**Описание разработки**

Создаётся пустой проект. Создаётся 2 папки: “client” и “server”. В папке “server” создаётся файл “index.js”.

*С него будет начинаться запуск приложения.*

После создания файла запускаем терминал и прописываем команду

“cd server” *для входа в папку сервера*

А затем команда

“npm init -y” *для инициализации проекта. Появится файл* “package.json”. В нем находится описание проекта. Зависимости, которые в нем используются.

Установим зависимости с помощью команды

“npm install express mssql sequelize cors dotenv”

А также

“npm install -D nodemon” для автоматического перезапуска сервера после сохранений.

Далее мы пишем скрипт в файле package.json

"scripts": {  
 "dev": "nodemon index.js"  
},

Скрипт по запуску приложения в режиме разработки. Команда dev и исполняемая команда nodemon index.js.

Переходим к созданию структуры приложения

(с помощью require можно импортировать модули в файл)

В файле index.js импортируем модуль «express»

const ***express*** = ***require***('express')

Следующим этапом создадим объект вызвав функцию express. С него и будет начинаться запуск нашего приложения.

const app = ***express***()

Дальше укажем порт, на котором наше приложение будет работать.

const PORT = 5000

Затем у app вызываем функцию listen в которой указываем какой порт должен прослушивать наш сервер, а вторым параметром передаем callback, который отработает при успешном запуске сервера. Напишем, что сервер стартовал на этом порту.

app.listen(PORT, () => console.log(`Сервер работает на порту ${PORT}`))

После чего в консоле пишем команду “npm run dev” для запуска скрипта.

Теперь создадим файл «.env»

В нём просто указываем название переменной и его значение

PORT=7000

В index.js иизменяем поле «const PORT = 5000» на «const PORT = process.env.PORT || 5000

Но для того, чтобы index.js мог считывать этот файл нужно импортировать этот файл

require(‘dotenv’).config()

Далее я немного изменил визуал вывода функции listen таким образом

app.listen(PORT,()=>***console***.log('\x1b[40m','\x1b[33m',`Сервер запущен на порту \x1b[31m ${PORT}`, '\x1b[0m'))

'\x1b[40m','\x1b[33m' означает изменение background на черный (как сброс) и изменение цвета основного текста на желтый.

\x1b[31m ${PORT}`, '\x1b[0m' изменяет цвет значения переменной PORT на красный, а затем сброс всех цветов на стандарт для дальнейших сообщений.

Следующим этапом сконфигурируем подключение к базе данных.

Создадим «db.js» в папке server.

В файле «db.js».

Импортируем sequelize. Сразу делаем деструктуризацию. Так как модуль большой и нам нужен конкретно этот класс. Затем на выходе мы экспортируем объект, который мы создаем из этого класса. В конструкторе как раз и будем указывать конфигурацию.

const{Sequelize} = ***require*** ('sequelize')  
***require*** ('tedious')  
***module***.exports = new Sequelize(

Все настройки подключения к базе данных также вынесем в переменные окружения. (файл .env)

PORT = 7000  
DB\_NAME = webisod  
DB\_USER = ubersk  
DB\_PASSWORD = 12511  
DB\_HOST = localhost  
DB\_PORT = 1433

В db.js обозначим все параметры подключения. Крайне важно, при использовании MSSQL обратить внимание на SQL Server и его стабильную работу. Особенно на возможность проверки подлинности SQL Server и Windows в настройках безопасности SSMS.

const{Sequelize} = ***require*** ('sequelize')  
***require*** ('tedious')  
***module***.exports = new Sequelize(  
 ***process***.env.DB\_NAME,  
 ***process***.env.DB\_USER,  
 ***process***.env.DB\_PASSWORD,  
 {  
 host: ***process***.env.DB\_HOST,  
 dialect: "mssql",  
 port: ***process***.env.DB\_PORT,  
 dialectOptions: {  
 },  
 }  
);

Возвращаемся в index.js. В нём импортируем объекткоторый мы сделалем в файле db и который оттуда же экспортировали.

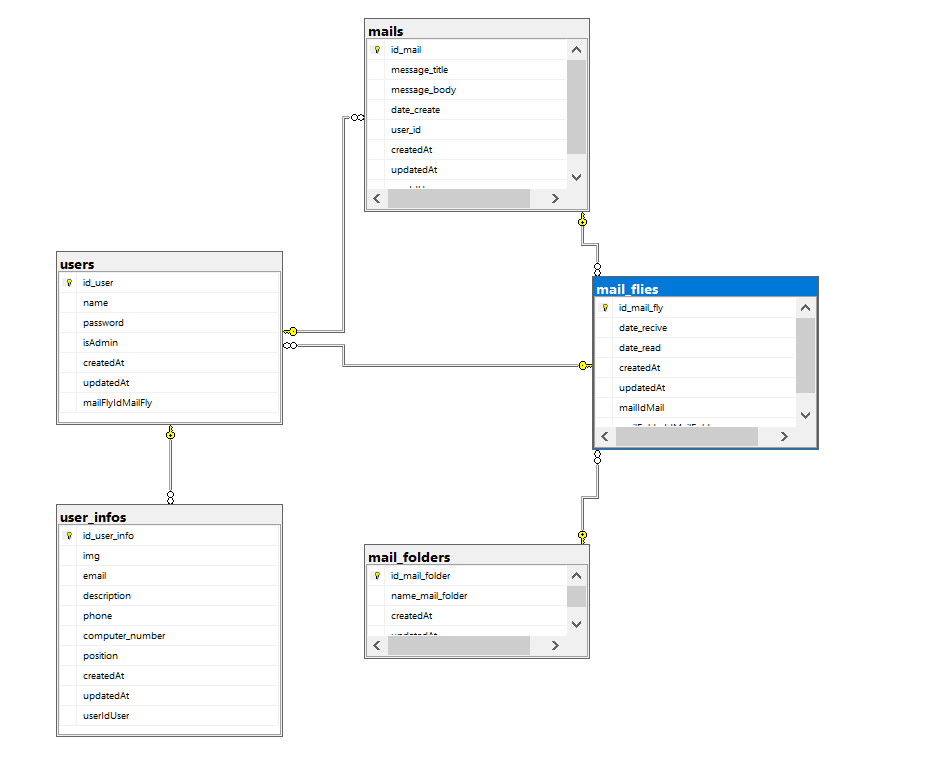
const ***sequelize*** = ***require***('./db')

Импортируем и вызываем функцию для подключения к базе данных.

Обязательно делаем ассинхронной. Всё что связано с базой данных должно быть ассинхронным. Вкладываем наш прослушиватель в эту функцию. Вызываем функцию authenticate для подключения к базе данных и далее вызываем функцию sync которая сверяет состояние базы данных со схемой данных.

const start = async()=>{  
 try{  
 await ***sequelize***.authenticate()  
 await ***sequelize***.sync()  
 app.listen(PORT,()=>***console***.log('\x1b[40m','\x1b[33m',`Сервер запущен на порту \x1b[31m ${PORT}`, '\x1b[0m'))  
 } catch (e){  
 ***console***.log(e)  
 }  
}

Далее мы создаем диаграмму БД.



Потом перенесём эту диаграмму в наш проект.

Создаем папку models и файл models.js.

В нём мы будем описывать модели базы данных.

Импортируем объект sequelize который мы создавали в db.js.

const ***sequelize*** = ***require***('../db')

Из самого пакета sequelize нам понадобится импортировать класс DataTypes, с помощью которого описываются типы того или иного поля.

const {DataTypes} = ***require*** ('sequelize')

Начинаем описывать модели. Начнем с модели пользователя.

У sequlize вызываем функцию define. Передаем туда объект. Первым параметром передаем название модели. И уже внутри объекта описываем поля, которые будут у этой модели.

const User = ***sequelize***.define("user", {  
 id\_user: { type: DataTypes.***INTEGER***, primaryKey: true, autoIncrement: true },  
 name: { type: DataTypes.***STRING*** },  
 password: { type: DataTypes.***INTEGER*** },  
 isAdmin: { type: DataTypes.BOOLEAN },  
});  
const UserInformation = ***sequelize***.define("user\_info", {  
 id\_user\_info: {  
 type: DataTypes.***INTEGER***,  
 primaryKey: true,  
 autoIncrement: true,  
 },  
 img: { type: DataTypes.***STRING*** },  
 email: { type: DataTypes.***STRING*** },  
 description: { type: DataTypes.***STRING*** },  
 phone: { type: DataTypes.***STRING*** },  
 computer\_number: { type: DataTypes.***STRING*** },  
 position: { type: DataTypes.***STRING*** },  
});  
  
const Mail = ***sequelize***.define("mail", {  
 id\_mail: { type: DataTypes.***INTEGER***, primaryKey: true, autoIncrement: true },  
 message\_title: { type: DataTypes.***STRING***(155) },  
 message\_body: { type: DataTypes.***TEXT*** },  
 date\_create: { type: DataTypes.***DATE***, defaultValue: DataTypes.NOW },  
 user\_id: { type: DataTypes.***INTEGER*** },  
});  
const Mail\_fly = ***sequelize***.define("mail\_fly", {  
 id\_mail\_fly: {  
 type: DataTypes.***INTEGER***,  
 primaryKey: true,  
 autoIncrement: true,  
 },  
 date\_recive: { type: DataTypes.***DATE***, defaultValue: DataTypes.NOW },  
 date\_read: { type: DataTypes.***DATE***, defaultValue: DataTypes.NOW },  
});  
const Mail\_folder = ***sequelize***.define("mail\_folder", {  
 id\_mail\_folder: {  
 type: DataTypes.***INTEGER***,  
 primaryKey: true,  
 autoIncrement: true,  
 },  
 name\_mail\_folder: { type: DataTypes.***STRING*** },  
});  
  
User.*hasOne*(UserInformation);  
UserInformation.*belongsTo*(User);  
  
User.*hasMany*(Mail);  
Mail.*belongsTo*(User);  
  
Mail.*hasMany*(Mail\_fly);  
Mail\_fly.*belongsTo*(Mail);  
  
Mail\_fly.*hasMany*(User);  
User.*belongsTo*(Mail\_fly);  
  
Mail\_folder.*hasMany*(Mail\_fly);  
Mail\_fly.*belongsTo*(Mail\_folder);  
  
***module***.exports = {  
 User,   
 Mail,  
 UserInformation,  
 Mail\_folder,  
 Mail\_fly,  
};

Как только прописали экспорт сущностей переходим в файл index.js и прописываем импорт модуля models.js, чтобы создать базу данных.

const ***models*** = ***require***("./models/models");

После этого сохраняем проект и в консоле будет информация о создание нашей базы данных со всеми связями.

Мы успешно подключили базу данных MSSQL, создали её и можем продолжить работу над реализацией запросов к базе данных.

В файле «index.js» пропишем

Настроем cors для того, чтобы мы могли принимать запрос с браузера. Cors – это совместное использование ресурсов между источниками. Для этого импортируем функцию cors из пакета в index.js

const cors = ***require***("cors");

и передадим эту фукцию в функцию use у app

app.use(cors());

и также в функцию use передадим express.json чтобы наше приложение могло парсить json формат

app.use(express.***json***());

и попробуем создать первый get метод.

Для этого вызываем функцию get. Первым параметром передаем url по которому этот запрос будет отрабатывать, а вторым параметром функцию callback которая принимает параметрами запрос и ответ (request и response).

В зависимости от ситуации мы можем на клиент возвращать разные статус коды. 200 говорит о том, что запрос произошел без ошибок и в функцию json мы передаем тело ответа. Передаем объект с телом ответа с полем message «Работает!»

app.get("/", (req, res) => {  
 res.status(200).json({message: "Работает!" });  
});

Заходим в браузер и прописываем адрес localhost:7000. Видим успешное выполнение запроса.

Тестировать все методы будем через программу Postman. Это клиент позволяющий отправлять запросы на сервер.

Удаляем написанный тестовый запрос и продолжим работу.

Начнём реализацию каркаса приложения. Начнём с маршрутов, по которым будут отрабатывать те или иные методы. Создаём папку routes. Создаём js файлы для наших методов. Не забываем про новый index.js в папке routes. Он как связующие звено для остальных путей. Отдельный файл с маршрутами. Основной роутер.

Получаем роутер из express.

const *Router* = ***require***("express");

Создаем объект этого роутера.

const *router* = new *Router*();

И по итогу этот роутер из этого файла мы экспортируем.

***module***.exports = *router*;

Т.к остальные файлы по сути «подроутеры» мы должны в основном роутере это указать. Вызываем функцию use.

*router*.use

Указываем URL по которому роутер будет отрабатывать.

("/user",

А вторым нужно будет передать сам роутер.

*router*.use("/user", );

Копируем и дублируем в другой роутер(любой). Будет использоваться метод post для создания и get для получения. В userRouter прописываем post логина и get авторизации, который мы пропишем позже по jwt токену.

const *Router* = ***require***("express");  
const *router* = new *Router*();  
  
*router*.post("/login");  
*router*.get("/auth”);  
  
***module***.exports = *router*;

Для остальных роутов.

const *Router* = ***require***("express");  
const *router* = new *Router*();  
  
*router*.post("/");  
*router*.get("/");  
  
***module***.exports = *router*;

mailRouter плюс один метод get для просмотра сообщения подробнее.

const *Router* = ***require***("express");  
const *router* = new *Router*();  
  
*router*.post("/");  
*router*.get("/");  
*router*.get("/:id");  
  
***module***.exports = *router*;

Возвращаемся в index.js и все прописанные роутеры импортируем в этот файл.

const *Routes* = ***require***("express");  
const *router* = new *Routes*();  
const *userRouter* = ***require***("./userRouter");  
const *userInfoRouter* = ***require***("./userInfoRouter");  
const *mailFlyRouter* = ***require***("./mailFlyRouter");  
const *mailFolderRouter* = ***require***("./mailFolderRouter");  
const *mailRouter* = ***require***("./mailRouter");

Далее сопоставляем маршруты с роутерами в router.use.

*router*.use("/user", *userRouter*);  
*router*.use("/userInfo", *userInfoRouter*);  
*router*.use("/mail", *mailRouter*);  
*router*.use("/mailFly", *mailFlyRouter*);  
*router*.use("/mailFolder", *mailFolderRouter*);

Мы объединили все роутеры, но нужно сказать серверу об этом. Возвращаемся в корневой index.js и импортируем основной роутер.

const *router* = ***require***("./routes/index");

Затем вызываем функцию app.use. Первым параметром указываем url по которому ротер должен обрабатываться. Вторым параметром сам router.

*app*.use("/api", *router*);

В файле userRouter в методе get вернем сообщение как это делали раньше.

*router*.get("/auth", (*req*, *res*) => {  
 *res*.json({ message: "Работает!" });  
});

Убеждаемся, что всё у нас работает. Все маршруты работают и с ними можно работать.

Теперь отсединим логику. Создадим папку controllers и в ней для каждого роутера создадим controller. Отметим, что мы не будем создавать контроллер для таблицы сообщения получателя(mailfly), ибо его запросы будут реализованы в mailController. Для userController создаём класс с двумя ассинхронными методами принимающих два параметра, request и result. Также незабываем про экспорт данного класса. Можно сразу проверить его работоспособность прописав json запрос.

class UserController {  
 async login(*req*, *res*) {}  
  
 async check(*req*, *res*) {  
 *res*.json("Example");  
 }  
}  
  
***module***.exports = new UserController();

В остальных контроллерах мы прописываем подобные классы

class MailFolderController {  
 async create(*req*, *res*) {}  
  
 async getAll(*req*, *res*) {}  
}  
  
***module***.exports = new MailFolderController();

Но не в mailController. В нём их будет чуть больше.

class MailController {  
 async createMsg(*req*, *res*) {}  
  
 async basketMsg(*req*, *res*) {}  
  
 async hideMsg(*req*, *res*) {}  
  
 async getAllMsg(*req*, *res*) {}  
  
 async getOneMsg(*req*, *res*) {}  
}  
  
***module***.exports = new MailController();

create на создание сообщения. basketMsg для переноса сообщения в корзину. hideMsg для скрытия сообщения пользователя(удаление) .getAllMsg на запрос показать все сообщения и getOneMsg чтобы посмотреть определенное сообщение.

Объясняем как работают функции.

const *Router* = ***require***("express");  
const *router* = new *Router*();  
const *userController* = ***require***("../controllers/userController");  
*router*.post("/login", *userController*.login);  
*router*.get("/auth", *userController*.check);  
  
***module***.exports = *router*;

В userRouter.js импортируется контроллер userController. Затем в каждый post или get методов вторым параметром передаем соответствующую функцию.

const *Router* = ***require***("express");  
const *router* = new *Router*();  
const *userController* = ***require***("../controllers/userController");  
*router*.post("/login", *userController*.login);  
*router*.get("/auth", *userController*.check);  
  
***module***.exports = *router*;

Обратим внимание, что передаем мы функции без скобок, то есть мы их не вызываем, а просто передаем как объект.

Эту процедуру нужно провести и для остальных роутов.

mailFolder.Router.js

const *Router* = ***require***("express");  
const *router* = new *Router*();  
const *mailController* = ***require***("../controllers/mailFolderController");  
*router*.post("/", *mailController*.createFolderMsg);  
*router*.get("/", *mailController*.getFolderMsg);  
  
***module***.exports = *router*;

mailRouter.js

const *Router* = ***require***("express");  
const *router* = new *Router*();  
const *mailController* = ***require***("../controllers/mailController");  
*router*.post("/", *mailController*.createMsg);  
*router*.get("/", *mailController*.getAllMsg);  
*router*.get("/:id", *mailController*.getOneMsg);  
  
***module***.exports = *router*;

userInfoRouter.js

const *Router* = ***require***("express");  
const *router* = new *Router*();  
const *userInfoController* = ***require***("../controllers/userInfoController");  
*router*.post("/", *userInfoController*.create);  
*router*.get("/", *userInfoController*.getAll);  
*router*.get("/:id", *userInfoController*.getOne);  
  
***module***.exports = *router*;

Функция попадает в get метод и когда мы открываем определенный URL эта функция отрабатывает.

Перед тем как мы займемся взаимодействием с базой данных и логикой выдачи данных на клиент сделаем универсальную обработку ошибок. «Универсальный handler». Рассмотрим на примере.

async check(*req*, *res*, *next*) {  
 const { *id* } = *req*.query;  
 if (!*id*) {  
 return *next*(*ApiError*.*badRequest*("Не задан ID"));  
 }  
 *res*.json(*id*);  
 }  
}

Откроем фукнцию check. Из строки запроса обращаясь к request query мы можем получать параметры строки запроса. Вернем на клиент и увидим в браузере, что ничего не передается, но если мы передадим в URL параметры, то они отобразятся успешно. Но допустим нас интересует только параметр id, чтобы не получать весь объект query можем воспользоваться деструктуризацией и сразу вытащить id. НО, допустим этот параметр обязателен и если пользвоатель не укажет, то мы должны бросить ошибку.

Создадим папку error и там файл ApiError.js. В нём мы создадим класс и реализуем констуруктор который принимает параметрами статус код и сообщение которо мы будем возвращать на клиент. Класс будет расширять error (extends Error). Вызываем родительский конструктор с помощью функции super(). И здесь мы присваеваем то, что получаем параметрами.

class ApiError extends Error {  
 constructor(*status*, *message*) {  
 super();  
 this.status = *status*;  
 this.message = *message*;

Теперь создадим пару статичных функций.

Статические функции – это функции которые можно вызывать без создания объекта. То есть мы можем обращаться напрямую к классу и вызывать ту или иную функцию. Первая функция (badrequest), которая будет возвращать новый объект ApiError. Первым параметром передается статус ошибки, вторым сообщение, которое будем получать параметром.

static *badRequest*(*message*) {  
 return new ApiError(404, *message*);  
}

Таким образом можно создать несколько функций под разные статус коды.

static *internal*(*message*) {  
 return new ApiError(500, *message*);  
 }  
  
 static *forbidden*(*message*) {  
 return new ApiError(403, *message*);  
 }  
}  
  
***module***.exports = ApiError;

Следующим этапом создадим первый middleware.

Что такое middleware? В веб разработке есть клиентская часть и есть сервер на котором крутится серверное приложение. С клиента отправляется запрос по определенному маршруту(req) и какая-то функция на сервере этот запрос должна обработать и либо вернуть какие-то данные либо их изменить и вот тут нужен middleware. В данном контексте middleware является промежуточным звеном между запросом и функцией. То есть прежде, чем запрос попадёт в функцию он должен пройти цепочку middleware. Их может быть много и все они вызываются последовательно. Как примером: сначала middleware проверка запроса на соответствие cors. Или фунция обрабатывает только авторизованных пользователей и авторизация проходит через token доступ. И прежде чем запрос попадает в эту функцию мы попадаем в middleware который проверит токен. Если токен валидный и пользователь авторизован, то вызывается следующий middleware. Если нет, то бросаем ошибку. То есть запрос до функции не дойдет. Как ещё один пример это проверка ролей. Допусти некотоыре функции доступны только администраторам, потому идет проверка является ли пользователь админом или нет. Только после этой цепочки запрос доходит до функции, обрабатывается и возвращается обработанный на клиент.

Внутри папки middleware создадим файл ErrorHandlingMiddleware.js. Импортируем туда класс ApiError. И из самого файла экспортируем функцию. По сути, данная функция и есть middleware. Она принимает следующие параметры: ошибка(err), запрос(req), ответ(res) и функция next, вызвав которую мы передадим управление следующему в цепочке middleware.

const *ApiError* = ***require***("../error/ApiError");  
***module***.exports = function (*err*, *req*, *res*, *next*) {

Дальшей в условие мы проверим. Если класс ошибки ApiError, то тогда на клиент возвращаем ответ со статус кодом, который будем получать из ошибки. И с сообщением, которое мы в эту ошибку поместили. Чтобы прекращение функции завершилось поставим return перед возвратом ответа и если вдруг сюда попадет ошибка, которая не является инстансом ApiError, то вернем 500 ошибку и в сообщение укажем, что ошибка была не предвиденной и мы её никак не обработали.

const *ApiError* = ***require***("../error/ApiError");  
***module***.exports = function (*err*, *req*, *res*, *next*) {  
 if (*err* instanceof *ApiError*) {  
 *res*.status(*err*.status).json({ message: *err*.message });  
 }  
 return *res*.status(500).json({ message: "Непредвиденная ошибка!" });  
};

Зарегестрируем данных middleware в корневом index.js.

const *errorHandler* = ***require***("./middleware/ErrorHandlingMiddleware");

Вызываем его строго после остальных app.use.

const *errorHandler* = ***require***("./middleware/ErrorHandlingMiddleware");  
*app*.use(*cors*());  
*app*.use(*express*.***json***());  
*app*.use("/api", *router*);  
*app*.use(*errorHandler*);

Попробуем воспользоваться. Переходим в userController. Если в функции check пользователь не укажет id, то вызываем функцию next, которая идёт третим параметром. Не забываем return, чтобы код дальше не выполнялся.

const *ApiError* = ***require***("../error/ApiError");  
  
class UserController {  
 async login(*req*, *res*) {}  
  
 async check(*req*, *res*, *next*) {  
 const { *id* } = *req*.query;  
 if (!*id*) {  
 return *next*(*ApiError*.*badRequest*("Не задан ID"));  
 }  
 *res*.json(*id*);  
 }  
}  
  
***module***.exports = new UserController();

В браузере можем проверить то, что получилось.

# Теперь научимся добавлять в базу данных объекты.

Реализуем это вначале с вкладками писем. Переходим в файл mailFolderController. Импортируем из models модель типа Mail\_folder. Импортируем и ApiError.

В функции создания (create) сразу делаем деструктуризацию.

const {*name\_mail\_folder*} = *req*.body;

И из тела запроса извлекаем название этой папки. Затем с помощью функции create эту папку мы создаем. Не забываем добавлять await, потому что это ассинхронная функция. В параметр функции create передаем объект где указываем нужные поля.

async createFolderMsg(*req*, *res*) {  
 const {*name\_mail\_folder*} = *req*.body;  
 const *mailFolder* = await *Mail\_folder*.*create*({*name\_mail\_folder*});  
 return *res*.json(*mailFolder*);  
}

Не забываем указывать функции, которые должен отрабатывать на тот или иной маршрут.

const *Router* = ***require***("express");  
const *router* = new *Router*();  
const *mailController* = ***require***("../controllers/mailFolderController");  
*router*.post("/", *mailController*.createFolderMsg);  
*router*.get("/", *mailController*.getFolderMsg);  
  
***module***.exports = *router*;

Вместе с этим в остальных роутах также сделаем связь с контроллерами.

Через postman можем проверить метод создания папок писем на работоспособность. Всё работает, теперь реализуем метод получения данных.

Создаём переменную, назовём mailFolders и у модели Mail\_folder вызываем функцию findAll, которая покажет нам все записи, которые есть в базе данных. Не забываем об ассинхронности функции.

const *mailFolders* = await *Mail\_folder*.*findAll*()

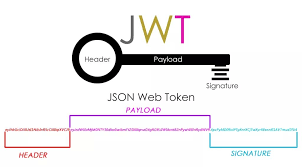
На клиент возвращаем полностью весь массив объектов.

async getFolderMsg(*req*, *res*) {  
 const *mailFolders* = await *Mail\_folder*.*findAll*()  
 return *res*.json(*mailFolders*)  
}

Работаем над авторизацией пользователя.

Что такое JWT-token. Это некоторая строка, состоящая из трех чатей.

1 – header, заголовок. 2 – payload, полезные данные. 3 – signature, подпись.



HEADER - содержит информацию о том, как должна вычисляться подпись. Определяет алгоритм хэширования.

PAYLOAD – полезные данные, которые хранятся внутри JWT. Эти данные также называют JWT-claims(заявки). Туда можно поместить ID пользователя, его роли, username и прочую полезную информацию.

SIGNATURE – допустим есть некий секретный ключ о котором знает только сервер. По этому ключу в сигнатуру шифруются какие-то данные. Именно по нему сервер потом может этот токен расшифровать.

В нашем случае в Payload мы будем закладывать UserID, ФИО и логическую проверку админ он или нет.

После того как пользователь ввёл ФИО и пароль для него генерируется подобный токен. Все его данные вшиваются в PAYLOAD и важно понимать, что эти данные никак не шифруются. Но они и не имеют секретной информации. На клиенте мы их можем спокойно расшифровывать. Но проверить валиден токен или нет можно только с помощью секретного ключа, который объявлен на сервере. Его никто знать не должен. В первую очередь мы проверяем существует ли пользователь с таким ФИО в нашей системе. Если да, то сравниваем пароль, который находится в базе данных с паролем который вводит пользователь. Если пароли совпадают, то мы генерируем JWT токен и отправляем его на клиент, после чего на клиенте этот токен сохраняется либо в куках либо в localstorage. Затем этот токен прикрепляется к запросам, где необходима авторизация. Допустим мы не можем зайти на почту если не авторизованы. Мы авторизуемся – сервер генерирует токен и возвращает его обратно на клиент. Затем мы заходим на почту и отправляется запрос на полчение писем. К этому запросу в HEADER в HTTP заголовке мы добавляем authorization, и туда помещаем тот самый токен. Его мы достаем из куки или localstorage. Сервер валидирует этот токен по секретному ключу. Если он валидный, то пользователь попадает на страницу и получает все письма. В общем с помощью токена обеспечивается доступ к той или иной функции. Приступим к реализации.

Установим несколько нужных модулей.

Npm i jsonwebtoken bcrypt.

Jsonwebtoken для генерации токена. Bcrypt для того чтобы хэшировать пароль и не хранить в базе данных в открытом виде