Для начала разработки необходимо создать package.json командой npm init.

Для создания веб-сервера необходимо добавить библиотеку express.

Для того чтобы использовать ES6 Modules вместо CommonJS Modules необходимо в package.json указать строку “type”: “module”.

Чтобы начать работу с express необходимо импортировать библиотеку.

# Express.js

import express from 'express';

const app = express();

Создает express приложение. Теперь вся логика приложения находится в переменной app. Для того чтобы работать с GET запросами есть метод get().

app.get('/', (request, response) => {

  response.send('Hello world');

});

Он принимает строку с путем первым параметром, вторым функцию, которая будет выполняться при GET-запросе. Эта функция содержит два параметра. В первом лежит то, что приходит от клиента, то есть запрос, второй отвечает за ответ клиенту от сервера. Чтобы вернуть что-то пользователю у response есть метод send(). Эта функция буквально будет выполняться каждый раз, когда клиент делает GET-запрос, что означает что там может быть описана логика, например, создания (рендеринга) каких-либо статей, удаление, отображение запрошенной информации. Таким похожим образом можно описывать все routes, то есть типы запросов, что будет происходить на GET-запрос, POST-запрос, и так далее.

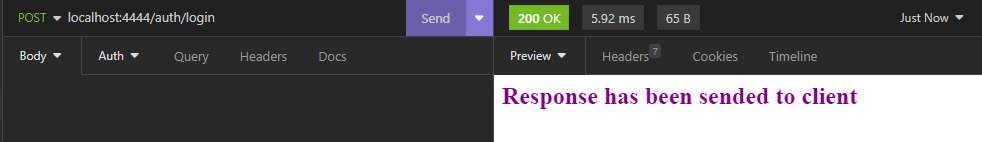
## response.send()

В качестве параметра эта функция может принимать объект Buffer, строку, в том числе с html кодом, объект javascript, или массив.

app.post('/auth/login', (request, response) => {

  response.send("<h2 style='color: purple'>Response has been sended to client</h2>");

});



## response.sendFile()

Метод send удобен для отправки строк, некоторого кода html небольшой длины, однако есть отправляемый код html довольно большой, то соответственно код приложения тоже становится громоздким. Однако гораздо лучше определять код html в отдельных файлах и затем эти файлы отправлять с помощью функции sendFile().

Следует учитывать, что в функцию sendFile необходимо передавать абсолютный путь к файлу, именно для этого с помощью \_\_dirname получаем абсолютный путь к текущему проекту и затем добавляем к нему путь к файлу в рамках текущего проекта.

## response.sendStatus()

В аргументы передается статусный код

Для того чтобы запустить веб-сервер, необходимо использовать метод listen(), в аргументы которого передается порт, на котором будет работать веб-сервер. Порт может быть абсолютно любым. Вторым параметром метод принимает функцию, в которой описывается поведение при начале работе сервера. Также там обрабатывают ошибку, если сервер не сможет запуститься.

app.listen(PORT, (error) => {

  if (error) {

    return console.log(error);

  }

  console.log(`Server is working on ${PORT}`);

});

## Библиотека nodemon

Данная библиотека необходима для того, чтобы каждый раз не перезапускать приложение для просмотра изменений на странице. Эта утилита занимается тем, что отслеживает всю файловую систему приложения, и автоматически перезапускает веб-сервер, если нужна такая необходимость, в данном случае если произошли изменения в коде. Для того чтобы начать ею пользоваться, в терминале вместо node index.js, необходимо написать nodemon index.js. Лучшей практикой будет добавить данную команду в раздел scripts в package.json. Скрипт может быть назван как угодно.

"scripts": {

    "start:dev": "nodemon index.js",

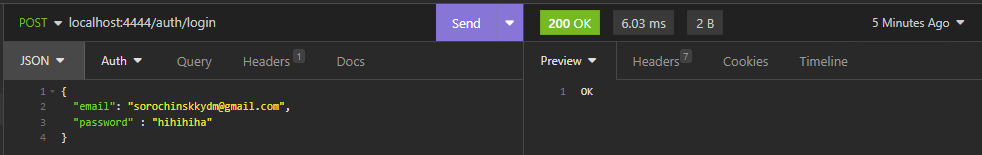
    "start": "node index.js"

  },

Теперь при сохранении кода nodemon будет самостоятельно перезапускать приложение, в то время как страницу нужно будет всего лишь перезагрузить внутри браузера.

## Работа с POST-запросами

Для того чтобы отправить какую-либо информацию в insomnia, необходимо выбрать JSON.



Данная информация будет храниться в request, но просто так получить ее не выйдет, так как express не понимает json. При попытке вывода информации будет получаться undefined. Чтобы решить эту проблему необходимо:

app.use(express.json());

## Библиотека jsonwebtoken

Данная библиотека выдает токен, который в будущем дает возможность понимания авторизован ли пользователь, имеет ли он доступ к закрытой информации или нет. Когда нужно использовать jwt? Документация советует использовать токены при авторизации, также при обмене информацией.

app.post('/auth/login', (request, response) => {

  const { email, password } = request.body;

  console.log(request.body);

  const token = jwt.sign(

    {

      email: email,

      password: password,

      fullname: 'sorochinskkydm',

    },

    'someDifficultKey',

  );

  response.send({

    token,

    status: response.status,

  });

});

При отправке POST-запроса создается токен с помощью sign(), первым аргументом является тело, то есть что будет шифроваться, вторым аргументом секретное слово, по которому будет зашифрована информация. Секретное слово может быть любой строкой. Далее этот токен возвращается обратно клиенту в методе send(). Что должно быть внутри токена? То, что необходимо зашифровать для дальнейшей работы, например, id пользователя.

## Библиотека mongoose

Данная библиотека позволяет работать с базой данных MongoDB.

mongoose

  .connect(

    'mongodb+srv://sorochinskkydm:wwwwww@cluster0.xvskprv.mongodb.net/?retryWrites=true&w=majority',

  )

  .then(() => console.log('DB is OK'))

  .catch(() => console.log('DB is not OK'));

При создании кластера mongoDB выдает ссылку для подключения. Если срабатывает ошибка, связанная с IP, то необходимо в облаке MongoDB в вкладке network access вместо IP поставить все нули, что буквально означает любой IP адрес. Ошибка должна уйти.

Далее создается отдельная папка models, где будет «шаблон» пользователя. В ней создается отдельный файл User.js, который имеет следующую структуру:

import mongoose from 'mongoose';

const UserSchema = new mongoose.Schema(

  {

    fullname: {

      type: String,

      required: true,

    },

    email: {

      type: String,

      required: true,

      unique: true,

    },

    passwordHash: {

      type: String,

      required: true,

    },

    imageUrl: String,

  },

  {

    timestamps: true,

  },

);

export default mongoose.model('User', UserSchema);

fullname, email, passwordHash, imageUrl – поля таблицы, то что находится внутри – настройки этого поля. Необходимо описать тип данных, также указать обязательным ли будет поле для заполнения. Поле “unique” говорит о том, что поле должно быть уникальным во всей базе.

## Библиотека express Validator

Перед тем как запрашивать какие-либо данные, и в дальнейшем с ними работать, данные необходимо валидировать, то есть проверить на правильность ввода. Для этого необходима библиотека express-validator. В файл необходимо импортировать body из express-validator, с помощью которого можно будет проверять есть ли в теле запроса какая-либо информация, и ее валидировать.

import { body } from 'express-validator';

export const registerValidator = [

  body('email', 'Неверный формат почты').isEmail(),

  body('password', 'Пароль должен содержать минимум 5 символов').isLength({ min: 5 }),

  body('fullname', 'Длина поля должна составлять минимум 2 символа').isLength({ min: 2 }),

  body('picture', 'Неверная ссылка на изображение').optional().isURL(),

];

После импорта необходимо указать эту валидацию в параметрах маршрута.

app.post('/auth/register', registerValidator, (request, response) => {});

Далее необходима функция, которая будет проверять, есть ли ошибки при валидации данных.

import { validationResult } from 'express-validator';

//Register route

app.post('/auth/register', registerValidator, (request, response) => {

  const errors = validationResult(request);

  if (!errors.isEmpty()) {

    return response.status(400).json(errors.array());

  }

  response.json({

    success: true,

  });

});

Для того чтобы создать пользователя через маршрут auth/register необходимо импортировать модель пользователя в файл, создать документ пользователя. Документом называется одна сущность пользователя, то есть отдельный пользователь. Хранилищем для документов, в данном случае пользователей выступает коллекция, то есть коллекция. В итоге имеем коллекцию пользователей, которая содержит в себе документы отдельных пользователей.

app.post('/auth/register', registerValidator, (request, response) => {

  const errors = validationResult(request);

  if (!errors.isEmpty()) {

    return response.status(400).json(errors.array());

  }

  const userDocument = new UserModel({

    email: request.body.email,

    passwordHash: request.body.password,

    fullname: request.body.fullname,

    pictureUrl: request.body.picture,

  });

  response.json({

    success: true,

  });

});

Данные о пользователе хранятся в объекте request.body. Но есть один нюанс, связанный с паролем пользователя. Нельзя просто так передавать пароль, его необходимо зашифровать перед передачей на стороне бекенда. Для этого необходима библиотека bcrypt.

## Библиотека bcrypt

//Хэширование пароля

  const password = request.body.password;

  const salt = await bcrypt.genSalt(10);

  const passwordHash = await bcrypt.hash(password, salt);

Соль – это некий алгоритм шифрования. По сути, это случайная строка данных, которая хэшируется вместе с паролем, для сохранения уникальности результата хэширования.

  //Создание отдельного пользователя в коллекции

  const userDocument = new UserModel({

    email: request.body.email,

    fullname: request.body.fullname,

    pictureUrl: request.body.picture,

    passwordHash,

  });

Теперь вместо открытого пароля на сервер передается зашифрованный пароль.

Чтобы закончить создание пользователя, необходимо сохранить документ.

Далее в строке подключения к базе данных между “/” и “?” необходимо указать название коллекции.

## Обработка ошибок

Все блоки, маршруты необходимо оборачивать в try – catch для обработки потенциальных ошибок.

Для того чтобы убрать какое-то поле из ответа, необходимо вытащить это поле с помощью деструктуризации, затем раскрыть с помощью spread оператора весь оставшийся объект.

## Авторизация

Алгоритм авторизации: авторизовать пользователя – означает найти его в базе данных.

В авторизации есть несколько шагов:

1. Поиск пользователя по какому-либо критерию (в данном случае по email), обработка ошибки в случае, если пользователь не найден
2. Сравнение пароля, введенного пользователем, с тем, что захэширован и лежит в базе данных, обработка ошибки если пароли не схожи.
3. Создание нового токена, в котором лежит \_id пользователя.

//Auth route

app.post('/auth/login', async (request, response) => {

  try {

    //Выборка пользователя по email

    const user = await UserModel.findOne({ email: request.body.email });

    if (!user) {

      return response.status(404).json({

        message: 'Пользователь найден',

      });

    }

    //Сравнение пароля пользователя и хэшированного пароля

    const isValidPassword = await bcrypt.compare(request.body.password, user.\_doc.passwordHash);

    if (!isValidPassword) {

      return response.status(404).json({

        message: 'Неверный логин или пароль',

      });

    }

    //Создание нового токена при авторизации

    const token = jwt.sign(

      {

        \_id: user.\_id,

      },

      'someDifficultKey',

      {

        expiresIn: '30d',

      },

    );

    const { passwordHash, ...userData } = user.\_doc;

    response.json({

      message: 'success authorize',

      ...userData,

      token,

    });

  } catch (error) {

    console.log(error);

    response.status(500).json({

      message: 'Не удалось авторизоваться',

    });

  }

});

## Создание функции-посредника (middleware)

Функции промежуточной обработки (middleware) - это функции, имеющие доступ к [объекту запроса](https://expressjs.com/ru/4x/api.html#req) (req), [объекту ответа](https://expressjs.com/ru/4x/api.html#res) (res) и к следующей функции промежуточной обработки в цикле “запрос-ответ” приложения. Следующая функция промежуточной обработки, как правило, обозначается переменной next. Проще говоря, эта функция необходима как промежуточный этап к выполнению маршрута. Такая функция вставляется в маршрут, и будет выполняться каждый раз при запросе. Имеет доступ к response, request, и функции next, которая позволяет выполняться коду дальше и вернуть ответ. Если не использовать эту функцию, ответ будет очень долго уходить.

//GetMe route

app.get('/auth/me', checkAuth, async (request, response) => {

Созданная функция будет проверять авторизован ли пользователь для доступа к маршруту:

import jwt from 'jsonwebtoken';

export default (request, response, next) => {

  //Получение токена из authorization

  const token = (request.headers.authorization || '').replace(/Bearer\s?/, '');

  if (token) {

    try {

      //Расшифровка токена

      const decodedToken = jwt.verify(token, 'someDifficultKey');

      //Добавление в request

      request.userId = decodedToken.\_id;

      next();

    } catch (error) {

      return response.status(403).json({

        message: 'Ошибка токена',

      });

    }

  } else {

    return response.status(403).json({

      message: 'Нет доступа к маршруту',

    });

  }

};

Bearer token – токен предъявителя, соответственно кто его предъявит – тот и является авторизованным пользователем. Так как токен доступен на клиенте, отправляется запрос на сервер, где в объекте в заголовках, то есть request.headers.authorization лежит этот токен.

//GetMe route

app.get('/auth/me', checkAuth, async (request, response) => {

  try {

    //Выборка пользователя по id из БД

    const user = await UserModel.findById(request.userId);

    //Если пользователь не найден

    if (!user) {

      return response.status(404).json({

        message: 'Пользователь не найден',

      });

    }

    //Если найден, возвращает данные из документа о пользователе

    const { passwordHash, ...userData } = user.\_doc;

    response.json({

      message: 'success',

      ...userData,

    });

  } catch (error) {

    console.log(error);

    response.status(500).json({

      message: 'something went wrong, try again later',

    });

  }

});

userId появился в response после того, как в checkAuth был расшифрован токен, в котором лежал id пользователя, предъявителя токена.

## CRUD операции

//CRUD routes

app.get('/posts', PostController.getAllController);

// app.get('/posts/:id', PostController.getOneController);

app.post('/posts', checkAuth, createValidator, PostController.createPostController);

// app.delete('/posts', createValidator, PostController.removePostController);

// app.patch('/posts', createValidator, PostController.updatePostController);

Создаются маршруты для всех crud операций – создания, чтения, обновления, удаления статей. Важно правильно указывать пути маршрутизации, плохой практикой является указывать /posts/create, /posts/delete etc.. Поэтому маршруты должны выглядеть следующим образом: на каждую операцию один и тот же маршрут, то есть разный http протокол, а там где необходимо получать конкретную статью, передается динамический параметр, который может быть назван как угодно. Прежде всего создается валидация полей для создания статей:

export const createValidator = [

  body('title', 'Введите заголовок статьи').isLength({ min: 3 }).isString(),

  body('text', 'Введите текст статьи').isLength({ min: 10 }).isString(),

  body('tags', 'Неверный формат тэгов (укажите массив)').optional().isArray(),

  body('imageUrl', 'Неверная ссылка на изображение').optional().isString(),

];

Затем необходимо создать модель, то есть шаблон статьи в папке models, которая имеет следующую структуру:

import mongoose from 'mongoose';

const postSchema = new mongoose.Schema(

  {

    title: {

      type: String,

      required: true,

    },

    text: {

      type: String,

      required: true,

    },

    tags: {

      type: Array,

      default: [],

    },

    viewsCount: {

      type: Number,

      default: 0,

    },

    user: {

      type: mongoose.Schema.Types.ObjectId,

      ref: 'User',

      required: true,

    },

    imageUrl: String,

  },

  {

    timestamps: true,

  },

);

export default mongoose.model('Post', postSchema);

Важный момент, что в поле user (в данном случае автор статьи, кто ее выложил) есть несколько замечаний: поле type – туда пробрасывается id пользователя, но не просто так, а специальным типом из mongoose, потому что id внутри этой БД хранится в формате ObjectId(id). Следующим полем создается ссылка на БД пользователей, то есть связь между ними. После этого создается новый контроллер (они появились после того как был проведен рефакторинг index.js, и перенос функций-маршрутов в отдельные файлы-контроллеры). Для создания статей необходимо использовать модель, то есть ранее созданный для БД шаблон и сделать новый документ, который будет хранить то, что передает клиент из объекта request.body. Затем необходимо сохранить документ, вернуть ответ клиенту и обработать ошибки.

export const createPostController = async (request, response) => {

  try {

    const doc = new PostModel({

      title: request.body.title,

      text: request.body.text,

      tags: request.body.tags,

      imageUrl: request.body.imageUrl,

      user: request.userId,

    });

    const post = await doc.save();

    response.json(post);

  } catch (error) {

    console.log(error);

    response.status(500).json({

      message: 'Не удалось создать статью',

    });

  }

};

Чтобы обработать маршрут на получение всех статей, достаточно сделать простую выборку из уже существующих документов, и вернуть их в ответе клиенту в виде массива статей, также обработать ошибки.

export const getAllController = async (request, response) => {

  try {

    const posts = await PostModel.find().populate(‘user’).exec();

    response.status(200).json(posts);

  } catch (error) {

    console.log(error);

    response.status(500).json({

      message: 'Не удалось получить статьи',

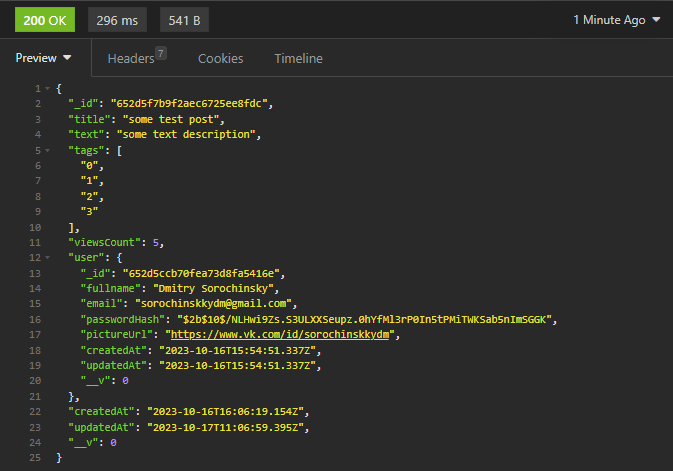
    });

  }

};

Метод populate() позволяет ссылаться на документы в других коллекциях. При создании схемы было указано, что в поле user есть ref. Все это нужно, для того чтобы в ответе от сервера отражался не просто id пользователя, а его полная информация.

Поле user теперь хранит не id, а полную информацию из другой коллекции, которую подтягивает за счет связи.



Для реализации получения одной статьи по ее id, необходимо:

export const getOneController = async (request, response) => {

  try {

    const postId = request.params.id;

    const onePost = await PostModel.findOneAndUpdate(

      { \_id: postId },

      { $inc: { viewsCount: 1 } },

      { new: true },

    )

      .populate('user')

      .exec();

    if (!onePost) {

      return response.status(404).json({

        message: 'Не удалось найти статью',

      });

    }

    response.json(onePost);

  } catch (error) {

    console.log(error);

    response.status(500).json({

      message: 'Не удалось получить статью',

    });

  }

};

Чтобы получить динамический параметр, который указывался в пути маршрута, необходимо обратиться к объекту request.params.{название параметра, который был указан в пути }. Для того чтобы сделать выборку одной статьи можно было использовать model.findOne(), но так как статьи имеют счетчик просмотров, необходимо и изменять документ вместе с выборкой. Метод model.findOneAndUpdate() принимает несколько аргументов: фильтр, по которому проходит выборка, изменение (чтобы инкрементировать что-то, есть запись: $inc: {что-инкрементировать: количество}), также сохранить уже измененный документ.

export const removePostController = async (request, response) => {

  try {

    const postId = request.params.id;

    const removedPost = await PostModel.findOneAndDelete({ \_id: postId });

    if (!removedPost) {

      return response.status(404).json({

        message: 'Не удалось найти статью, вероятно ее уже не существует',

      });

    }

    response.json({

      message: 'deleted successfully',

      removedPost: removedPost.\_id,

    });

  } catch (error) {

    console.log(error);

    response.status(500).json({

      message: 'Не удалось удалить статью',

    });

  }

};

export const updatePostController = async (request, response) => {

  try {

    const postId = request.params.id;

    const updatedPost = PostModel.findOneAndUpdate(

      { \_id: postId },

      {

        title: request.body.title,

        text: request.body.text,

        tags: request.body.tags,

        imageUrl: request.body.imageUrl,

      },

      {

        new: true,

      },

    )

      .populate('user')

      .exec();

    if (!updatedPost) {

      return response.status(404).json({

        message: 'Не удалось найти статью',

      });

    }

    response.json({

      message: 'successfully updated',

      updatedPost: updatedPost.\_id,

    });

  } catch (error) {

    console.log(error);

    response.status(500).json({

      message: 'Не удалось изменить статью',

    });

  }

};