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

Оглавление

[Инициализация npm 1](#_Toc42200928)

[На следующем шаге нужно зарегестрировать роуты, которые будут обрабатывать апи запросы с фронтенд. 6](#_Toc42200929)

[Далее переходим на работу с фронт-ендом. Пишем команду и разворачиваем реакт-апп в папке Клиент 11](#_Toc42200930)

[Запуск фронтенда и бэкенда одновременно 11](#_Toc42200931)

[Библиотека materialize 12](#_Toc42200932)

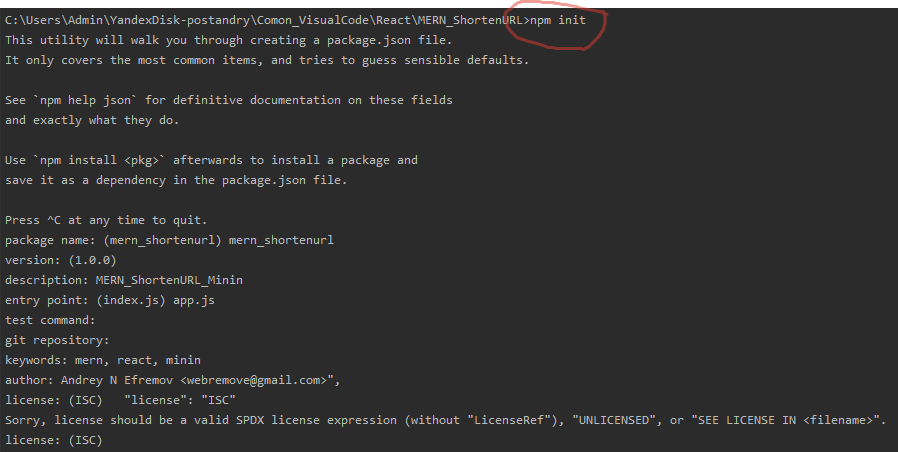
[Для переключения страниц подключить реакт-роутер-дом в папке реакта (Клиент) 12](#_Toc42200933)

[Готовим шаблоны страниц с материалайз 14](#_Toc42200934)

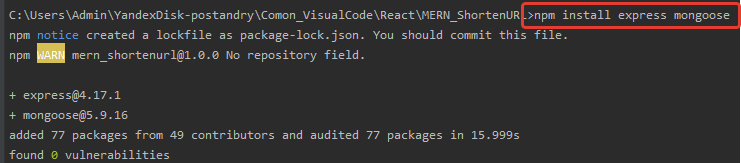
[Как отправлять определенные запросы на сервер для регистрации и авторизации 15](#_Toc42200935)

[Будем проксировать запросы с клиента на сервер. 19](#_Toc42200936)

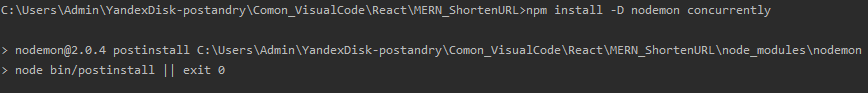
# Инициализация npm



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.

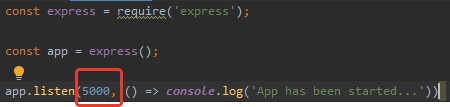


В пэккедж джейсон стартовую страницу меняем на апп, и добавляем сервер, который также стартет с апп.жс.

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

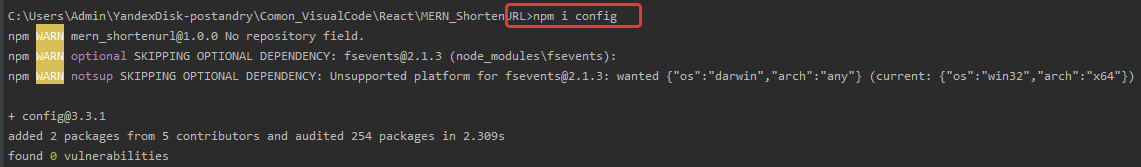


Чтобы не запускать сервер на захардкоденном порту 5000, нужно это вынести в общий конфиг.



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.

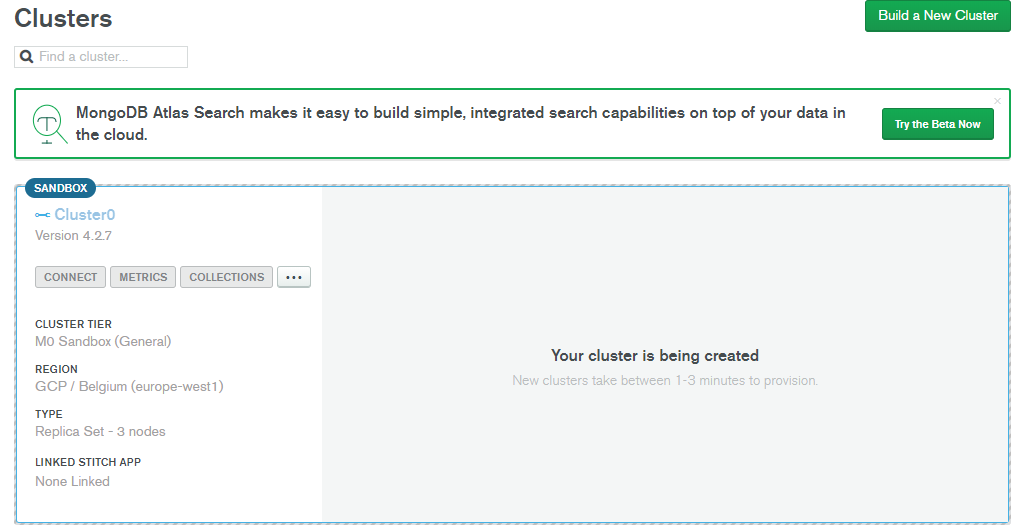


Для подключения к МонгоДБ подключаем пакет монгус. И вызвать метод коннект, чтобы подключиться к базе данных. Делаем через асинк эуэйт обертку. В конфиг добавляем пустое значение монгоУри, которое передаем в коннект первым параметром. Это тот адрес, по которому будем добавлять базу данных.

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

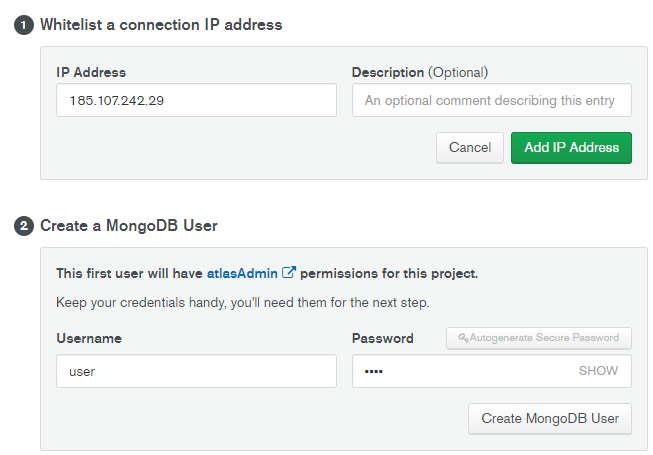
async function start() {  
 try {  
 await mongoose.connect(config.get('mongoUri'), {  
//пока пустые опции  
 })  
 app.listen(PORT, () => ***console***.log(`App has been started on port ${PORT}...`));  
 } catch (e) {  
 ***console***.log('Server Error', e.message)  
 process.exit(1)  
 }  
}  
  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

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

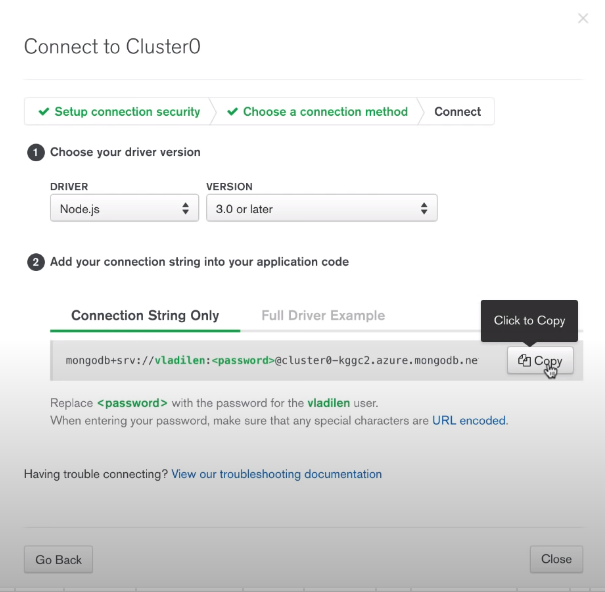


Настроим МонгоКоннект, добавим параметры, чтобы коннект успешно работал.

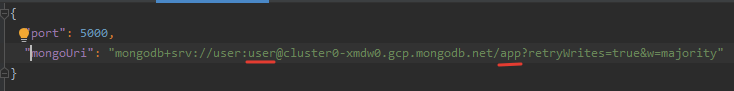
try {  
 await mongoose.connect(config.get('mongoUri'), {  
 useNewUrlParser: true,  
 useUnifiedTopology: true,  
 useCreateIndex: true  
 })

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

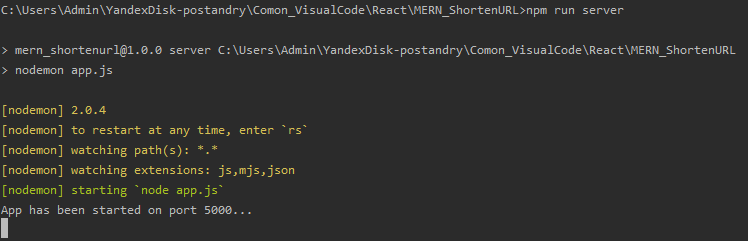
Потом опцию коннект йо аппликейшн. Копируем строку коннекта к базе. И вставляет в папке Конфиг в файл дефолт.жсон в строку монгоУри.



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}  
})  
  
model.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'}]  
});  
  
model.exports = model('user', schema);

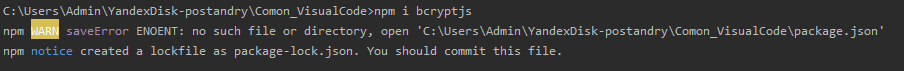
Модель подключаем в аус.роутес.жс.

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

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

Хэшируем пароль для безопасности. С помощью библиотеки bcrytp.js



Добавляем в аус.роутерс.жс

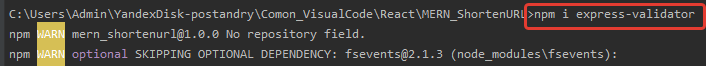
const bcrypt = require('bcryptjs')

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.

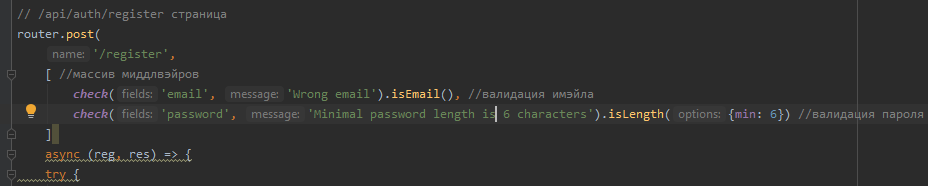


Подключаем и экспортируем методы чек и валидэйшнРезалт.

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  
],

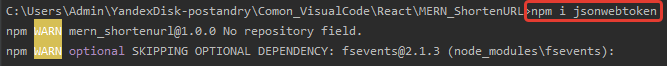
Далее логика по логированию пользователя. Сначала ищем есть ли зареганный и если нет даем ошибку

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

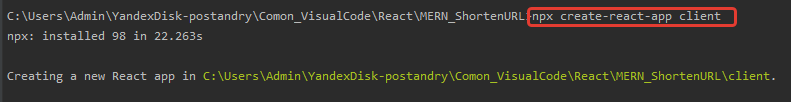
const token = jwt.sign(  
 {userId: user.id}, //данные, которые будут зашифрованы в этом токене. можно добавить мэйл, логин и другое  
 config.get('jwtSecret'), //секретную фразу передаём из настроек  
 {expiresIn: '1h'} //время жизни токена  
)

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

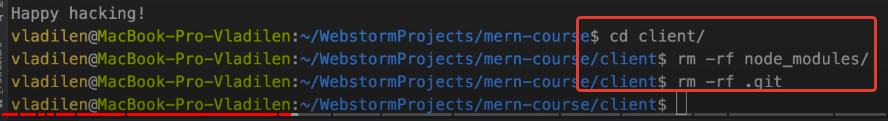
res.json({token, userId: user.id}) //ответ 200, успешная авторизация

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

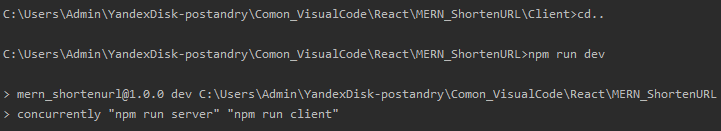
В пакэдж.жсон добавляем скрипт, запускающий реакт из папки клиент. Ранее только сервер запускался.

"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.



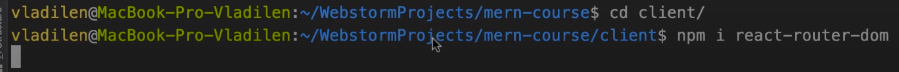
Импортируем в апп.жс материалайз цсс

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

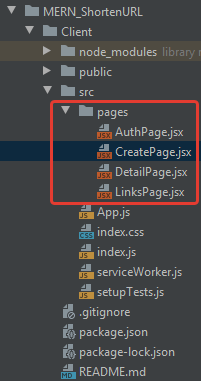
в индекс.цсс импортируем

@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

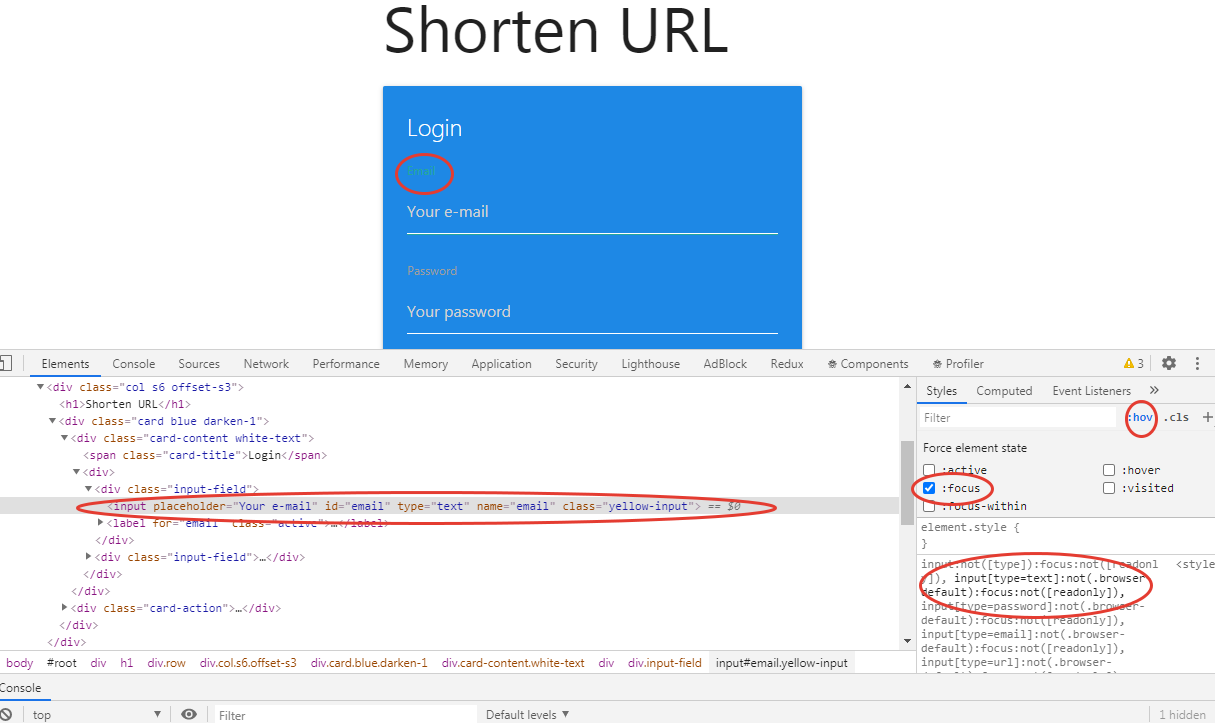
.col.s6.offset-s3

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

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}  
}

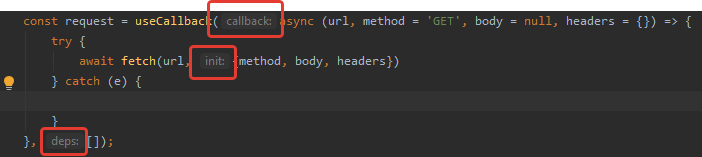
Также добавляем стэйт еррор с фнукцией сетЕррор и умолчание значение фолс.

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) {  
  
 }  
}

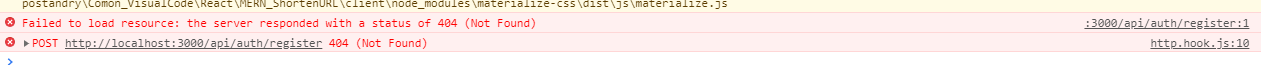
Этот метод привязываем к кнопке регистрации. И если работает режим лоадинг (загружаются данные), то нам нужно задизэйблить кнопку, то есть дзейбл должен быть тру, если лоадинг. Делаем это для обеих кнопок

<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

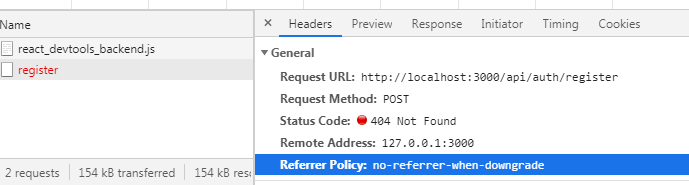
const data = await request('/api/auth/register', 'POST', {...form});  
***console***.log('data', data)

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.

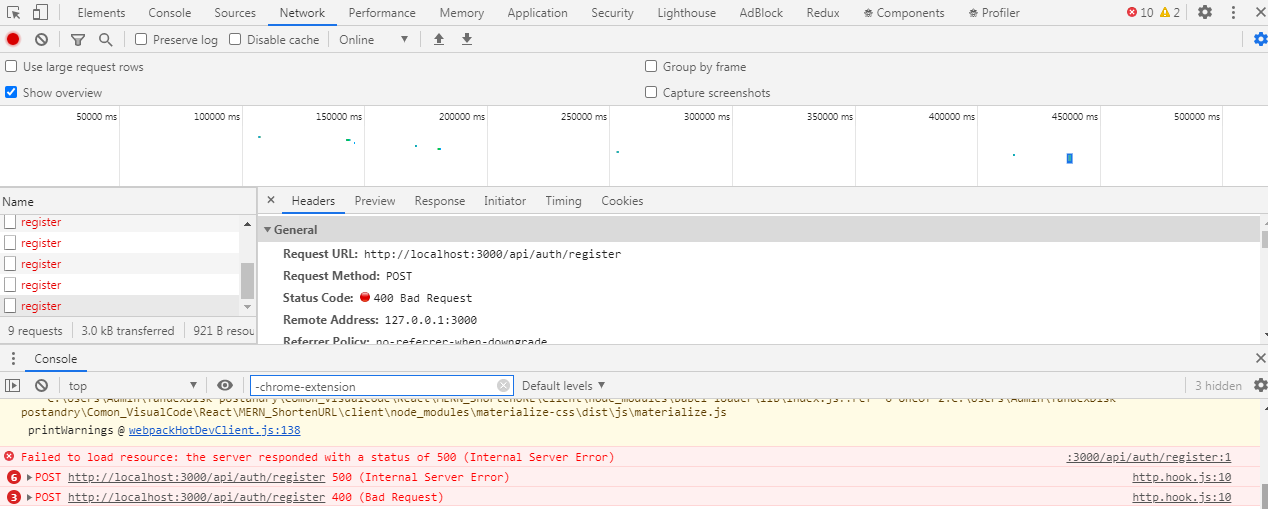
# Будем проксировать запросы с клиента на сервер.

Останавливаем сервер.

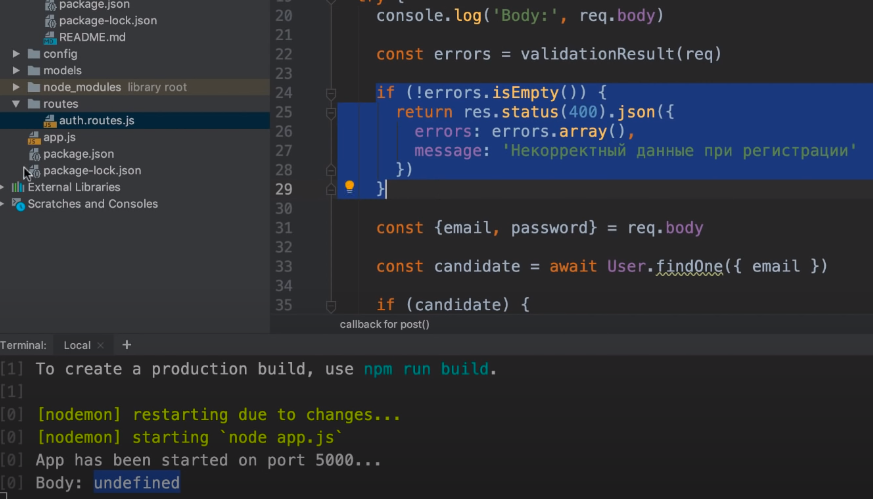
Для проксирования в папке реакта Клиент открыть пэккэдж.жсон. Добавить поле прокси с проксирование всех запросов на порт 5000 (на сервер). Это для стадии разработки. На продакшене по-другому будет приложение запускаться.

"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).



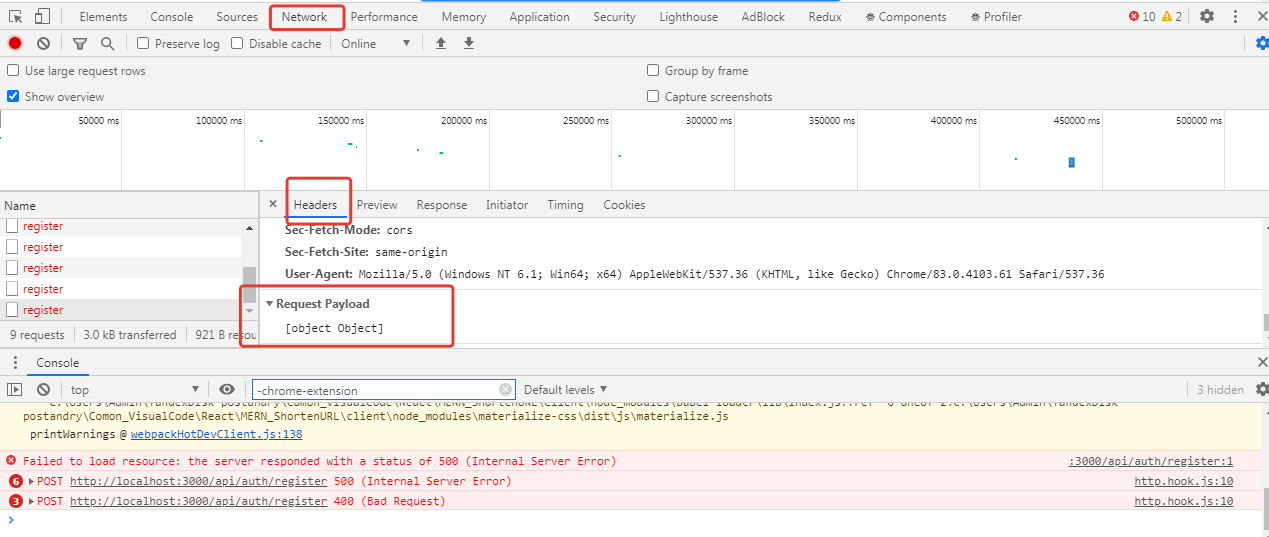
По умолчанию, нод жс воспринимает бади как стрим. Чтобы избежать ошибок, нужно добавить новый миддлвэйр для бади. Он будет парсить в жсон формат бади для сервера. Для этого идем в главный файл сревера апп.жс и добавляем миддлвэйр экспресса, называется жсон. И передаем один парамеьтр экстендед: тру.

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