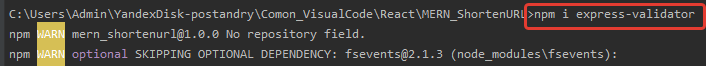
const bcrypt = require('bcryptjs')

Позволяет этот модуль хэшировать и сравнивать пароли. Асинхронный код, поэтому с эуэйт. И создаем пользователя.

const hashedPassword = await bcrypt.hash(password, 12); //хэшируем пароль, 12 - сила хэша как я понял  
const user = new User({email, password: hashedPassword}); //новый пользователь с хэшироанным паролем  
await user.save(); //сохраняем пользователя  
res.status(201).json({message: 'The user is created'}) //при статусе 201 (создан) выводим сообщение

Это весь процесс регистрации.

Но нужно еще сделать валидацию данных на сервере (некорретный имэйл, пустой пароль и др.). Для этого ставим пакет экспресса валидатор.

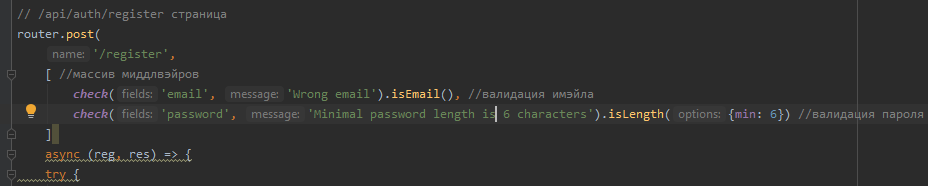


Подключаем и экспортируем методы чек и валидэйшнРезалт.

const {***check***, ***validationResult***} = require('express-validator') //подключение валидатора

В экспрессе мы можем добавлять неограниченно миддлвэйров. Поэтому в роутер пост между параметрами регистр и асинхронной функцией добавляем массив миддлвэйров, которые будут делать валидацию.

Вызываем Чек, первым проверяем имэйл, добавляем ошибку. Метод изМэйл добавляем. Через запятую другие валидаторы – пароль. С другим сообщением ошибки.



Далее эту валидацию вставляем перед логикой создания в блок трай.

a**sync (req, res) => {  
try {  
 const errors = *validationResult*(req); //проверка результата валидации  
 if (!errors.isEmpty()) {  
 return res.status(400).json({ //если в валидации есть ошибки - возвращаем на фронтэнд  
 errors: errors.array(),  
 message: 'Invalid registration data'  
 })  
 }**

Метод регистрации готов, переходим к методу Логин. Копируем весь блок Трай из Регистра. Так же получаем имэйл и пароль, проверяем валидацию. Оставляем переват ошибок, а логику удаляем.

router.post(  
 '/login',  
 [   
   
 ],  
 async (req, res) => {  
 try {  
 const errors = ***validationResult***(req); //проверка результата валидации  
 if (!errors.isEmpty()) {  
 return res.status(400).json({ //если в валидации есть ошибки - возвращаем на фронтэнд  
 errors: errors.array(),  
 message: 'Invalid login data'  
 })  
 }  
  
 } catch (e) {  
 res.status(500).json({message: 'Something went wrong, try again'})  
 }  
 });

Добавляем массив миддлвэйров-валидаторов.

[  
 ***check***('email', 'Enter valid email').normalizeEmail().isEmail(), //приведение и проверка имэйла  
 ***check***('password', 'Enter password').exists() //проверка на введение пароля  
],

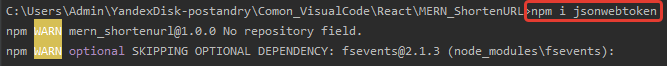
Далее логика по логированию пользователя. Сначала ищем есть ли зареганный и если нет даем ошибку

const user = await User.findOne({email}) // ищем пользователя в базе по имэйлу  
if (!user) {  
 return res.status(400).json({message: 'User is not found'}) //если пользователя нет то сообщаем  
}

Проверка на соответствие введеного пароля

const isMatch = await bcrypt.compare(password, user.password); //проверка на соответствие введенного пароля  
if (!isMatch) {  
 return res.status(400).json({message: 'Login or password is incorrect'})  
}

Если эти проверки прошли, то нужно авторизовать пользователя. Авторизацию в СПА приложении делаем через дживити-токен. Чтобы генерить этот токен, нужна библиотека джейсон-вэб-токен. Ставим



Подключаем в переменную jwt.

const jwt = require('jsonwebtoken'); //авторизация юзера чезе токен

Создаем в Логине токен. Для секретного ключа создаем в дефолт.жсон еще запись, куда передаем любую строку (секретную).

{  
 "port": 5000,  
 "jwtSecret": "shortenURL secret",  
 "mongoUri": "mongodb+srv://user:user@cluster0-xmdw0.gcp.mongodb.net/app?retryWrites=true&w=majority"  
}

Подключаем этот секрет в аус.роутер.жс

const config = require('config'); //подключение дефолт.жсок в конфиге с секретной фразой

Настраиваем токен в аус.роутес.жс

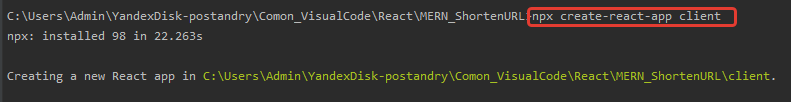
const token = jwt.sign(  
 {userId: user.id}, //данные, которые будут зашифрованы в этом токене. можно добавить мэйл, логин и другое  
 config.get('jwtSecret'), //секретную фразу передаём из настроек  
 {expiresIn: '1h'} //время жизни токена  
)

Далее пишем успешный ответ, если все этапы пройдены. Код получается ответа 200, мы его не пишем по умолчанию.

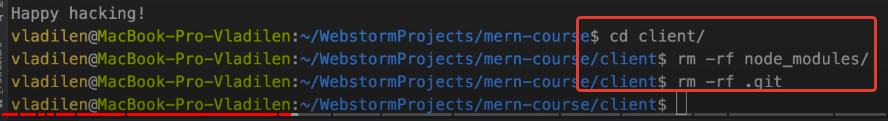
res.json({token, userId: user.id}) //ответ 200, успешная авторизация

На этом завершили с роутами, которые отвечают за регистрацию/авторизацию.

# Далее переходим на работу с фронт-ендом. Пишем команду и разворачиваем реакт-апп в папке Клиент



Удаляем из папки Клиент реакта нод\_модулес и .гит. Мы их используем в корне приложения.



У меня команды не прошли, удалил руками.

Из папки реакта эсэрси удаляем ненужные файлы: апп.цсс, апп.тест, лого.свг. Полностью очищаем индекс.цсс и удаляем все лишнее в апп.жс.

import React from 'react';  
  
function App() {  
 return (  
 <div>  
 <h1>ShortenURL App</h1>  
 </div>  
 );  
}  
  
export default App;

В терминале в папку Клиент переходим (папка реакта) и запускаем установку библиотек



# Запуск фронтенда и бэкенда одновременно

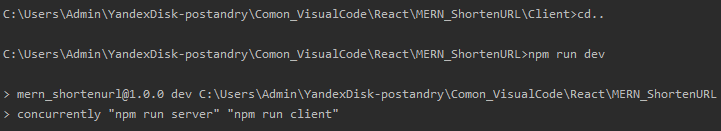
В пакэдж.жсон добавляем скрипт, запускающий реакт из папки клиент. Ранее только сервер запускался.

"server": "nodemon app.js", //запуск сервера в девелопмент режиме  
"client": "npm run start --prefix client" //запуск фронтенда реакта из папки клиент

Далее дописываем запуск сразу двух клиентов (чтобы не пускать один, потом второй)

"dev": "concurrently \"npm run server\" \"npm run client\"" //запуск сразу фронта и бэка одним скриптом

Чтобы запустить сайт нужно в папке проекта запустить npm run dev



# Библиотека materialize

Для упрощения работы с фронтом ставим библиотеку материалайз. Поскольку это фронт библиотека, то ставим в папку Клиент.



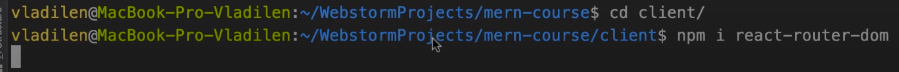
Импортируем в апп.жс материалайз цсс

import React from 'react';  
import 'materialize-css';

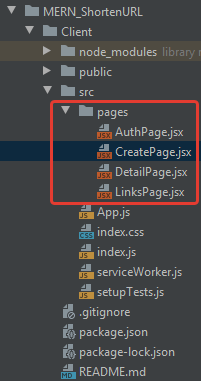
в индекс.цсс импортируем

@import "~materialize-css/dist/css/materialize.min.css";

# Для переключения страниц подключить реакт-роутер-дом в папке реакта (Клиент)



Создаем папку пэйджес в папке реакта (клиент и внутри src папки) и в ней страницы приложения (реакт-компоненты по сути).



В папке src реакта создаем файл роутес.жс (для перелинковок)

import React from "react";  
import {Redirect, ***Route***, ***Switch***} from "react-router-dom";  
import {LinksPage} from "./pages/LinksPage";  
import {CreatePage} from "./pages/CreatePage";  
import {DetailPage} from "./pages/DetailPage";  
import {AuthPage} from "./pages/AuthPage";  
  
export const useRoutes = isLogined => {  
 if (isLogined) { //логика если пользователь залогинен  
 return (  
 <Switch>  
 <Route path='/links' exact>  
 <LinksPage />  
 </Route>  
 <Route path='/create' exact>  
 <CreatePage />  
 </Route>  
 <Route path='/detail/:id'>  
 <DetailPage />  
 </Route>  
 <Redirect to='/create' />  
 </Switch>  
 )  
 }  
 return ( //логика если пользователь не залогинен  
 <Switch>  
 <Route path='/' exact>  
 <AuthPage />  
 </Route>  
 <Redirect to='/' />  
 </Switch>  
 )  
}

В апп.жс импортируем эти роуты из файла роутес.жс. Флаг логинизации по умолчанию фолс. И роуты в контент пока вместо Н1.

import {useRoutes} from "./routes";  
  
function App() {  
 const routes = useRoutes(false)  
 return (  
 <div className='container'>  
 <h1>{routes}</h1>

Но так работать не будет, будет ошибка You should not use <Switch> outside a <Router>. Поэтому в апп.жс нужно импортнуть БраузерРоутер как Роутер и обернуть константу роутес в этот Роутер.

import {***BrowserRouter*** as Router} from 'react-router-dom';

return (  
 <Router>  
 <div className='container'>  
 <h1>{routes}</h1>  
 </div>  
 </Router>  
);

# Готовим шаблоны страниц с материалайз

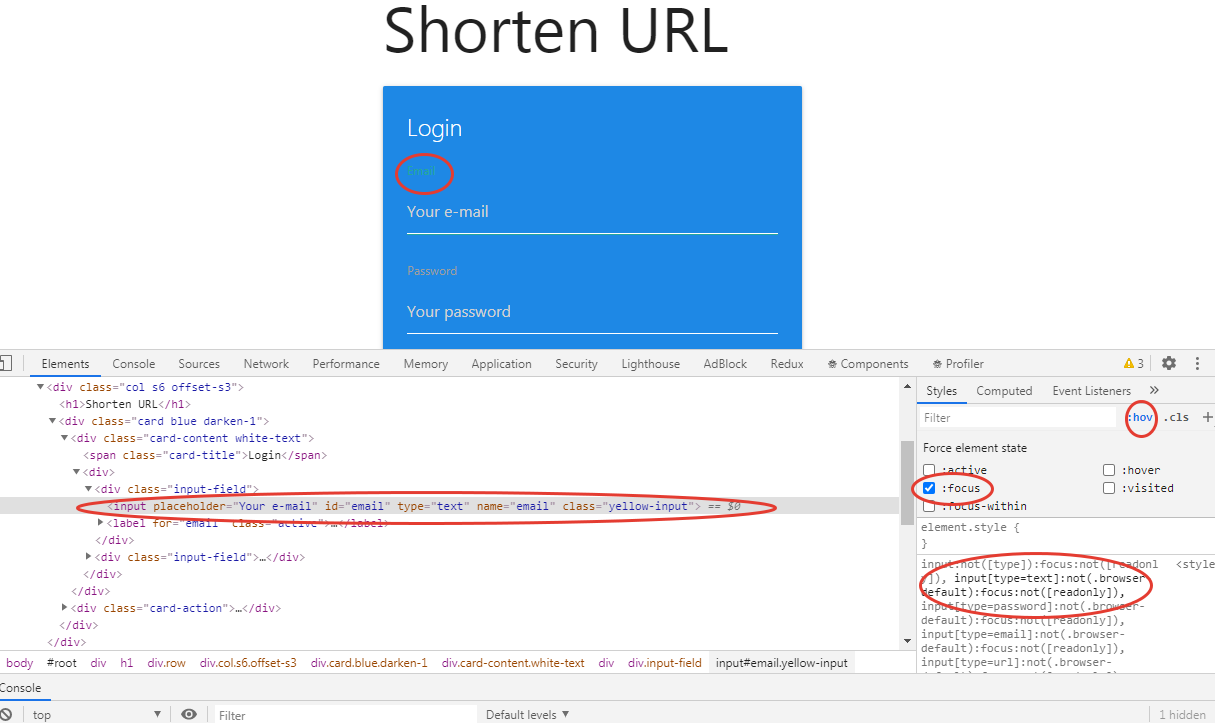
.col.s6.offset-s3

Такая запись при ТАБ дает имя классу.

Делаем компоненту и она же страницу АусПэйдж.жсх. Из материалайз копируем блок Кард. Потом копируем блок Текст Инпут. И вставляем в страницу (не забыть тег инпут закрыть слэшем).

import React from 'react';  
  
export const AuthPage = () => {  
 return (  
 <div className='row'>  
 <div className="col s6 offset-s3">  
 <h1>Shorten URL</h1>  
 <div className="card blue darken-1">  
 <div className="card-content white-text">  
 <span className="card-title">Login</span>  
 <div>  
 <div className="input-field">  
 <input  
 placeholder="Your e-mail"  
 id="email"  
 type="text"  
 name="email"  
 className='yellow-input'  
 />  
 <label htmlFor="email">Email</label>  
 </div>  
 <div className="input-field">  
 <input  
 placeholder="Your password"  
 id="password"  
 type="password"  
 name="password"  
 className='yellow-input'  
 />  
 <label htmlFor="password">Password</label>  
 </div>  
 </div>  
 </div>  
 <div className="card-action">  
 {/\* стайл делать только в css, это учебная демонстрация \*/}  
 <button className='btn yellow darken-4' style={{marginRight: 20}}>Sign in</button>  
 <button className='btn grey lighten-1 black-text'>Sign up</button>  
 </div>  
 </div>  
 </div>  
 </div>  
 )  
};

Чтобы изменить цвет при наведении и клике на поле ввода с зеленого на желтый, создадим в индекс.цсс класс. Увидеть действующий класс непросто в матералайз. Нужно в инспекторе через ховер аттрибут фокус.



Здесь сложный селектор и можно просто накатить новый сверху через создание в индекс.цсс и дать ему аттрибут !импортант. И он будет приоритетным (наш класс).

Добавляем в инпуты класс

className='yellow-input'

Далее в индекс.цсс прописываем

input.yellow-input {  
 border-bottom: 1px solid #fff**!important**;  
 box-shadow: 0 1px 0 0 #fff;  
}  
  
input.yellow-input + label {  
 color: #ffffff**!important**;  
}  
  
input.yellow-input:focus + label {  
 color: #ffeb3b**!important**;  
}  
  
input.yellow-input:focus {  
 border-bottom: 1px solid #ffeb3b**!important**;  
 box-shadow: 0 1px 0 0 #ffeb3b**!important**;  
}

# Как отправлять определенные запросы на сервер для регистрации и авторизации

В Ауспэйдж создаем хук юзСтейт, со стейтом форм и методом сетФорм. В стейте будет имейл и пассворд (пустые).

export const AuthPage = () => {  
 const [form, setForm] = useState({  
 email: '',  
 password: ''  
 });

Далее создаем метод чейнджХендлер (также до ретурна). Метод принимает ивэнт и меняет поле в форме (вызывая метод сетФорм). Разворачиваем оператором спред всю форму и обращаемся к дин.ключу ивент.таргет.нэйм и задаем ему ивент.таргет.вэлью. ЮзСтейт не забывать импортнуть из реакта.

const changeHandler = event => {  
 setForm({...form, [event.target.name]: event.target.value})  
};

В каждом инпуте у нас есть этот нэйм (имэйл и пассворд). Поэтому дописываем в инпуты онЧейндж и передаем туда метод чейндХендлер. Таким образом будет обновляться форма.

className='yellow-input'  
onChange={changeHandler}

Дальше делаем запрос на сервер. Это можно прописать внутри аусПэйдж, либо подключить аксиос. Но лучше использовать свои созданные хуки. Для этого в корне соурс реакта создаем папку Хукс и файл хттп.хук.жс. И здесь экспортируем хук ЮзХТТП, он позволит работать с асинхронными запросами на сервер, использую нативный АПИ браузера фетч, но уже в формате хуков.

export const useHttp = () => {  
   
};

Этот хук будет позволять работать с сервером и экспортировать сущности, которые мы сгруппируем в данном модуле. Эти сущности – функция, позволяющая сделать запрос. СтейтЛоадинг – процесс загрузки и потенциальные ошибки, если они есть. Импортируем юзСтейт и задаем хуку лоадинг фолс.

export const useHttp = () => {  
 const[loading, setLoading] = useState(false)  
 const request = () => {  
  
 }  
};

После этого возвращаем объект с флагом лоадинг и функцией реквест.

const request = () => {  
 }  
 return {loading, request}  
}

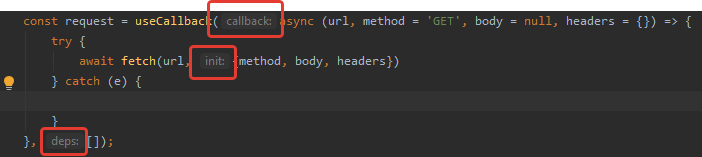
Также добавляем стэйт еррор с фнукцией сетЕррор и умолчание значение фолс.

const[loading, setLoading] = useState(false);  
const[error, setError] = useState(null);

Когда делаем реквест – функция принимает Урл, метод Гет, бади нулл и хедеры пустой объект. Функция асинхронная и добавляем асинк. Чтобы корректно сформировать её нужно обернуть в хук ЮзКоллбек, чтобы реакт не входил в рекурсию.

const[error, setError] = useState(null);  
  
const request = useCallback(async (url, method = 'GET', body=null, headers={}) => {  
});

Внутрь реквеста добавляем блок трай/кэтч, т.к. работаем с асинк-эвэйт. И нужно внутри обратиться к методу фетч. В него передаем урл и набор опций (метод, бади, хедерс – мы их так называем и они совпадают со стандартным АПИ). И в набор зависимостей добавляем пока пустой массив (это второй параметр хука юзКоллбек).



При копировании текста не видно подсказок, потому решил скрин сделать.

Когда мы дождались когда делается ответ на сервер мы дождались ответ Респонс и распарсиваем его, приводят к формату Джсон.

try {  
 const response = await fetch(url, {method, body, headers});  
 const data = await response.json();

Делаем проверку, если респонс с полем ок ошибочен, то выкидываем ошибку, с сообщением ошибки. Выведем дата.месседж, если он есть в дате, если нет – то дефолтный текст наш захардкоденный. Если проверка прошла нормально, тогда возвращаем дату, которая прилетела с сервера.

const data = await response.json();  
   
 if (!response.ok) {  
 throw new ***Error***(data.message || 'Something went wrong')  
 }  
   
 return data  
} catch (e) {

Теперь нужно правильно отобразить лоадинг. Сразу когда запрос, даже вне блока трай, отправляем лоадинг с параметром тру. Если была ошибка, то после выброса ошибки вставляем сетЛоадинг с параметром фолс (загрузки больше нет)

const request = useCallback(async (url, method = 'GET', body = null, headers = {}) => {  
 setLoading(true)  
 try {

if (!response.ok) {  
 throw new ***Error***(data.message || 'Something went wrong')  
}  
  
setLoading(false)  
  
return data

В блок кэтч мы попадаем когда есть ошибка, сетЛоадинг ставим в значение фолс. СетЕррос вызываем с методом е.мессадж. И выкидываем эту ошибку.

} catch (e) {  
setLoading(false)  
 setError(e.message)  
 throw e  
 }  
 }, []);

Можем еще экспортировать функцию, которая чистит ошибки. Она будет менять стейт локальный со значением нулл. И экспортим её (в ретурн добавляем)

}, []);  
  
 const clearError = () => setError(null)  
   
 return {loading, request, error, clearError}  
};

Далее используем этот хук. Вставляем его на странице аусПэйдж и импортируем. И достаем из него объект, в которые получаем определенные поля – импортируем лоадинг, еррор, функция реквест.

export const AuthPage = () => {  
 const {loading, error, request} = useHttp();

Для осуществления запроса на сервер нам нужно два метода: для регистрации и логина. Функция регистрации объявляем, асинхронную. Сразу пихаем трай/кэтч, так как могут быть ошибки. Но кэтч пропускаем, мы его обработали уже в хуке хттп. Здесь нам нужно получить дату с сервера и ожидаем выполнения функции реквест с параметрами урл аус/апи/регистр (мы его осуществили в роуте на бэкенде уже ранее), метод Пост и дату для передачи на сервер (имейл и пароль – для этого можем развернуть форму).

const registerHandler = async () => {  
 try {  
 const data = await request('/api/auth/register', 'POST', {...form});  
 } catch (e) {  
  
 }  
}

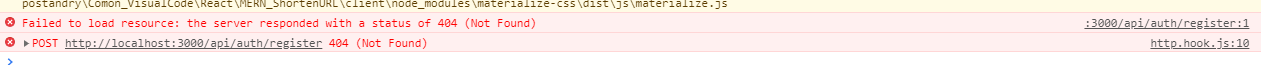
Этот метод привязываем к кнопке регистрации. И если работает режим лоадинг (загружаются данные), то нам нужно задизэйблить кнопку, то есть дзейбл должен быть тру, если лоадинг. Делаем это для обеих кнопок

<button className='btn grey lighten-1 black-text'  
 onClick={registerHandler}  
 disabled={loading}>  
 Sign up  
</button>

Получаем дату в регистрХэндлер, смотрим что в консоли при попытке ввести данные

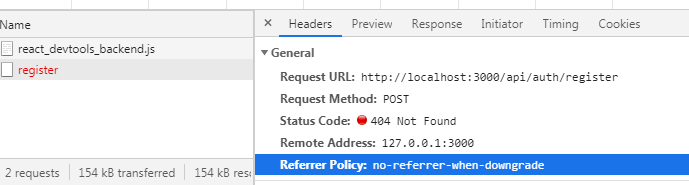
const data = await request('/api/auth/register', 'POST', {...form});  
***console***.log('data', data)

Тестируем. Вводим данные и нажимаем кнопку регистрации.



Валится ошибка, что не найден путь регистрации на локалохост:3000.

И это правильно, сервер-то стартовал у нас на порту 5000. Хедерсы мы отправляем правильные, но не туда.



Но и прописывать просто 5000 порт в пути некорректно. При переносе на хостинг там будет другой урл. И для данной ситуации используем прокси.

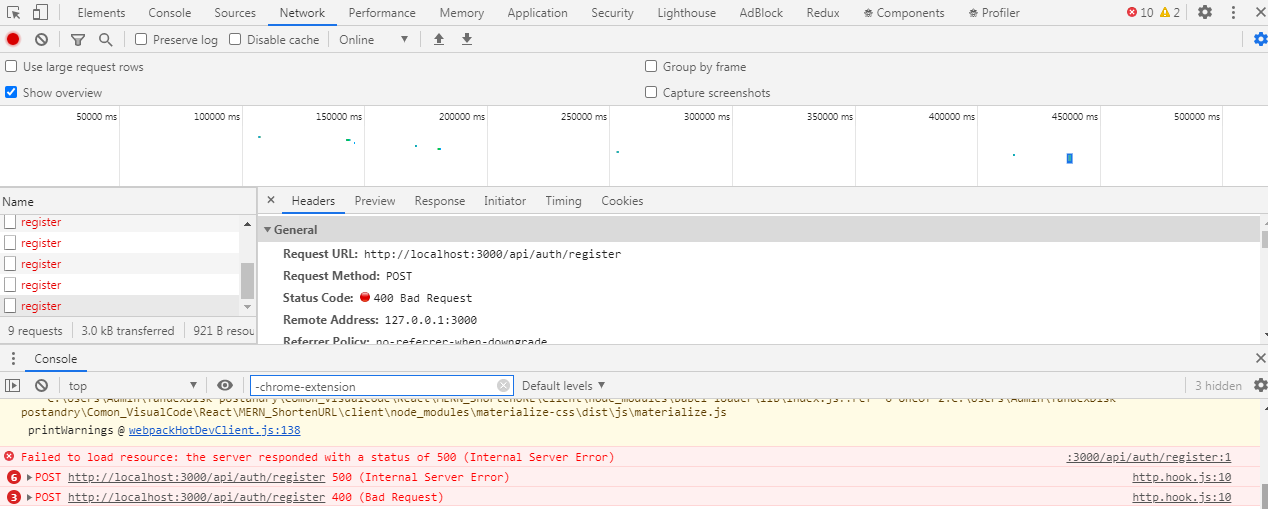
# Будем проксировать запросы с клиента на сервер.

Останавливаем сервер.

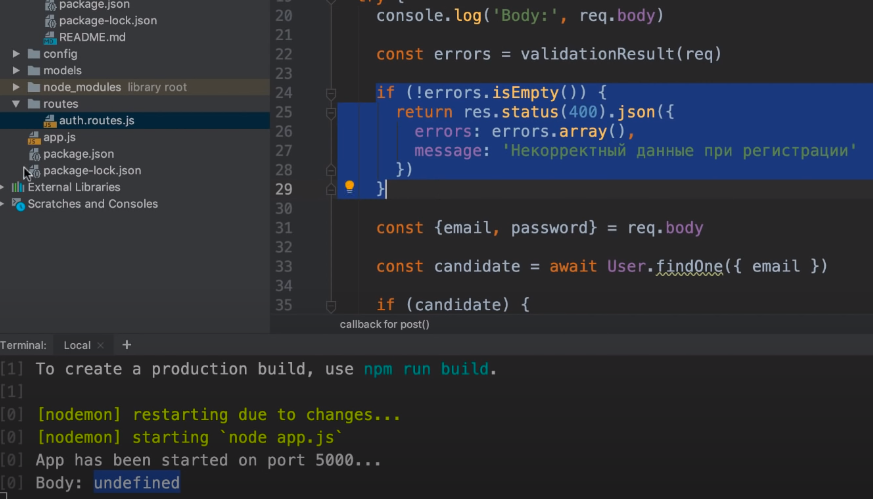
Для проксирования в папке реакта Клиент открыть пэккэдж.жсон. Добавить поле прокси с проксирование всех запросов на порт 5000 (на сервер). Это для стадии разработки. На продакшене по-другому будет приложение запускаться.

"eslintConfig": {  
 "extends": "react-app"  
},  
"proxy": "http://localhost:5000",  
"browserslist": {  
 "production": [

Запускаем опять сервер npm run dev. Падает 400 ошибка.



Далее пока разбираю ошибку автора. Приходят сообщения, что невалидный имеэйл и пароль. Это происходит из-за того, что в рек.бади в аус.роут.жс (на стороне сервера) не приходят данные. Если законсолить здесь этот бади, то он будет андефайнд (в консоль сервера падают данные здесь, это сервер, не браузер).



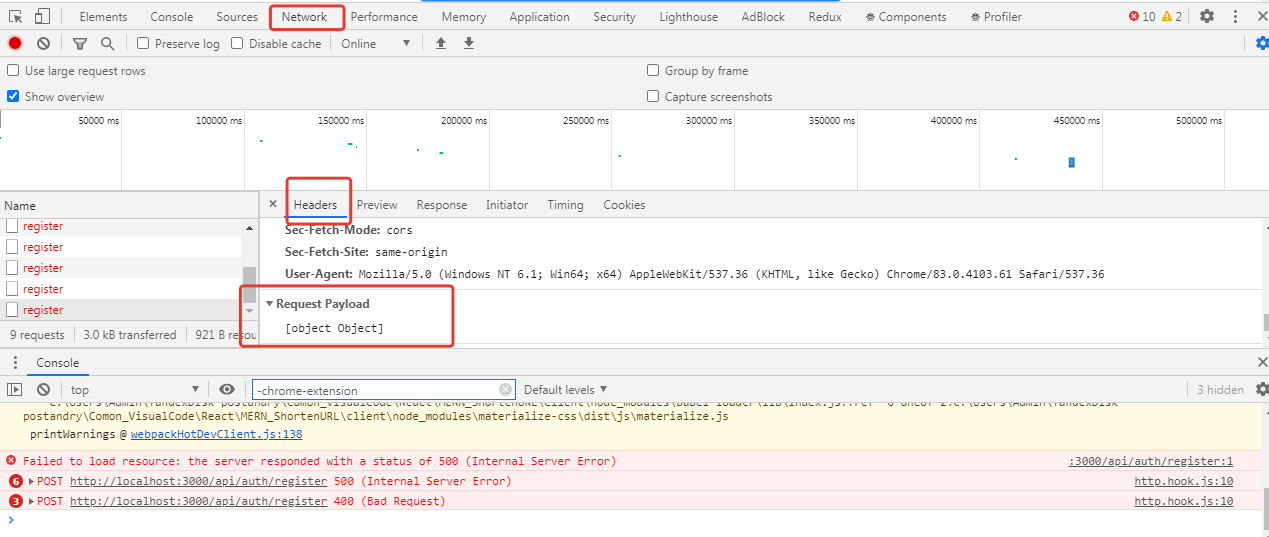
По умолчанию, нод жс воспринимает бади как стрим. Чтобы избежать ошибок, нужно добавить новый миддлвэйр для бади. Он будет парсить в жсон формат бади для сервера. Для этого идем в главный файл сревера апп.жс и добавляем миддлвэйр экспресса, называется жсон. И передаем один парамеьтр экстендед: тру.

const app = express();  
  
app.use(express.json({extended: true}));  
  
app.use('/api/auth', require('./Routes/auth.routes'));

После этого добавления миддлвэйр бади будет уже не андефайнд, а пустым объектом.



Но еще не работает. Если посмотреть в хедере в бразузере во вкралтке нетворк, то там реквест пэйлоад как [обжект Обжект].



А должен быть имейл и пассворд. Мы некорректно отправляем эти данные. И нужно доработать файл хттп.хук.жс. Сейчас файл не приводит бади к строке (бади = нулл) и превращает его в обжект обжект.

Сделаем проверку сразу после трай, что если мы передаем бади в реквест и он непустой, то его необходимо привести к строке. А если пустой (по умолчанию прописан выше), то бади равен нулл.

const request = useCallback(async (url, method = 'GET', body = null, headers = {}) => {  
 setLoading(true)  
 try {  
 if(body) {  
 body = ***JSON***.stringify(body)  
 }

Ошибка упала все-равно, но в реквест-пэйлоаде хедера данные передаем уже нужные: имейл и пассворд. Но на сервере получаем так же пустой объект. С ЖСОН нужно явно указать, что по сети передаем жсон. Поэтому в ту же проверку иф в хттп.хук.жс добавляем хедер поле.

const request = useCallback(async (url, method = 'GET', body = null, headers = {}) => {  
 setLoading(true)  
 try {  
 if (body) {  
 body = ***JSON***.stringify(body)  
 headers['Content-Type'] = 'application/json'  
 }

Все сработало, в консол.лог сервера приходит бади данные. Еще есть ошибка регистрации, потому что имэйл невалидный вводили (просто буквы). Но в реквест он уже уходит.

# Обработка ошибок на сервере в аус.роутес.жс при регистрации

Эту обработку делаем на фронтэнде, на реакте на странице аусПэйдж. Используем хук юзЭффект и следим за ошибкой. Получаем эррот в константе из юзХттр хука, и следим за эррор в зависимостях юзЭффекта.

const {loading, error, request} = useHttp();  
const [form, setForm] = useState({  
 email: '',  
 password: ''  
});  
  
useEffect(() => {  
  
}, [error])

Если есть ошибка (она обновилась), то нужно вывести её для пользователя. Массив эроррс обрабатывать не будем (где серверная конкретика что произошло, пароль или логин), а просто выведем общую ошибку, месседж, где нет конкретики (но можно обрабатывать и массив эррорс, чтобы явно показывать пользователю, что не так).

Чтобы показывать месседж, пользуемся материалайз с методом тоаст. Для этого еще один хук делам на клиенте, создаем файл мессадж.хук.жс.

Экспортируем константу юзМессадж, которая будет возвращать коллбэк (сразу юзКоллбэк хук импортируем из реакта, для того, чтобы реакт не входил в циклическую рекурсию). Возвращаем этот импортированный юзКоллбэк,

Первым параметром передаем функцию, второым – набор зависимостей (по умолчанию пустой объект).

В функцию принимаем параметр текст, и спрашиваем, если в объекте виндоу существует объект М (это встроенный объект), и мы передали текст, то тогда показывать сообщение виндоу.М.тоаст с возвращение в хтмл текст.

import {useCallback} from 'react';  
  
export const useMessage = () => {  
 return useCallback(text => {  
 if (***window***.M && text) {  
 ***window***.M.toast({html: text})  
 }  
 }, [])  
}

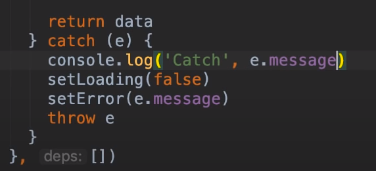
В аусПэйдж.жс импортируем хук юзМессадж, получаем в константе объект мессадж. Добавляем как зависимость для юзЭффекта.

import {useMessage} from "../hooks/message.hook";  
  
  
export const AuthPage = () => {  
 const message = useMessage()  
 const {loading, error, request} = useHttp();  
 const [form, setForm] = useState({  
 email: '',  
 password: ''  
 });  
  
 useEffect(() => {  
  
 }, [error, message])

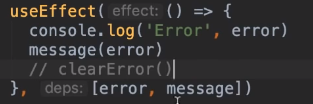
И если меняется объект эррор, просто показываем эту ошибку в функции мессадж (в теле юзЭффект прописываем это). И сразу после этого очищаем ошибку фнкцией клиарЕррор, вызвав ее тоже из юзХттп хука, и добавив ее как зависимость.

const {loading, error, request, clearError} = useHttp();  
const [form, setForm] = useState({  
 email: '',  
 password: ''  
});  
  
useEffect(() => {  
 message(error);  
 clearError()  
}, [error, message, clearError]);

Пока не работает, ищем баг. Консолим юзЭффект (сразу после объявления). Эррор нулл сейчас. Поэтому разбираем хттп.хук.жс. Консолим мессадж.



Здесь сообщение корректное, неправильно передает в юзЭффект это сообщение. Убрали в юзЭффект клиарЭррор – и в консоли корректно все обработано. Ошибка в клиарЭррор

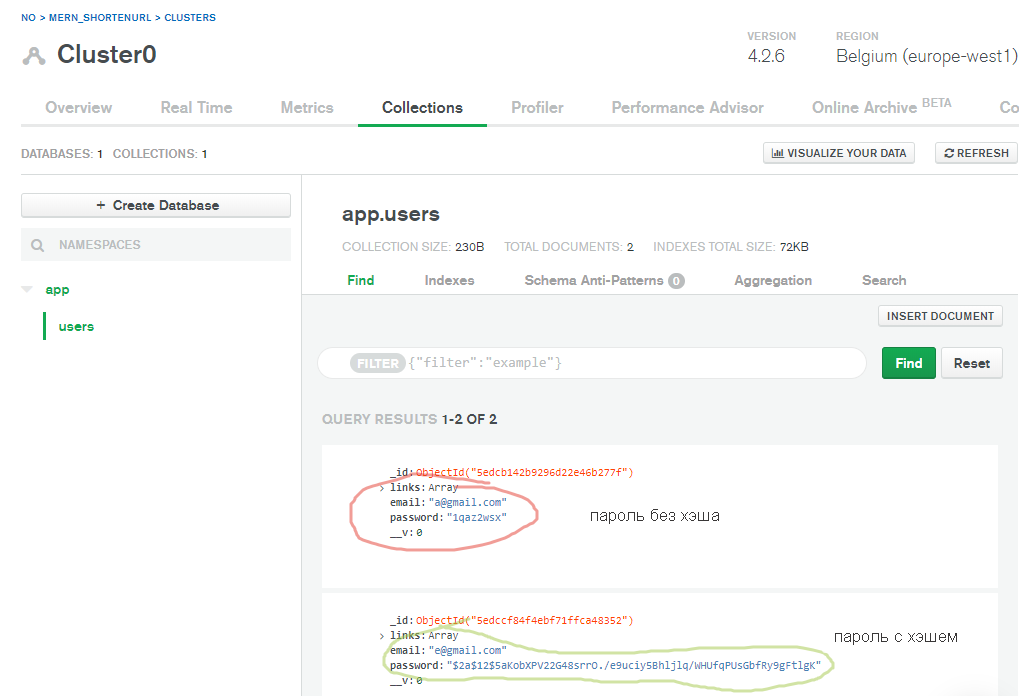
 так работает

Ошибка – нужно в хттп.хук.жс в объявлении клиарЭррор обернуть сетЭррор юзКоллбэком с пустыми зависимостями.

const clearError = useCallback(() => setError(null), []);

# Логинизация пользователя

Вводим любой (даже несуществующий) имэйл. Проверяем в монгоДБ пользователей, во вкладке Коллекшнс.



Массив Линкс пока пустой. При повторной регистрации имэйл появляется ошибка – пользователь существует. Все корректно пока отрабатывается.

Интересная заметка – первого юзера я сделал без функции хэшПассворд, а потом включил её. И зайти первым юзером я больше не могу в систему. Удаляю его.

Далее нужно сделать логин, чтобы зайти нужным пользователем.

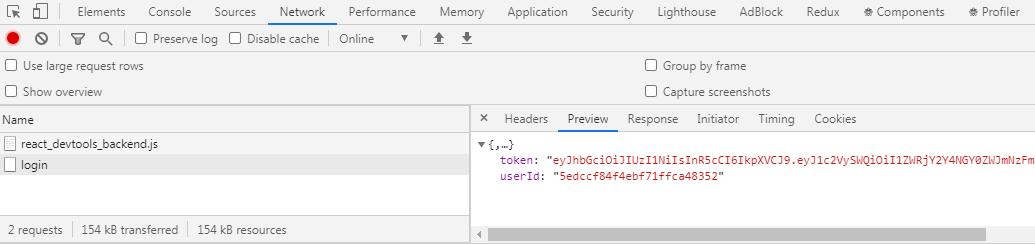
Называем логинХэндлер функцию (делаем ее в аусПэйдж) и делаем запрос на эндпоинт /логин. Тип запроса ПОСТ.

const loginHandler = async () => {  
 try {  
 const data = await request('/api/auth/login', 'POST', {...form});  
 message(data.message)  
 } catch (e) {  
  
 }  
};  
  
return (  
 <div className='row'>  
 <div className="col s6 offset-s3">

Прикручиваем этот метод к кнопке здесь же в аусПэйдж.

<button className='btn yellow darken-4'  
 style={{marginRight: 20}}  
 onClick={loginHandler}  
 disabled={loading}  
>  
 Sign in  
</button>

В случае, если хорошо с логином мы отдаем жсон с токеном и юзерайди. Это в аус.роутах прописано. Пробуем войти пользователем. Получаем логин со статусом 200 и токеном с юзерайди сгенерированным.

****

# Система авторизации на фронтэнде

Для этого будем использовать хук. Создаем аус.хук.жс в папке хуков, клиента Реакта (фронт). Это будет модуль, работает исключительно с авторизацией пользователя. Коллбэк функция, в которую будем импортировать вход или выход. С жвт токеном нам необходимо его хранить в локалсторедже (при перезагрузке страницы токен сохранится в локалсторедже и обратно в систему будет брошен, без повторной авторизации в текущей сессии).

Импортируем хуки из реакта: юзСтейт и юзКоллбэк. Создаем стейт, отвечающий за токен, по умолчанию нулл. Потребуется юзерайди, также нулл по умолчанию.

И два метода: логин в юзКоллбек и логаут так же в юзКоллбек. И два метода будем возвращать.

import {useState, useCallback} from 'react';  
  
export const useAuth = () => {  
 const [token, setToken] = useState(null);  
 const [userId, setUserId] = useState(null);  
  
 const login = useCallback(() => {  
   
 }, []);  
  
 const logout = useCallback(() => {  
  
 }, []);  
  
 return {login, logout}  
};

В методе логин задаем токен, принимаем в нее жвт токен с бэкенда плюс юзерайди. Задаем жвтТокен для локального стейта и вызвыаем сетЮзерАйди с передачей айди. И все это записываем в локалсторедж.

Отдельно создаем переменную стореджНэйм (чтобы не дублировать) и передаем эту строку в локалсторедж. Далее, чтобы корректно передать в локалсторедж объект, делаем жсон.стрингифай и в его объект записываем юзерайди и токен с сервера.

const storageName = 'userData';  
  
export const useAuth = () => {  
 const [token, setToken] = useState(null);  
 const [userId, setUserId] = useState(null);  
  
 const login = useCallback((jwtToken, id) => {  
 setToken(jwtToken)  
 setUserId(id)  
  
 ***localStorage***.setItem(storageName, ***JSON***,stringify({  
 userId, token  
 }))  
 }, []);

В методе логАут вызываем сетТокен нулл (чистим значение) и сетЮзерАйди также нулл. Плюс очищаем локалСторедж.

const logout = useCallback(() => {  
 setToken(null);  
 setUserId(null);  
 ***localStorage***.removeItem(storageName)  
}, []);

Ретурним кроме логина, логаута еще токен и юзерАйди.

return {login, logout, token, userId}

Еще один момент интересный: пока приложение загружается, нужно чтобы хук смотрел есть ли данные в локалсторедже. Если есть – чтобы из локалстореджа этот хук записал данные в локальное состояние. Используем юзЭффект хук. Импортируем его из реакта.

Вызываем после логаута, передаем в него коллбэк, передаем потенциальные данные из локалСторэдж. Приводим это к объекту через ЖСОН.парс. Далее если дата есть (не нулл) и в нем есть поле токен, то вызываем функцию логин с этими значениями. Куда передаем дату.токен и дату.юзерайди (они были записаные в локалсторэдж). И как зависимость для данного юзЭффекта нужно указать метод Логин (потому что он здесь используется). И именно для этого оборачивали логин в юзколлбек.

useEffect(() => {  
 const data = ***JSON***.parse(***localStorage***.getItem(storageName));  
  
 if (data && data.token) { //одновременно два условия: есть и дата, и в дате токен  
 login(data.token, data.userId)  
 }  
}, [login]);

Далее в файле Апп.жс (клиента, на фронте в срц папке) воспользуем хуков юзАус и получить все данные (токен, логин, логаут и юзерАйди). Импортируем этот хук.

import {useAuth} from "./hooks/auth.hook";  
import 'materialize-css';  
  
function App() {  
 const {token, login, logout, userId} = useAuth();  
 const routes = useRoutes(false);

ЮзАус вызываем без параметров. Далее эти значения нужно передать всему приложению через контекст. В папке срц Клиента создаем папку контекст. В ней компонент АусКонтекст.жсх. Контекст создается через функцию креэйтКонтекст. Создаем контектс, передаем базовое состояние контекста (токен = нулл, юзерайди = нулл, логин – пустая функция, логаут – пустая функция, флаг изАутентикэйтед = фолс).

import {createContext} from 'react';  
  
function noop () {}  
  
export const ***AuthContext*** = createContext({  
 token: null,  
 userId: null,  
 login: noop,  
 logout: noop,  
 isAuthenticated: false  
});

Чтобы передавать весь этот контекст по всему приложению, нужно обернуть приложение в Контекст. Это в файле Апп.жс Клиента. И в качестве значения вэлью передаем объект с теми значениями, которые получили из хука юзАус (логин, логаут, юзерАйди, токен и флаг юзАутентикэйдет – определяем флаг просто наличия токена, через булево значение два отрицания). И этот флаг мы передаем в юзРоутес, как динамическое значение.

function App() {  
 const {token, login, logout, userId} = useAuth();  
 const isAuthenticated = !!token;  
 const routes = useRoutes(isAuthenticated);  
 return (  
 <AuthContext.Provider value={{  
 token, login, logout, userId, isAuthenticated  
 }}>  
 <Router>  
 <div className='container'>  
 {routes}  
 </div>  
 </Router>  
 </AuthContext.Provider>  
 )  
}

Ошибки на аус.хук.жс посыпались, что не хватает зависимостей для токен и юзерАйди в юзКоллбек. Но мы можем не передавать эти зависимости, а передать значения в объекте локалСторэдж, потмоу что они одинаковые.

***localStorage***.setItem(storageName, ***JSON***.stringify({  
 userId: id, token: jwtToken  
}))

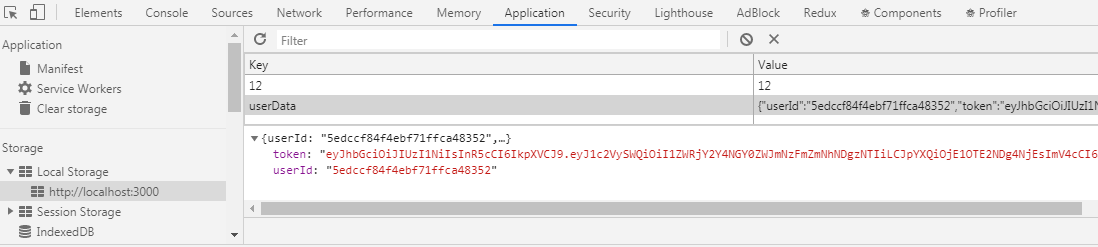
Далее воспользуемся контекстом на странице АусПэйдж с помощью хука юзКонтекст. Импортируется из реакта, константим как аус и передаем контекст, который создали. Теперь здесь в объекте созданном аус все данный, доступные провайдеру контекста.

import {***AuthContext***} from "../context/AuthContext";  
  
  
export const AuthPage = () => {  
 const auth = useContext(***AuthContext***);

И в логинХендлере вместто дата.мессадж (не нужно сообщение что авторизованы), передаем аус.логин, первым параметром их аус.хукс дата.токен и дата.юзерайди.

const loginHandler = async () => {  
 try {  
 const data = await request('/api/auth/login', 'POST', {...form});  
 auth.login (data.token, data.userId)

И теперь после входа нас перебрасывает на страницу КреэйтПэйдж (по умолчанию), логика полностью отработалась без необходимости доп.прописываний. И в аппликейшн есть юзер дата и токен теперь.



И если удалить в адресе страницу и перезагрузить – мы все равно попадаем на эту страницу. Локалсторадж работает.

# Добавляем Navbar в залогиненную страницу

D папке эсэрси клиента новую папку Компонентс создаем. И в ней компонент Навбар.жсх. Если залогинены в системе – показываем немного другой лэйаут. Импорт реакта и экспорт константы Навбар. В ретурн вставялем код из материалайз с меню навигационным. В коде материалайза только ссылки поправляем. Для этого импортируем Навлинк из реакт-роутер-дом. И вместо ссылок а-хреф будут ссылки НавЛинк (по страницам). Ссылка на кнопку на странице будет а-хреф.

import React from 'react';  
import {***NavLink***} from "react-router-dom";  
  
export const Navbar = () => {  
 return (  
 <nav>  
 <div className="nav-wrapper">  
 <a href="#" className="brand-logo">Logo</a>  
 <ul id="nav-mobile" className="right hide-on-med-and-down">  
 <li><NavLink to="/create">Create</NavLink></li>  
 <li><NavLink to="/links">Create</NavLink></li>  
 <li><a href="/" onClick={logoutHandler}>SignOut</a></li> {/\*href потому что это не ссылка на страницу, а на кнопку\*/}  
 </ul>  
 </div>  
 </nav>  
 )  
}

Создаем функцию логаутХэндлер, активирующую кнопку выхода. Для этого потребуется еще контекст (авторизации). В логаутХэндлер стопим дефолтное поведение, чтобы ссылка не обрабатывалась. И аус.логаут (метод из хука аус, обнуляет айди и локалсторэдж) вызываем.

export const Navbar = () => {  
 const auth = useContext(***AuthContext***);  
 const logoutHandler = event => {  
 event.preventDefault();  
 auth.logout()  
 };

Делаем еще редирект. Импортируем юзХистори. Получить объект хистори. И запушить с ним прееход на главную страницу в логаутХэндлер.

import {***NavLink***, useHistory} from "react-router-dom";

export const Navbar = () => {  
 const history = useHistory();  
 const auth = useContext(***AuthContext***);  
 const logoutHandler = event => {  
 event.preventDefault();  
 auth.logout();  
 history.push('/')

Вместо лого в разметке сделали переход на главную и имя изменили

<a href="/" className="brand-logo">Shorten URL</a>

Подкключаем компонент в Апп.жс клиента. В роутере перед дивом контейнер добавляем проверку. Если залогинены – будет показываться компонент Навбар (импортнуть его из папки компонент).

<Router>  
 {isAuthenticated && <Navbar/>} {/\*если авторизован показать навбар\*/}  
 <div className='container'>

Меняем в Навбаре компоненте цвет на синий (весь враппер стиль).

return (  
 <nav>  
 <div className="nav-wrapper blue darken-1">

В заголовке меняем ссылку на спан и добавляем врапперу еще паддинги (только для учебы, в реале через отдельный цсс модуль).

<div className="nav-wrapper blue darken-1" style={{padding: '0 2rem'}}>  
 <span className="brand-logo">Shorten URL</span>

Возникла проблема, что при логауте и переходе на главную страницу – инпуты формы неактивны. Для активизации на странице АусПэйдж добавляем еще один юзЭффект с пустыми зависимостями и обращаемся к виндоу.М и методу апдейтТекстФилд, который позволяет сделать активными инпуты после релогина.

useEffect( () => {  
 ***window***.M.updateTextFields()  
},[])

# Генерация сокращенных сылок

Возвращаемся в бэкенд и создаем новый роутер линк.роутес.жс. Этот роутер будет отвечать за генерацию ссылок сокращаемых.

Создаем новую модель Линк в папке моделс. Копируем содержание Юзера модели.

const {Schema, model, Types} = require('mongoose');  
  
const schema = new Schema({  
 from: {type: ***String***, required: true}, //от кого сформирована ссылка  
 to: {type: ***String***, required: true, unique: true}, //куда ведет ссылка, ссылка уникальная будет  
 code: {type: ***String***, required: true, unique: true}, //для взаимодействия с кодом  
 date: {type: ***Date***, default: ***Date***.now}, //дата создания ссылки, дефолт - текущее время. но не вызываем дэйт.нау, а передаем как референс  
 clicks: {type: ***Number***, default: 0}, //количество кликов по ссылке  
 ownner: {type: Types.ObjectId, ref: 'User'} //пользователь создавший ссылку, реф до коллекции (модели) Юзер  
});  
  
***module***.exports = model('Link', schema);

Далее организуем сам роутер. Подключаем экспресс (и роутер). Подключаем модель и создаем роутер. Экспортируем роутер. В данном роутере потребуется обработать несколько запросов. Первый пост заброс обрабатывает ссылку (эндпоинт) генереэйт, которая генерирует короткую ссылку. Описываем асинхронную функцию.

Далее гет запрос для получения всех ссылок.

Далее гет запрос для получения ссылок по айди.

const {Router} = require('express'); //роутер  
const Link = require('../Models/Link')  
const router = Router(); //константа роута  
  
// /api/auth/generate страница  
router.post(  
 '/generate', //первый параметр метода POST  
 async (req, res) => { //второй параметр метода POST  
  
 });  
  
router.get(  
 '/',  
 async (req, res) => {  
  
 });  
  
router.get(  
 '/:id',  
 async (req, res) => {  
  
 });  
  
***module***.exports = router;

Добавляем общую обработку роутеров (трай/кэтч из роута аус и сообщение об ошибке). Для всех эндпоинтов.

try {  
   
} catch (e) {  
 res.status(500).json({message: 'Login - Something went wrong, try again'})  
}

В апп.жс бэкенда нужно подключить этот линк.роутер (миддлвэйр он же).

app.use('/api/auth', require('./Routes/auth.routes'));  
app.use('/api/link', require('./Routes/link.routes'));

Теперь описываем логику каждого запроса в роутере линк. Проще всего с запросом ГЕТ, они только получают данные из БД (и возвращают).

В блок трай для получения всех ссылок, получаем объект Линкс и ждем пока модель Линк найдет все ссылки, которые относятся к текущему пользователю. Чтобы определить какой пользователь работает – сделаем позднее.

Если все хорошо, то в Жсоне возвращаем линкс.

router.get(  
 '/', //получаем список всех ссылок  
 async (req, res) => {  
 try {  
 const links = await Link.find({owner: null}) // как определить пользователя владельца???  
 res.json(links)  
 } catch (e) {

Почти то же в поиске ссылки по айди, меняем только метод поиска. Получаем отдельную ссылку (обращаемся к реквест парамс айди) и возвращаем ее в жсоне.

router.get(  
 '/:id', //получаем ссылку по айди  
 async (req, res) => {  
 try {  
 const link = await Link.findById(req.params.id);  
 res.json(link)

Возращаемся к пользователю и его определению. Нам нужно из фронтенда получить пользователя айди. Сделать можно по жвт-токену, в нем закодирован юзер.айди. Можем создать миддлвэйр для данного роутера, который будет проверять авторизован ли пользователь, валидел ли жвт-токен. И если да – получать данные нужные (позднее то же самое нужно будет сделать в методе женерейт здесь, модель новой ссылки привязать к пользователю).

В корне проекта новая папка миддлвэйр. В ней файл аус.миддлвэйр.жс. (миддлвэйр по сути это обычная функция, которая перехватывает данные и делает определенную логику).

К стандартным параметрам запроса рек и рес, добавляется некст, который позволяет продолжить запрос.

***module***.exports = (req, res, next) => {  
   
}

Сначала делаем проверку, что если реквест метод равен ОПШИНС (спец.метод в рестАПИ, который просто проверяет доступность сервера) и если норм, то продолжаем запрос.

***module***.exports = (req, res, next) => {  
 if (req.method === 'OPTIONS') {  
 return next()  
 }  
};

Дальше, если обычные запросы ПОСТ или ГЕТ, то выполняем в блоке трай/кэтч. Для начала нужно получить объект токена из реквест хедерс и дальше у хедеров берем поле авторизейшн (из фронт-енда берем). Мы можем эту строку распарсить, засплитить по пробелу и забрать токен (первый элемент).

Если токена нет, тогда делаем респонс, говорим что статус 401 (нет авторизации) и отправляем месседж «нет авторизации» и останавливаем код ретёрном.

try {  
 const token = req.headers.authorization.split(' ')[1]  
  
 if (!token) {  
 return res.status(401).json({message: 'No authorization'})  
 }  
} catch (e) {  
}

Если токе есть, тогда нужно его раскодировать. Для этого нужно подключить жвтвебтокен библиотеку. Подключаем и вызываем метод верифай. А вторым параметром передаем секретный ключ, передаваемый при создании. Подключаем и конфиг. Когда получаем раскодированный токен, можем положить его в объект реквеста. И ывполняем некст для продолжения запроса.

const jwt = require('jsonwebtoken');  
const config = require('config');

if (!token) {  
 return res.status(401).json({message: 'No authorization'})  
}  
  
const decoded = jwt.verify(token, config.get('jwtSecret'));  
req.user = decoded  
next()

Но в случае, если ошибка (при попытке достать токен, например) мы проваливаемся в кэтч и выводим ту же ошибку, что что-то пошло не так. И на этом миддлвэйр готов.

} catch (e) {  
 return res.status(401).json({message: 'Catch. No authorization'})  
}

Подключаем этот миддлвэйр как аус в файле линк.роутес.жс.

const auth = require('../Middleware/auth.middleware');

Далее этот миддлвэйр просто добавляем в запрос гет перед объявлением асинка.

router.get(  
 '/', //получаем список всех ссылок  
 auth,  
 async (req, res) => {

За счет того, что появился аус, в поле реквест у нас есть юзер. И у него поле юзер.айди. достаем этот айди. Соотвесно проблема получения всех ссылок пользователя решена.

router.get(  
 '/', //получаем список всех ссылок  
 auth,  
 async (req, res) => {  
 try {  
 const links = await Link.find({owner: req.user.userId});

Осталось реализовать метод дженерейт.