



# Инструкция для первого файла

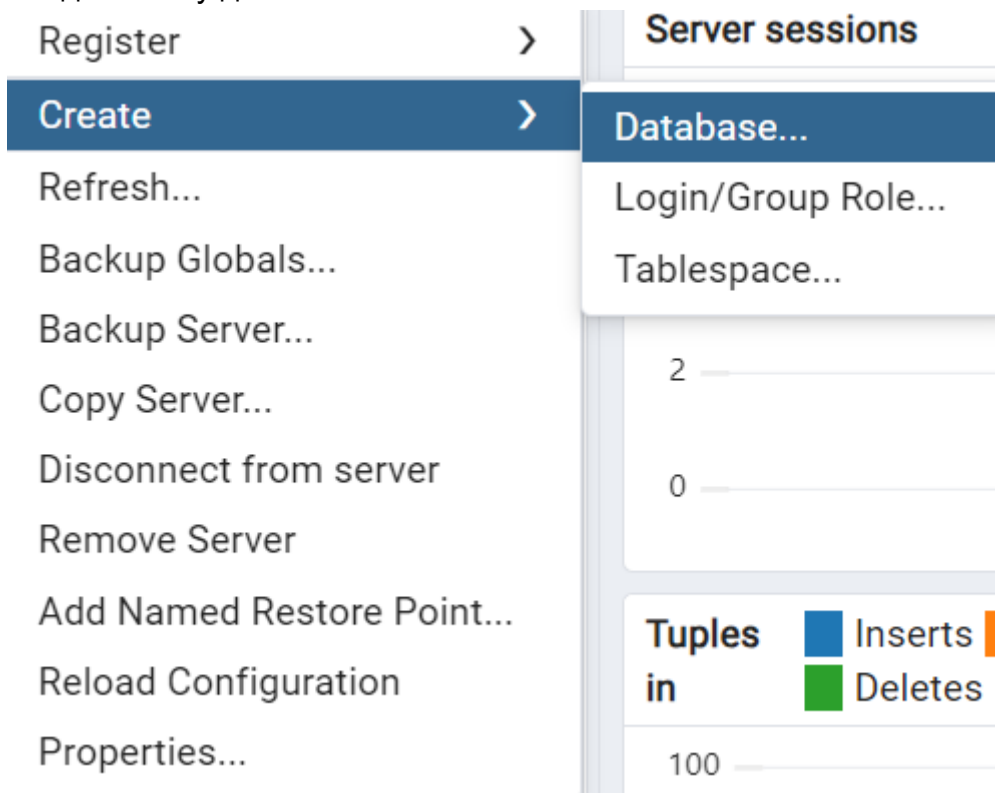
Для начала нужно установить соединение с бд, для этого нужно открыть pgAdmin4, его можно найти в поиске приложений.

1. Входим в установленную бд

✓  Servers (2)

>  PostgreSQL 16

2. Создаём базу данных





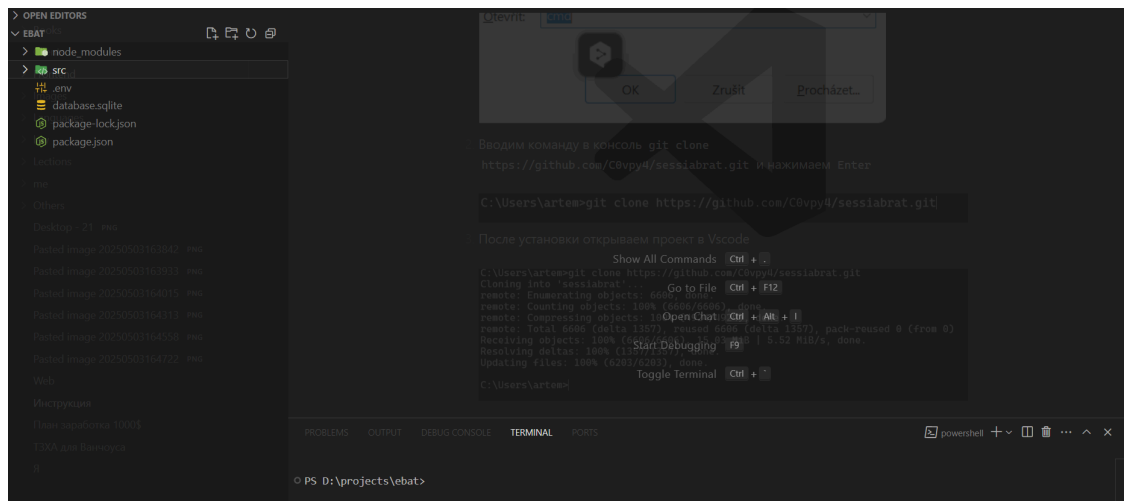
```
`git clone https://github.com/w1n17/appointments.git
```

```
C:\Users\artem>git clone https://github.com/C0vpy4/sessiabrat.git
```

### 3. После установки открываем проект в Vscode

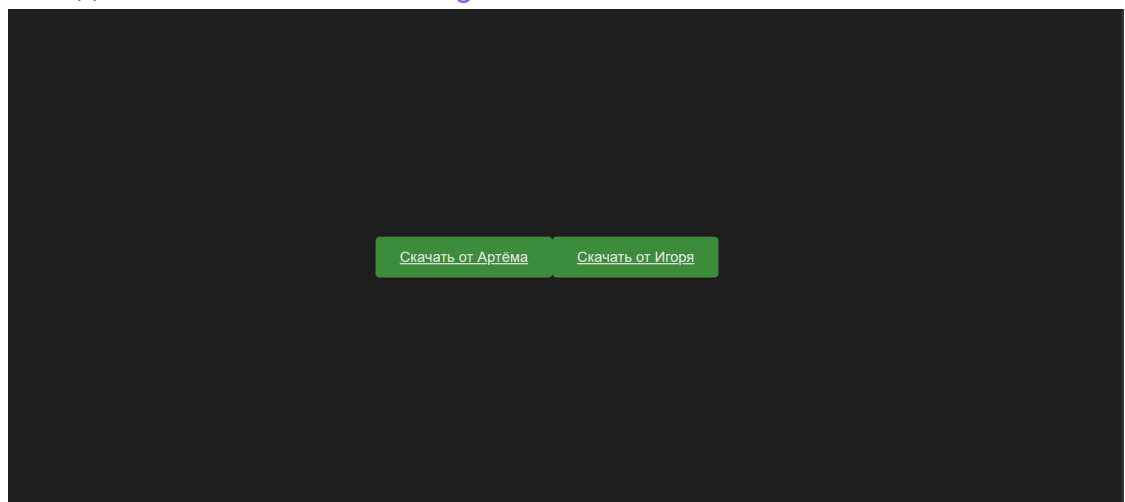
```
C:\Users\artem>git clone https://github.com/C0vpy4/sessiabrat.git
Cloning into 'sessiabrat'...
remote: Enumerating objects: 6606, done.
remote: Counting objects: 100% (6606/6606), done.
remote: Compressing objects: 100% (4924/4924), done.
remote: Total 6606 (delta 1357), reused 6606 (delta 1357), pack-reused 0 (from 0)
Receiving objects: 100% (6606/6606), 15.03 MiB | 5.52 MiB/s, done.
Resolving deltas: 100% (1357/1357), done.
Updating files: 100% (6203/6203), done.

C:\Users\artem>
```



## 2. Скачиваем при помощи сайта [Download Page](#)

### 1. Заходим на сайт [Download Page](#)



### 2. Нажимаем на кнопку Скачать(выберите подходящий вариант)

### 3. После установки, распаковываем архив и открываем его в VScode

### 4. Дальше я буду объяснять по проекту от Артёма

## 4. Переходим к настройке самого проекта

## 1. Настраиваем файл .env

```
1 DB_HOST=localhost
2 DB_PORT=5433
3 DB_NAME=postgres
4 DB_USER=postgres
5 DB_PASSWORD=123456
6
7
```

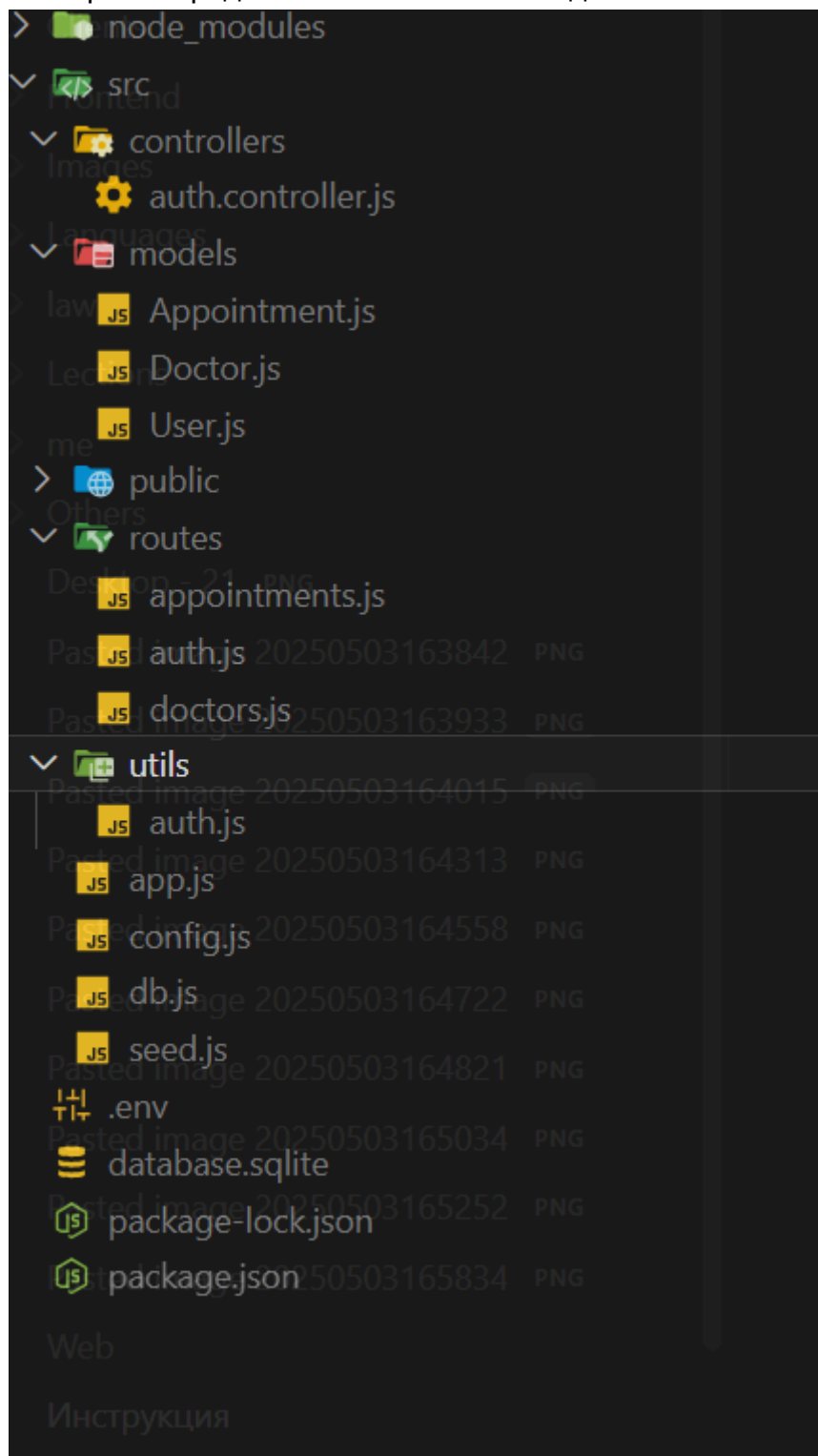
1. DB\_HOST оставляем таким каким он есть
2. DB\_PORT - выставляй 5432
3. DB\_NAME - здесь пиши название своей бдхи
4. DB\_USER - пиши пользователя, по умолчанию postgres
5. DB\_PASSWORD - пароль от нашей бдхи
6. В СЛУЧАЕ ОШИБКИ ИСПРАВЬТЕ ФАЙЛ И ПОДСТАВЬТЕ ТЕЖЕ ЗНАЧЕНИЯ(config.js)

```
1 const path = require("path");
2
3 // Конфигурация приложения, которую можно легко изменить под требования билета
4 module.exports = {
5   // Настройки базы данных
6   database: {
7     dialect: process.env.DB_DIALECT || "sqlite",
8     storage:
9       process.env.DB_STORAGE || path.join(__dirname, "..", "database.sqlite"),
10    host: process.env.DB_HOST || "localhost",
11    port: process.env.DB_PORT || 5433,
12    username: process.env.DB_USER || "postgres",
13    password: process.env.DB_PASSWORD || "postgres",
14    database: process.env.DB_NAME || "appointment_service",
15    logging: console.log,
16  },
17
18  // Настройки приложения
19  app: {
20    port: process.env.PORT || 3000,
21    timeSlotInterval: 30, // Интервал слотов в минутах
22    workingHours: {
23      start: 10, // Начало рабочего дня (часы)
24      end: 20, // Конец рабочего дня (часы)
25    },
26  },
27 };
28
```

2. После редактирования файла, нужно установить все зависимости, в случае если у вас нет папки node\_modules

## 5. Объяснение самого проекта

1. сам проект представляет собой взаимодействие нескольких файлов



2. Начнём с папки models, в неё мы прописываем наши таблицы, например User

```
const { DataTypes } = require("sequelize"); -  
импорт модуля НЕ ТРОГАЕМ  
const sequelize = require("../db"); - импорт модуля  
НЕ ТРОГАЕМ  
const User = sequelize.define("User", { - Создание  
нашей таблицы User  
  email: { type: DataTypes.STRING, unique: true },  
- Одно из полей таблицы  
  password: DataTypes.STRING,  
  name: DataTypes.STRING,  
  phone: DataTypes.STRING,  
});  
module.exports = User; - Экспортируем нашу таблицу
```

```
const { DataTypes } = require("sequelize"); 1.6M (gzipped: 282.9K)  
const sequelize = require("../db");
```

✪Cody

```
const Doctor = sequelize.define("Doctor", {  
  firstName: DataTypes.STRING,  
  lastName: DataTypes.STRING,  
  middleName: DataTypes.STRING,  
  specialization: DataTypes.STRING,  
  availableSlots: {  
    type: DataTypes.JSON,  
    defaultValue: [],  
  },  
});
```

```
module.exports = Doctor;
```

```
const { DataTypes } = require("sequelize"); 1.6M (gzipped: 282.9K)  
const sequelize = require("../db");  
const User = require("../User");  
const Doctor = require("../Doctor");  
const Appointment = sequelize.define("Appointment", {  
  slot: DataTypes.STRING,  
});  
User.hasMany(Appointment);  
Appointment.belongsTo(User);  
Doctor.hasMany(Appointment);  
Appointment.belongsTo(Doctor);  
module.exports = Appointment;
```

Для всего остального:

# Руководство по изменению файла `appointments.js` на любую другую тему

Это руководство поможет вам изменить файл `appointments.js`, который содержит API-роутеры для управления записями на прием к врачу, на любую другую тему. Мы рассмотрим, как изменить существующие роутеры и добавить новые для соответствия новой теме.

## 1. Определение новой темы

Перед началом изменений, определите новую тему и функциональность, которые вы хотите реализовать. Например, если вы хотите изменить файл для управления задачами, то новая тема будет "управление задачами".

## 2. Изменение импортов

Измените импорты, чтобы они соответствовали новым моделям и middleware, которые вы будете использовать.

```
const express = require("express");
const router = express.Router();
const Task = require("../models/Task");
const User = require("../models/User");
const { authenticate } = require("../utils/auth");
```

## 3. Изменение роутеров

Измените существующие роутеры или создайте новые, чтобы они соответствовали новой теме.

### Создание новой задачи

```
router.post("/", authenticate, async (req, res) => {
  try {
    const { title, description, dueDate } = req.body;
    if (!title || !description || !dueDate) {
      return res.status(400).json({ error: "Название, описание и дата выполнения задачи обязательны" });
    }

    const task = await Task.create({
      UserId: req.user.userId,
      title: title,
      description: description,
      dueDate: dueDate,
    });
  }
});
```

```

    res.status(201).json({
      message: "Задача успешно создана",
      task: {
        id: task.id,
        title: task.title,
        description: task.description,
        dueDate: task.dueDate,
      },
    });
  } catch (error) {
    console.error("Ошибка создания задачи:", error);
    res.status(500).json({ error: "Внутренняя ошибка сервера" });
  }
});

```

## Получение списка задач текущего пользователя

```

router.get("/my", authenticate, async (req, res) => {
  try {
    const tasks = await Task.findAll({
      where: {
        UserId: req.user.userId,
      },
    });

    res.status(200).json(tasks);
  } catch (error) {
    console.error("Ошибка получения списка задач:", error);
    res.status(500).json({ error: "Внутренняя ошибка сервера" });
  }
});

```

## Обновление задачи

```

router.put("/:id", authenticate, async (req, res) => {
  try {
    const taskId = req.params.id;
    const { title, description, dueDate, completed } = req.body;

    const task = await Task.findByPk(taskId, {
      where: {
        UserId: req.user.userId,
      },
    });

    if (!task) {

```



```

        return res.status(404).json({ error: "Задача не найдена или не
принадлежит текущему пользователю" });
    }

    await task.update({ title, description, dueDate, completed });

    res.status(200).json({
        message: "Задача успешно обновлена",
        task: {
            id: task.id,
            title: task.title,
            description: task.description,
            dueDate: task.dueDate,
            completed: task.completed,
        },
    });
} catch (error) {
    console.error("Ошибка обновления задачи:", error);
    res.status(500).json({ error: "Внутренняя ошибка сервера" });
}
});

```

## Удаление задачи

```

router.delete("/:id", authenticate, async (req, res) => {
    try {
        const taskId = req.params.id;

        const task = await Task.findByPk(taskId, {
            where: {
                UserId: req.user.userId,
            },
        });

        if (!task) {
            return res.status(404).json({ error: "Задача не найдена или не
принадлежит текущему пользователю" });
        }

        await task.destroy();

        res.status(200).json({ message: "Задача успешно удалена" });
    } catch (error) {
        console.error("Ошибка удаления задачи:", error);
        res.status(500).json({ error: "Внутренняя ошибка сервера" });
    }
});

```

## 4. Экспорт роутера

Не забудьте экспортировать роутер в конце файла.

```
module.exports = router;
```

## 5. Примеры запросов

### Создание задачи

```
curl -X POST -H "Content-Type: application/json" -H "Authorization: Bearer <token>" -d '{"title": "Новая задача", "description": "Описание задачи", "dueDate": "2024-05-15"}' http://localhost:3000/tasks/
```

### Получение списка задач

```
curl -H "Authorization: Bearer <token>" http://localhost:3000/tasks/my
```

### Обновление задачи

```
curl -X PUT -H "Content-Type: application/json" -H "Authorization: Bearer <token>" -d '{"title": "Обновленная задача", "description": "Новое описание задачи", "dueDate": "2024-05-20", "completed": true}' http://localhost:3000/tasks/1
```

### Удаление задачи

```
curl -X DELETE -H "Authorization: Bearer <token>" http://localhost:3000/tasks/1
```

## Руководство по файлу `seed.js`

Этот файл предназначен для инициализации или заполнения базы данных тестовыми данными. Это полезно для настройки среды разработки или тестирования. В этом файле используется библиотека Sequelize для взаимодействия с базой данных.

### 1. Импорты и Настройка

```
const sequelize = require("./db");
const User = require("./models/User");
const Doctor = require("./models/Doctor");
const Appointment = require("./models/Appointment");
```

- `sequelize` : Экземпляр Sequelize, который используется для подключения к базе данных.
- `User` : Модель пользователя.
- `Doctor` : Модель доктора.
- `Appointment` : Модель записи на прием.

## 2. Генерация временных слотов

```
function generateTimeSlots() {
  const slots = [];
  for (let hour = 10; hour < 20; hour++) {
    const formattedHour = hour.toString().padStart(2, "0");
    slots.push(`${formattedHour}:00-${formattedHour}:30`);
    slots.push(
      `${formattedHour}:30-${(hour + 1).toString().padStart(2, "0")}:00`
    );
  }
  return slots;
}
```

- `generateTimeSlots` : Функция для генерации временных слотов с 10:00 до 20:00 с интервалом 30 минут. Используется для заполнения доступных слотов у докторов.

## 3. Функция заполнения базы данных

```
async function seedDatabase() {
  try {
    await sequelize.sync({ force: true }); // Синхронизация моделей с базой
    данных
    console.log("База данных синхронизирована");
  }
}
```

- `sequelize.sync({ force: true })` : Синхронизирует модели с базой данных. Параметр `force: true` удаляет существующие таблицы перед созданием новых. Используйте это с осторожностью, так как это приведет к потере данных.

## 4. Создание тестовых пользователей

```
const users = await User.bulkCreate([
  {
    email: "user1@example.com",
    password: "123456",
    name: "Иван Иванов",
    phone: "+7 (999) 123-45-67",
  },
  {
```

```

        email: "user2@example.com",
        password: "123456",
        name: "Петр Петров",
        phone: "+7 (999) 765-43-21",
    },
]);
console.log(`Создано ${users.length} пользователей`);

```

- `User.bulkCreate` : Метод для bulk-создания нескольких пользователей. Возвращает массив созданных пользователей.

## 5. Создание тестовых докторов

```

const doctors = await Doctor.bulkCreate([
  {
    firstName: "Александр",
    lastName: "Смирнов",
    middleName: "Иванович",
    specialization: "Терапевт",
    availableSlots: generateTimeSlots(),
  },
  {
    firstName: "Елена",
    lastName: "Кузнецова",
    middleName: "Петровна",
    specialization: "Кардиолог",
    availableSlots: generateTimeSlots(),
  },
  {
    firstName: "Дмитрий",
    lastName: "Соколов",
    middleName: "",
    specialization: "Невролог",
    availableSlots: generateTimeSlots(),
  },
]);
console.log(`Создано ${doctors.length} докторов`);

```

- `Doctor.bulkCreate` : Метод для bulk-создания нескольких докторов. Возвращает массив созданных докторов.

## 6. Создание тестовых записей на прием

```

const appointments = await Appointment.bulkCreate([
  { UserId: 1, DoctorId: 1, slot: "10:00-10:30" },
  { UserId: 2, DoctorId: 2, slot: "14:00-14:30" },
]);

```

```
]);  
console.log(`Создано ${appointments.length} записей на прием`);
```

- `Appointment.bulkCreate` : Метод для bulk-создания нескольких записей на прием. Возвращает массив созданных записей.

## 7. Обновление доступных слотов у докторов

```
const doctor1 = await Doctor.findByPk(1);  
doctor1.availableSlots = doctor1.availableSlots.filter(  
  (slot) => slot !== "10:00-10:30"  
);  
await doctor1.save();  
  
const doctor2 = await Doctor.findByPk(2);  
doctor2.availableSlots = doctor2.availableSlots.filter(  
  (slot) => slot !== "14:00-14:30"  
);  
await doctor2.save();
```

- `Doctor.findByPk` : Метод для поиска доктора по его ID.
- `filter` : Метод массива для фильтрации слотов, чтобы удалить занятые.
- `save` : Метод для сохранения изменений доктора в базу данных.

## 8. Завершение и закрытие соединения

```
console.log("Заполнение базы данных завершено успешно");  
} catch (error) {  
  console.error("Ошибка при заполнении базы данных:", error);  
} finally {  
  await sequelize.close(); // Закрываем соединение с базой данных  
}  
}  
  
seedDatabase();
```

- `finally` : Блок, который выполняется всегда, независимо от успешного завершения или возникновения ошибки.
- `sequelize.close()` : Закрывает соединение с базой данных.

## Использование файла

1. **Установите зависимости:** Убедитесь, что у вас установлены все необходимые зависимости, такие как `sequelize`, `sqlite3` и другие, указанные в вашем файле `package.json`.

2. **Запустите файл:** Вы можете запустить файл `seed.js` с помощью Node.js:

```
node seed.js
```

3. **Проверьте базу данных:** После запуска файла проверьте вашу базу данных, чтобы убедиться, что все тестовые данные были успешно добавлены.

## Руководство по файлу `app.js`

Этот файл `app.js` является основным файлом приложения Express.js. Он отвечает за настройку сервера, подключение middleware, маршрутизацию и взаимодействие с базой данных. Вот полное руководство по этому файлу:

### 1. Импорты и Настройка

```
const express = require("express");
const path = require("path");
const bodyParser = require("body-parser");
const sequelize = require("./db");
const config = require("./config");

const app = express();
```

- `express` : Основной фреймворк для создания сервера.
- `path` : Модуль Node.js для работы с путями файлов.
- `body-parser` : Middleware для обработки тел запросов JSON и URL-encoded.
- `sequelize` : Настройка подключения к базе данных.
- `config` : Настройки приложения (порт, ключи и т.д.).

### 2. Middleware

#### Raw Parser для JSON

```
app.use((req, res, next) => {
  if (
    req.method === "POST" &&
    !req.is("application/json") &&
    req.headers["content-length"]
  ) {
    let data = "";
    req.setEncoding("utf8");
    req.on("data", (chunk) => {
      data += chunk;
    });
    req.on("end", () => {
```

```

    try {
      req.body = JSON.parse(data);
      console.log("Принудительно обработан JSON:", req.body);
      next();
    } catch (e) {
      console.error("Ошибка парсинга JSON:", e);
      next();
    }
  });
} else {
  next();
}
});

```

- **Цель:** Обрабатывает тело POST-запросов как JSON, даже если заголовок Content-Type указан неверно.
- **Логика:** Считывает данные из потока, пытается распарсить как JSON и сохраняет в req.body.

## Стандартные Парсеры

```

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

```

- `bodyParser.json()` : Парсит JSON-тело запроса.
- `bodyParser.urlencoded()` : Парсит URL-encoded тело запроса (обычно используется в формах).

## CORS Middleware

```

app.use((req, res, next) => {
  res.header("Access-Control-Allow-Origin", "*");
  res.header(
    "Access-Control-Allow-Headers",
    "Origin, X-Requested-With, Content-Type, Accept, Authorization"
  );
  res.header("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE, OPTIONS");

  if (req.method === "OPTIONS") {
    return res.sendStatus(200);
  }

  next();
});

```

- **Цель:** Управление кросс-доменными запросами (CORS).
- **Логика:** Добавляет заголовки для разрешения запросов с любого источника ( \* ).

## Middleware для Отладки

```
app.use((req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  console.log("Headers:", req.headers);
  console.log("Body:", req.body);
  next();
});
```

- **Цель:** Логирование входящих запросов для отладки.
- **Логика:** Выводит метод, URL, заголовки и тело запроса в консоль.

## Статические Файлы

```
app.use(express.static(path.join(__dirname, "public")));
```

- **Цель:** Обслуживание статических файлов (CSS, JavaScript, изображения и т.д.).
- **Логика:** Указывает папку `public` как источник статических файлов.

## 3. Роутеры

```
const authRouter = require("./routes/auth");
const doctorsRouter = require("./routes/doctors");
const appointmentsRouter = require("./routes/appointments");

app.use("/auth", authRouter);
app.use("/doctors", doctorsRouter);
app.use("/appointments", appointmentsRouter);
```

- **Цель:** Подключение маршрутов для аутентификации, докторов и записей на приём.
- **Логика:** Роутеры обрабатывают запросы к соответствующим endpoint'ам ( `/auth` , `/doctors` , `/appointments` ).

## 4. Корневой Роут

```
app.get("/", (req, res) => {
  res.sendFile(path.join(__dirname, "public", "index.html"));
});
```

- **Цель:** Отправка главной HTML-страницы при запросе корневого URL.



- **Логика:** Использует `sendFile` для отправки файла `index.html` из папки `public`.

## 5. Обработка Ошибок

```
app.use((err, req, res, next) => {
  console.error("Ошибка:", err);
  res.status(500).json({
    error: "Внутренняя ошибка сервера",
    message: err.message,
  });
});
```

- **Цель:** Обработка ошибок на сервере.
- **Логика:** Логирует ошибку и отправляет клиенту статус 500 и информацию об ошибке.

## 6. Настройка Базы Данных и Запуск Сервера

```
const PORT = config.app.port;

// Импортируем все модели для уверенности, что они загружены
const User = require("./models/User");
const Doctor = require("./models/Doctor");
const Appointment = require("./models/Appointment");

// Синхронизируем модели с базой данных
sequelize
  .sync({ force: false }) // force: true удалит и пересоздаст таблицы
  (осторожно!)
  .then(() => {
    console.log("Модели синхронизированы с базой данных");

    app.listen(PORT, () => {
      console.log(`Сервер запущен на порту ${PORT}`);
    });
  })
  .catch((err) => {
    console.error("Ошибка синхронизации моделей с базой данных:", err);
  });
```

- **PORT** : Порт, на котором будет запущен сервер (берется из конфигурации).
- **sequelize.sync()** : Синхронизирует модели с базой данных. Параметр `force: false` означает, что таблицы не будут удалены и пересозданы (в отличие от `force: true`).
- **app.listen()** : Запускает сервер на указанном порту.

## 7. Экспорт приложения

```
module.exports = app;
```

- **Цель:** Экспортирует экземпляр Express для использования в других файлах.

## Использование файла

**Запустите сервер:** Выполните следующую команду, чтобы запустить сервер:

```
npm run dev
```

## Руководство по файлу `auth.js`

Этот файл `auth.js` содержит API-роутеры для аутентификации и управления пользователями. Он предоставляет функциональность для входа/регистрации пользователя, получения текущего пользователя и выхода из системы. Вот полное руководство по этому файлу:

### 1. Импорты и Настройка

```
const express = require("express");
const router = express.Router();
const User = require("../models/User");
const { generateToken, authenticate, sessions } = require("../utils/auth");
const bcrypt = require("bcrypt");
const getRawBody = require("raw-body");
```

- `express` : Фреймворк для Node.js, используемый для обработки HTTP-запросов.
- `express.Router` : Объект, который используется для создания роутера.
- `User` : Модель БД для пользователя.
- `generateToken` : Функция для генерации токена аутентификации.
- `authenticate` : Middleware для проверки аутентификации пользователя.
- `sessions` : Объект для хранения сессий пользователей.
- `bcrypt` : Библиотека для хеширования паролей.
- `getRawBody` : Middleware для обработки сырых тел запросов.

### 2. Роутеры

#### Вход/Регистрация пользователя

```
router.post("/login", async (req, res) => {
  try {
```

```
console.log("Начало обработки запроса на вход/регистрацию");

if (!req.body) {
  console.log("req.body отсутствует");
  return res.status(400).json({ error: "Отсутствует тело запроса" });
}

const email = req.body.email;
const password = req.body.password;
const name = req.body.name;
const phone = req.body.phone;

if (!email || !password) {
  console.log("Отсутствует email или пароль");
  return res.status(400).json({ error: "Email и пароль обязательны" });
}

let user = await User.findOne({ where: { email } });

if (!user) {
  console.log(`Пользователь ${email} не найден, создаем нового`);
  const hashedPassword = await bcrypt.hash(password, 10);
  user = await User.create({
    email,
    password: hashedPassword,
    name: name || "",
    phone: phone || "",
  });
}

const checkUser = await User.findOne({ where: { email } });
if (checkUser) {
  console.log(`Проверка: пользователь ${email} найден в базе`);
} else {
  console.log(`Проверка: пользователь ${email} НЕ найден в базе после создания!`);
}

if (name || phone) {
  await user.update({
    name: name || user.name,
    phone: phone || user.phone,
  });
}
```

```

    }

    const token = generateToken(user);
    res.status(200).json({
      message: "Вход выполнен успешно",
      token,
      user: {
        id: user.id,
        email: user.email,
        name: user.name,
        phone: user.phone,
      },
    });
  } catch (error) {
    console.error("Ошибка входа:", error);
    res.status(500).json({ error: "Внутренняя ошибка сервера", details:
error.message });
  }
});

```

- **/login POST** : Роутер для входа/регистрации пользователя.
- **Проверка тела запроса**: Убедитесь, что тело запроса не пустое и содержит `email` и `password`.
- **Поиск пользователя**: Ищем пользователя по электронной почте.
- **Создание пользователя**: Если пользователя нет, создаем нового с хешированным паролем.
- **Проверка пароля**: Сравниваем提供的 пароль с хешированным паролем в БД.
- **Обновление данных**: Обновляем имя и телефон пользователя, если они предоставлены.
- **Генерация токена**: Генерируем токен аутентификации и возвращаем его вместе с данными пользователя.

## Получение текущего пользователя

```

router.get("/me", authenticate, async (req, res) => {
  try {
    const user = await User.findByPk(req.user.userId, {
      attributes: { exclude: ["password"] },
    });

    if (!user) {
      return res.status(404).json({ error: "Пользователь не найден" });
    }

    res.status(200).json(user);
  } catch (error) {

```

```
    console.error("Ошибка получения пользователя:", error);
    res.status(500).json({ error: "Внутренняя ошибка сервера" });
  }
});
```

- `/me GET` : Роутер для получения текущего пользователя.
- `authenticate` : Middleware для проверки аутентификации пользователя.
- **Получение пользователя**: Получаем пользователя по ID из токена аутентификации.
- **Исключение пароля**: Исключаем поле `password` из ответа.

## Выход из системы

```
router.post("/logout", authenticate, (req, res) => {
  const authHeader = req.headers.authorization;
  const token = authHeader.split(" ")[1];

  if (sessions[token]) {
    delete sessions[token];
  }
  res.status(200).json({ message: "Выход выполнен успешно" });
});
```

- `/logout POST` : Роутер для выхода из системы.
- `authenticate` : Middleware для проверки аутентификации пользователя.
- **Удаление сессии**: Удаляем токен из объекта сессий.
- **Успешный ответ**: Возвращаем сообщение об успешном выходе.

## 3. Экспорт роутера

```
module.exports = router;
```

Этот файл предоставляет API-роутеры для управления аутентификацией и пользователями. Роутеры обеспечивают вход/регистрацию, получение текущего пользователя и выход из системы, а также взаимодействуют с моделью БД для выполнения операций.

## Использование API

- **Вход/Регистрация**

```
curl -X POST -H "Content-Type: application/json" -d '{"email":
"user@example.com", "password": "123456"}' http://localhost:3000/auth/login
```

- **Получение текущего пользователя**

```
curl -H "Authorization: Bearer <token>" http://localhost:3000/auth/me
```

- **Выход**

```
curl -X POST -H "Authorization: Bearer <token>"  
http://localhost:3000/auth/logout
```

## Ключевые моменты

- **Аутентификация:** Middleware `authenticate` проверяет наличие действительного токена в заголовке запроса.
- **Хеширование паролей:** Пароли хешируются с помощью библиотеки `bcrypt` для безопасности.
- **Управление сессиями:** Сессии пользователей хранятся в объекте `sessions` и удаляются при выходе.
- **Логирование:** Логирование процесса для отладки и мониторинга.

## Руководство по файлу `doctors.js`

Этот файл `doctors.js` содержит API-роутеры для управления докторами. Он предоставляет функциональность для получения списка всех докторов и получения информации о конкретном докторе. Вот полное руководство по этому файлу:

### 1. Импорты и Настройка

```
const express = require("express");  
const router = express.Router();  
const Doctor = require("../models/Doctor");  
const { authenticate } = require("../auth");
```

- `express` : Фреймворк для Node.js, используемый для обработки HTTP-запросов.
- `express.Router` : Объект, который используется для создания роутера.
- `Doctor` : Модель БД для доктора.
- `authenticate` : Middleware для проверки аутентификации пользователя.

### 2. Роутеры

#### Получение списка всех докторов

```
router.get("/", async (req, res) => {  
  try {
```

```

const doctors = await Doctor.findAll({
  attributes: [
    "id",
    "firstName",
    "lastName",
    "middleName",
    "specialization",
    "availableSlots",
  ],
});

const formattedDoctors = doctors.map((doctor) => ({
  id: doctor.id,
  fullName: `${doctor.lastName} ${doctor.firstName} ${doctor.middleName}
|| ""}`.trim(),
  specialization: doctor.specialization,
  availableSlots: doctor.availableSlots,
}));

res.status(200).json(formattedDoctors);
} catch (error) {
  console.error("Ошибка получения списка докторов:", error);
  res.status(500).json({ error: "Внутренняя ошибка сервера" });
}
});

```

- `/ GET` : Роутер для получения списка всех докторов.
- `Doctor.findAll` : Метод для поиска всех докторов в БД.
- **Форматирование ответа**: Преобразование данных о докторе в readable формат.
- **Отправка ответа**: Возвращается список докторов в JSON-формате.

## Получение информации о конкретном докторе

```

router.get("/:id", async (req, res) => {
  try {
    const doctor = await Doctor.findByPk(req.params.id);

    if (!doctor) {
      return res.status(404).json({ error: "Доктор не найден" });
    }

    const formattedDoctor = {
      id: doctor.id,
      fullName: `${doctor.lastName} ${doctor.firstName} ${doctor.middleName}
|| ""}`.trim(),
      firstName: doctor.firstName,
      lastName: doctor.lastName,
      middleName: doctor.middleName,

```

```

        specialization: doctor.specialization,
        availableSlots: doctor.availableSlots,
    };

    res.status(200).json(formattedDoctor);
  } catch (error) {
    console.error("Ошибка получения информации о докторе:", error);
    res.status(500).json({ error: "Внутренняя ошибка сервера" });
  }
});

```

- `/:id` GET : Роутер для получения информации о конкретном докторе.
- `Doctor.findByPk` : Метод для поиска доктора по его ID.
- **Проверка существования доктора**: Если доктора не существует, возвращается ошибка 404.
- **Форматирование ответа**: Преобразование данных о докторе в readable формат.
- **Отправка ответа**: Возвращаются данные о докторе в JSON-формате.

### 3. Экспорт роутера

```

module.exports = router;

```

Этот файл предоставляет API-роутеры для управления докторами. Роутеры обеспечивают получение списка докторов и информацию о конкретном докторе, а также взаимодействуют с моделью БД для выполнения операций.

## Использование API

- **Получение списка докторов**

```

curl http://localhost:3000/doctors/

```

- **Получение информации о конкретном докторе**

```

curl http://localhost:3000/doctors/1

```

## Ключевые моменты

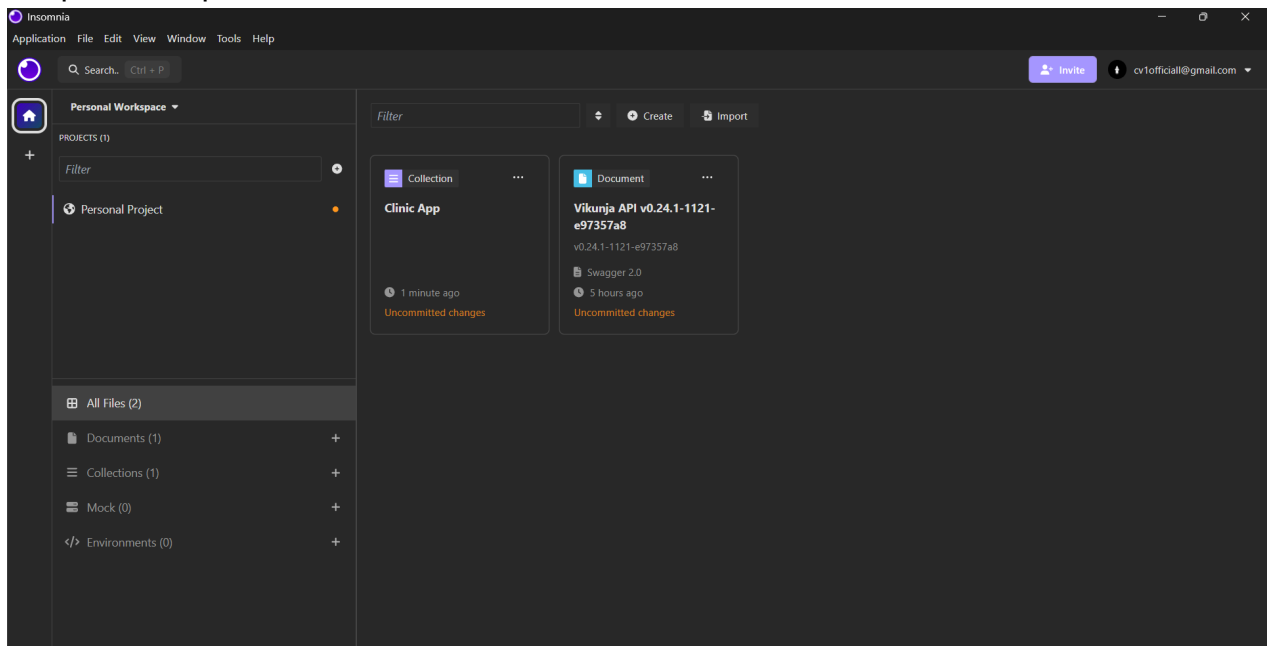
- **Форматирование данных**: Данные о докторях форматируются для удобства чтения.
- **Обработка ошибок**: Обработка ошибок для случаев, когда доктор не найден или возникает внутренняя ошибка сервера.



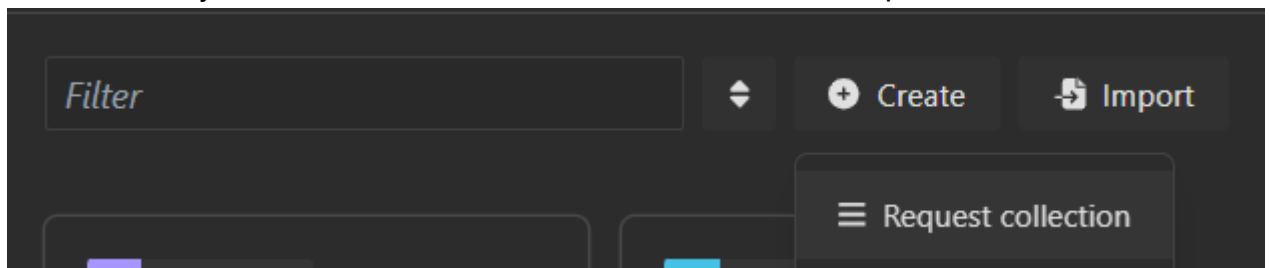
- **Интеграция с БД:** Роутеры используют модель **Doctor** для взаимодействия с базой данных.

# Проверка с использованием Insomnia:

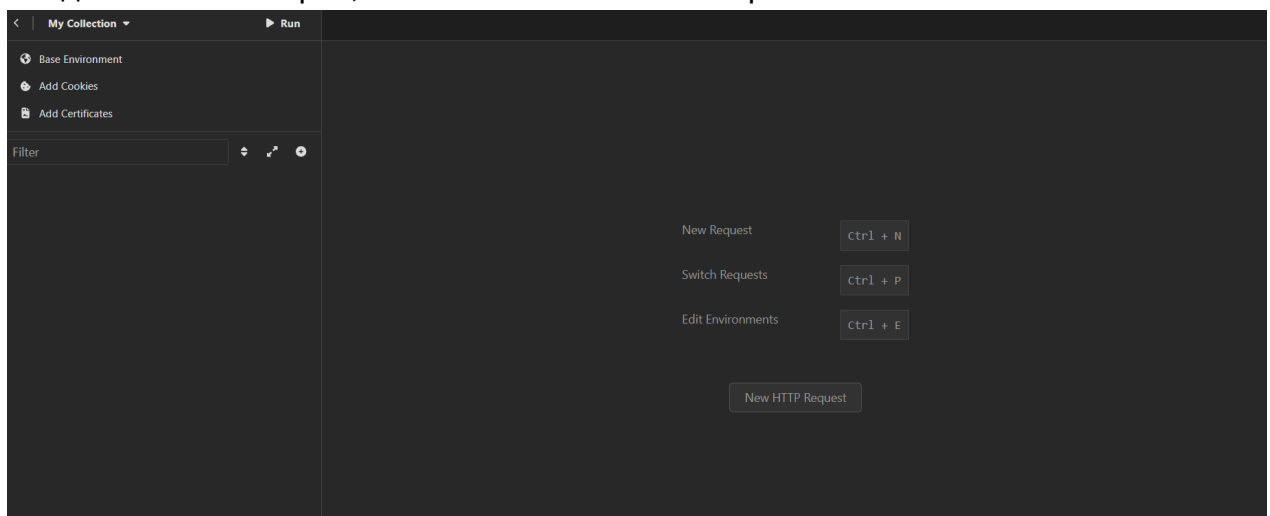
## 1. Открываем приложение и входим



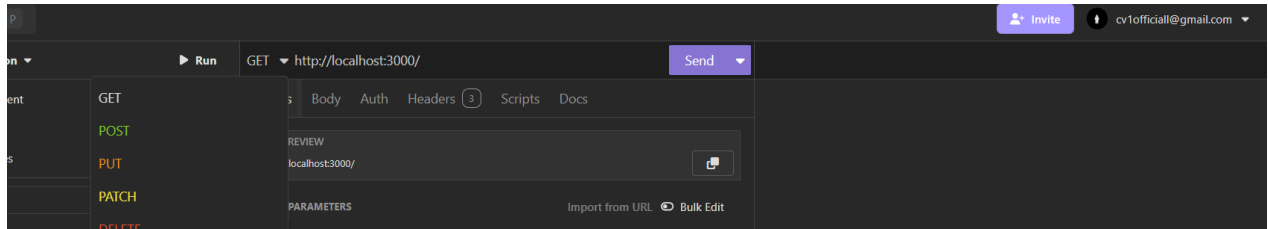
## 2. Создаём новую коллекцию, нажимаем на Create затем Request Collection



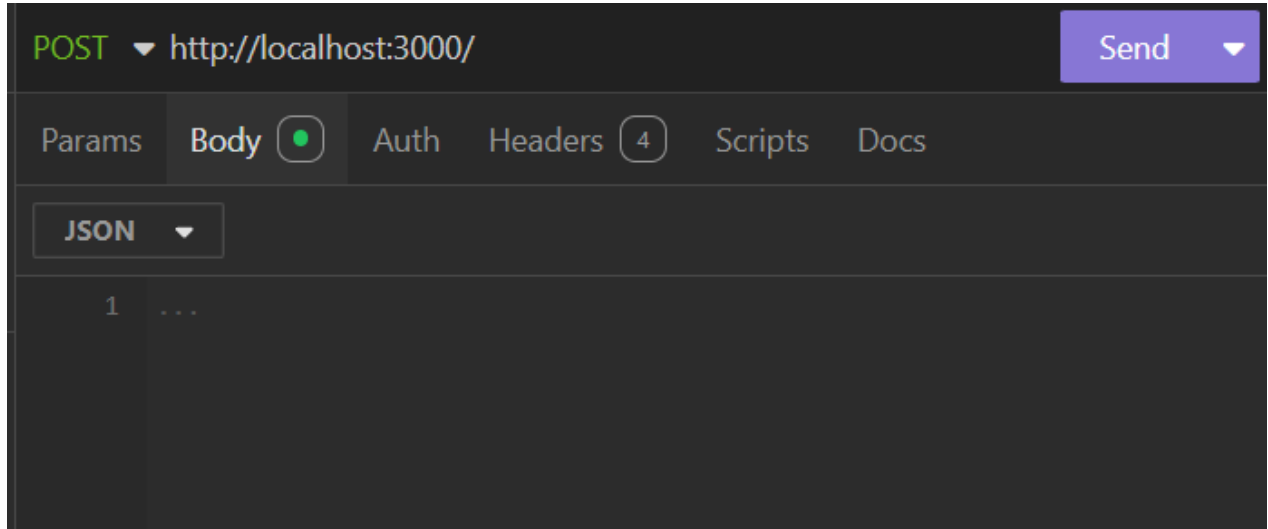
## 3. Создаём новый запрос, нажимаем New HTTP Request



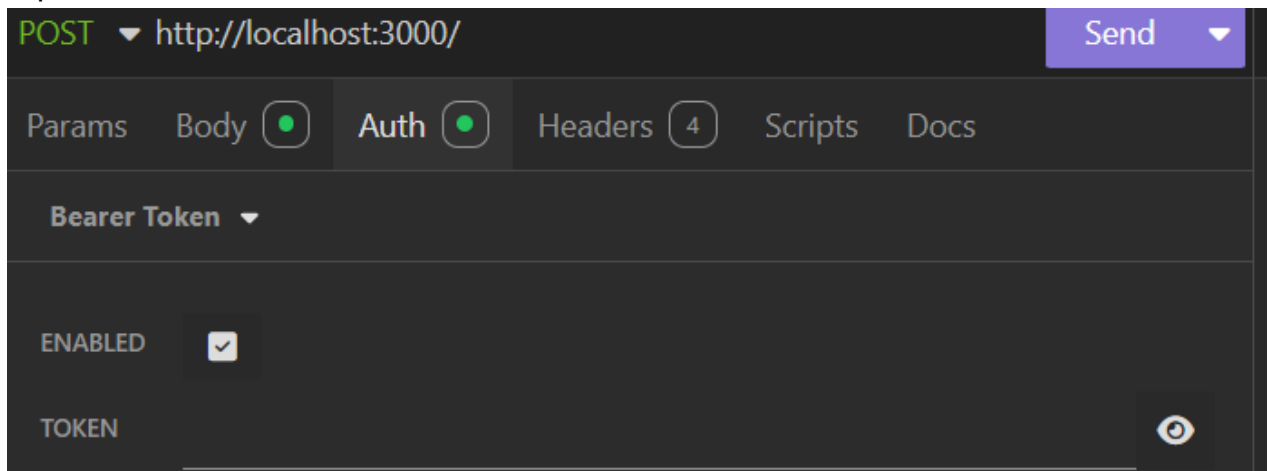
#### 4. Выбираем метод и прописываем куда мы хотим отправить наш запрос



#### 5. Если это запрос POST то необходимо добавить Body, делаем всё как на скриншоте

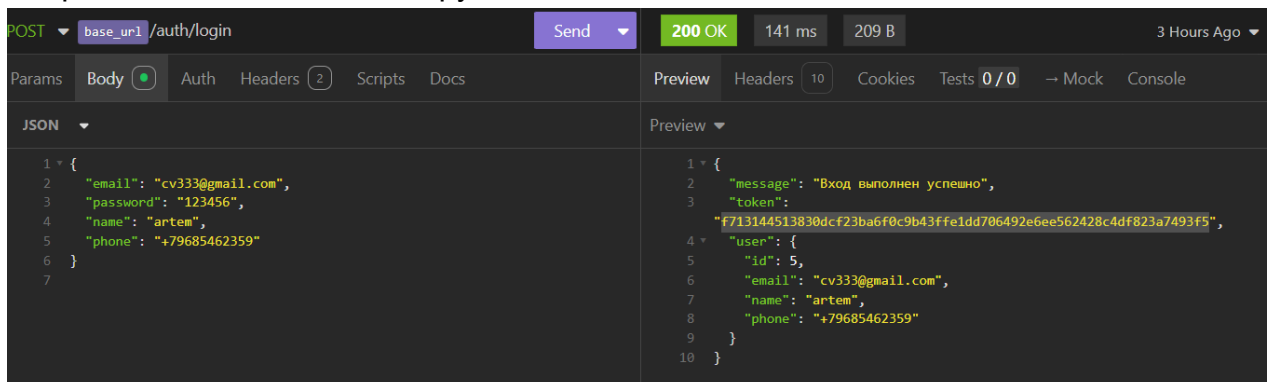


#### 6. Если возникает ошибка авторизации, то берёте токен, после того как авторизовались и вставляете в поле для ввода Token, и делаем так как на скриншоте



Тестовый запрос на тему API для записи человека на прием к врачу.

#### 7. Запрос на вход, здесь и копируется Token



## 8. Запрос на получение докторов

GET ▼ base\_url /doctors Send ▼ 200 OK 107 ms 1174 B 3 Hours Ago ▼

Params Body Auth Headers 2 Scripts Docs

Preview Headers 10 Cookies Tests 0/0 → Mock Console

No Body ▼

1 \* [  
2 \* {  
3 \* "id": 1,  
4 \* "fullName": "Смирнов Александр Иванович",  
5 \* "specialization": "Терапевт",  
6 \* "availableSlots": [  
7 \* "10:30-11:00",  
8 \* "11:00-11:30",  
9 \* "11:30-12:00",  
10 \* "12:00-12:30",  
11 \* "12:30-13:00",  
12 \* "13:00-13:30",  
13 \* "13:30-14:00",  
14 \* "14:00-14:30",  
15 \* "14:30-15:00",  
16 \* "15:00-15:30",  
17 \* "15:30-16:00",  
18 \* "16:00-16:30",  
19 \* "16:30-17:00",  
20 \* "17:00-17:30",  
21 \* "17:30-18:00",  
22 \* ]  
23 \* }  
24 \* ]

## 9. Запрос на запись на приём

POST ▼ base\_url /appointments Send ▼ 201 Created 90 ms 129 B

Params Body ● Auth ● Headers 2 Scripts Docs

Preview Headers 10 Cookies Tests 0/0 → Mock Console

JSON ▼

1 { "doctorId": 2, "slot": "10:00-10:30" }

Preview ▼  
1 {  
2 "message": "Запись на прием успешно создана",  
3 "appointment": {  
4 "id": 3,  
5 "doctorId": 2,  
6 "slot": "10:00-10:30"  
7 }  
8 }

## 10. Запрос на получение записей пользователя

GET ▼ http://localhost:3000/appointments/my Send ▼ 200 OK 51 ms 293 B 3 Hours Ago ▼

Params Body Auth ● Headers 3 Scripts Docs

Preview Headers 10 Cookies Tests 0/0 → Mock Console

No Body ▼

1 \* [  
2 \* {  
3 \* "id": 3,  
4 \* "slot": "10:00-10:30",  
5 \* "doctor": {  
6 \* "id": 2,  
7 \* "fullName": "Кузнецова Елена Петровна",  
8 \* "specialization": "Кардиолог"  
9 \* }  
10 \* },  
11 \* {  
12 \* "id": 4,  
13 \* "slot": "10:30-11:00",  
14 \* "doctor": {  
15 \* "id": 2,  
16 \* "fullName": "Кузнецова Елена Петровна",  
17 \* "specialization": "Кардиолог"  
18 \* }  
19 \* }  
20 \* ]