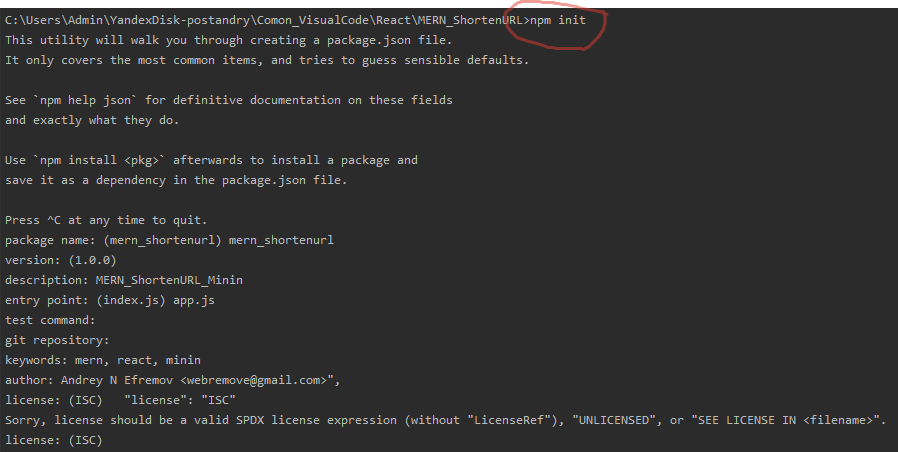
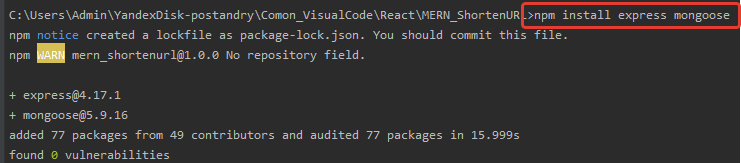
<https://www.youtube.com/watch?v=ivDjWYcKDZI&t=25s>

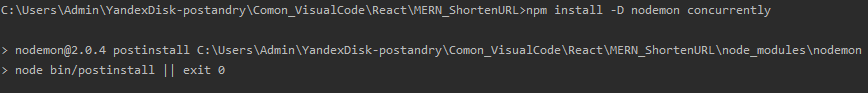
Инициализация npm



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.

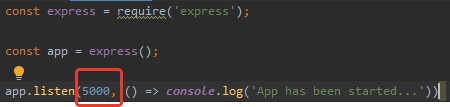


В пэккедж джейсон стартовую страницу меняем на апп, и добавляем сервер, который также стартет с апп.жс.

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

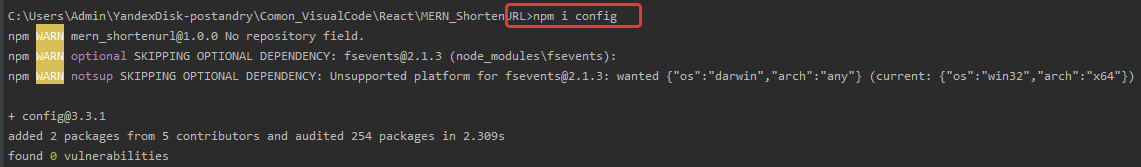


Чтобы не запускать сервер на захардкоденном порту 5000, нужно это вынести в общий конфиг.



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.

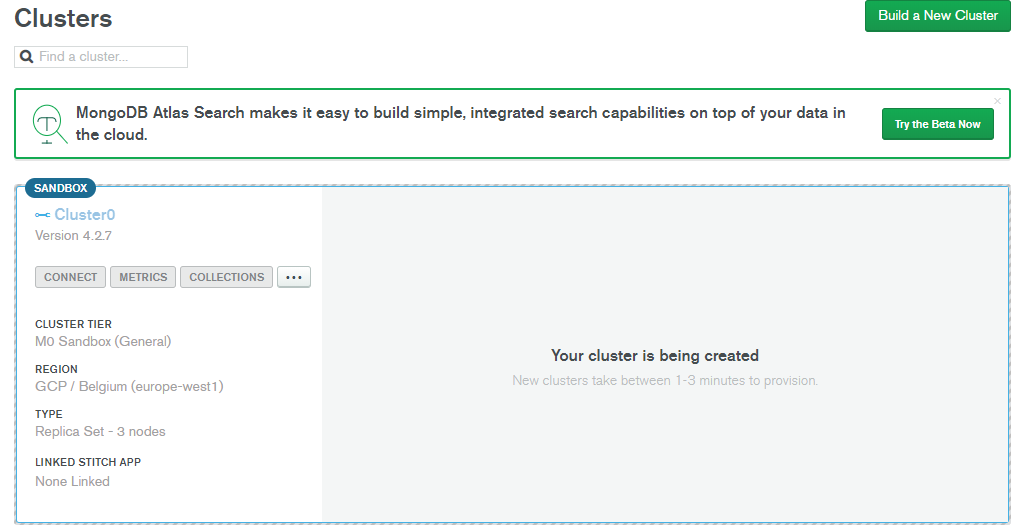


Для подключения к МонгоДБ подключаем пакет монгус. И вызвать метод коннект, чтобы подключиться к базе данных. Делаем через асинк эуэйт обертку. В конфиг добавляем пустое значение монгоУри, которое передаем в коннект первым параметром. Это тот адрес, по которому будем добавлять базу данных.

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

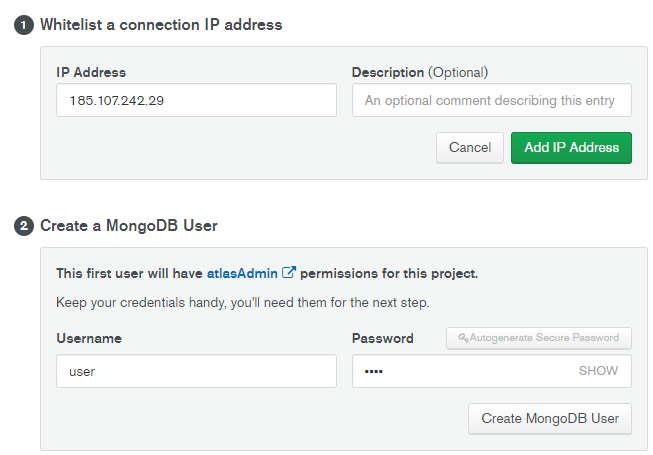
async function start() {  
 try {  
 await mongoose.connect(config.get('mongoUri'), {  
//пока пустые опции  
 })  
 app.listen(PORT, () => ***console***.log(`App has been started on port ${PORT}...`));  
 } catch (e) {  
 ***console***.log('Server Error', e.message)  
 process.exit(1)  
 }  
}  
  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

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

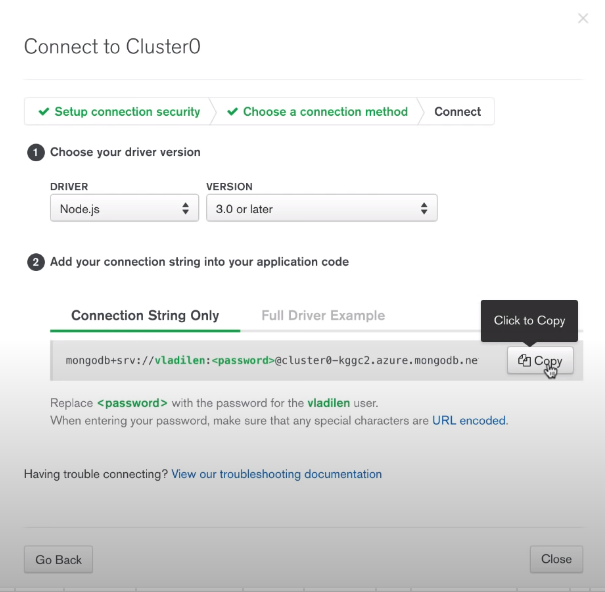


Настроим МонгоКоннект, добавим параметры, чтобы коннект успешно работал.

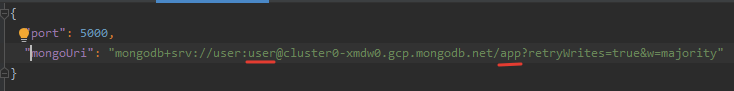
try {  
 await mongoose.connect(config.get('mongoUri'), {  
 useNewUrlParser: true,  
 useUnifiedTopology: true,  
 useCreateIndex: true  
 })

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

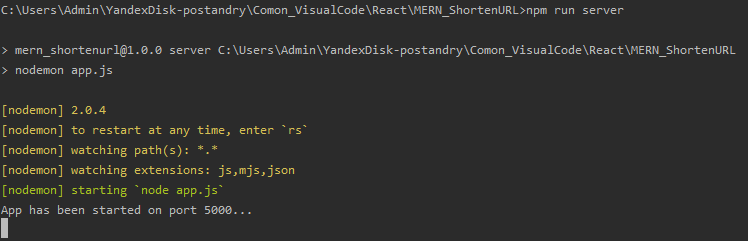
Потом опцию коннект йо аппликейшн. Копируем строку коннекта к базе. И вставляет в папке Конфиг в файл дефолт.жсон в строку монгоУри.



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

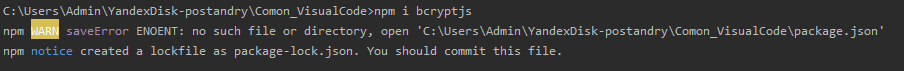
Модель подключаем в аус.роутес.жс.

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

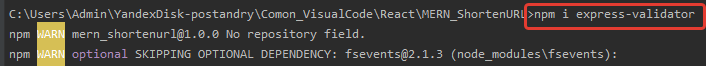
const bcrypt = require('bcryptjs')

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.

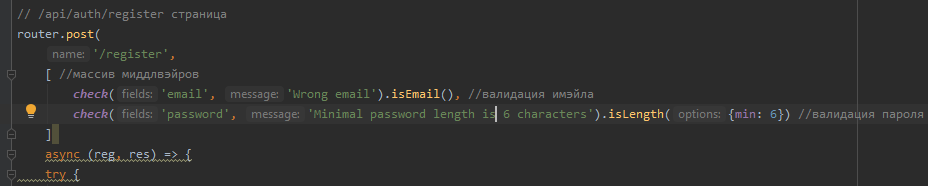


Подключаем и экспортируем методы чек и валидэйшнРезалт.

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 (reg, 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 (reg, 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'})  
}