САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Бэк-энд разработка

Отчет

Лабораторная работа №1

Выполнил:
Григорян Самвел Ашотович
группа К3340

Проверил:
Добряков Д. И.

Санкт-Петербург

2025 г.

Задача

Создать boilerplate на Express.js + TypeORM + TypeScript с явным разделением на:
- Модели (entities)
- Контроллеры (controllers)
- Роуты (routes)

Ход работы

1. Структура проекта

Проект организован по принципу разделения ответственности (Separation of Concerns) со следующей структурой: src/

```
├── config/
│   └── database.ts         # Конфигурация TypeORM
├── controllers/
│   ├── BaseController.ts    # Базовый контроллер
│   └── userController.ts    # Логика работы с пользователями
├── entities/
│   ├── Base.ts             # Базовая
│   └── User.ts             # Модель пользователя
├── middleware/
│   └── auth.ts             # Middleware для аутентификации
├── routes/
│   ├── auth.ts             # Маршруты аутентификации
│   ├── users.ts            # Маршруты пользователей
└── index.ts                # Точка входа приложения
```

2. Настройка зависимостей

В package.json определены основные зависимости:

```
"dependencies": {
    "bcryptjs": "^2.4.3",
    "class-transformer": "^0.5.1",
    "class-validator": "^0.14.0",
    "cors": "^2.8.5",
    "dotenv": "^16.3.1",
    "express": "^4.18.2",
    "helmet": "^7.1.0",
    "jsonwebtoken": "^9.0.2",
    "morgan": "^1.10.0",
    "multer": "^1.4.5-lts.1",
    "reflect-metadata": "^0.1.13",
    "sqlite3": "^5.1.6",
    "swagger-jsdoc": "^6.2.8",
```

```
    "swagger-ui-express": "^5.0.1",
    "typeorm": "^0.3.17"
  },
  "devDependencies": {
    "@types/cors": "^2.8.17",
    "@types/express": "^4.17.21",
    "@types/jsonwebtoken": "^9.0.5",
    "@types/morgan": "^1.9.10",
    "@types/multer": "^1.4.11",
    "@types/node": "^20.10.0",
    "@types/swagger-jsdoc": "^6.0.4",
    "@types/swagger-ui-express": "^4.1.8",
    "nodemon": "^3.0.2",
    "ts-node": "^10.9.1",
    "typescript": "^5.3.2"
  }
```

3. Конфигурация базы данных

В файле src/config/database.ts настроено подключение к БД через TypeORM:

```
import { DataSource } from "typeorm";
import { User } from "../entities/User";
import { Apartment } from "../entities/Apartment";
import { Building } from "../entities/Building";
import { Contract } from "../entities/Contract";
import dotenv from "dotenv";

dotenv.config();

export const AppDataSource = new DataSource({
  type: "sqlite",
  database: process.env.DATABASE_PATH || "./database.sqlite",
  synchronize: true,
  logging: process.env.NODE_ENV === "development",
  entities: [User, Apartment, Building, Contract],
  subscribers: [],
  migrations: [],
});

export const initializeDatabase = async () => {
  try {
    await AppDataSource.initialize();
    console.log("Database connection established");
  } catch (error) {
    console.error("Error connecting to database:", error);
    process.exit(1);
  }
};
```

4. Модели (Entities)

С помощью TypeORM реализованы все необходимые сущности. Ниже представлена модель пользователя с использованием TypeORM декораторов:

```
@Entity("users")
export class User {
  @PrimaryGeneratedColumn("increment")
  UserID!: number;

  @Column({ type: "varchar", length: 150, unique: true })
  username!: string;

  @Column({ type: "varchar", length: 150 })
  first_name!: string;

  @Column({ type: "varchar", length: 150 })
  last_name!: string;

  @Column({ type: "varchar", length: 254, unique: true })
  email!: string;

  @Column({ type: "varchar", length: 128 })
  password!: string;

  @Column({ type: "varchar", length: 100, nullable: true })
  Passport!: string;

  @Column({ type: "varchar", length: 11, nullable: true })
  Phone!: string;

  @Column({ type: "date", nullable: true })
  BirthDate!: Date;

  @Column({ type: "varchar", length: 255, nullable: true })
  Photo!: string;

  @Column({ type: "boolean", default: false })
  is_staff!: boolean;

  @Column({ type: "boolean", default: true })
  is_active!: boolean;

  @Column({ type: "boolean", default: false })
  is_superuser!: boolean;

  @Column({ type: "datetime", nullable: true })
  last_login!: Date;

  @CreateDateColumn()
  date_joined!: Date;
```

```
  @UpdateDateColumn()
  updated_at!: Date;

  @OneToMany(() => Contract, contract => contract.AgentID)
  agentContracts!: Contract[];

  @OneToMany(() => Contract, contract => contract.ClientID)
  clientContracts!: Contract[];
}
```

5. Контроллеры (Controllers)

Реализован CRUD-контроллер для пользователей:

```
import { Request, Response } from "express";
import { AppDataSource } from "../config/database";
import { User } from "../entities/User";
import bcrypt from "bcryptjs";
import jwt from "jsonwebtoken";

const userRepository = AppDataSource.getRepository(User);

export const getAllUsers = async (req: Request, res: Response) => {
  try {
    const users = await userRepository.find({
      select: ["UserID", "username", "first_name", "last_name", "email", "is_staff",
"is_active"]
    });
    res.json(users);
  } catch (error) {
    res.status(500).json({ message: "Error fetching users", error });
  }
};

export const getAgents = async (req: Request, res: Response) => {
  try {
    const agents = await userRepository.find({
      where: { is_staff: true },
      select: ["UserID", "username", "first_name", "last_name", "email", "is_active"]
    });
    res.json(agents);
  } catch (error) {
    res.status(500).json({ message: "Error fetching agents", error });
  }
};

export const getClients = async (req: Request, res: Response) => {
  try {
    const clients = await userRepository.find({
      where: { is_staff: false },
      select: ["UserID", "username", "first_name", "last_name", "email", "is_active"]
    });
    res.json(clients);
```

```
  } catch (error) {
    res.status(500).json({ message: "Error fetching clients", error });
  }
};

export const getUserById = async (req: Request, res: Response) => {
  try {
    const { id } = req.params;
    const user = await userRepository.findOne({ where: { UserID: parseInt(id) } });

    if (!user) {
      return res.status(404).json({ message: "User not found" });
    }

    res.json(user);
  } catch (error) {
    res.status(500).json({ message: "Error fetching user", error });
  }
};

export const createUser = async (req: Request, res: Response) => {
  try {
    const { username, first_name, last_name, email, password, Passport, Phone,
BirthDate, is_staff } = req.body;

    const hashedPassword = await bcrypt.hash(password, 10);

    const user = userRepository.create({
      username,
      first_name,
      last_name,
      email,
      password: hashedPassword,
      Passport,
      Phone,
      BirthDate: BirthDate ? new Date(BirthDate) : undefined,
      is_staff: is_staff || false
    });

    const savedUser = await userRepository.save(user);
    res.status(201).json(savedUser);
  } catch (error) {
    res.status(500).json({ message: "Error creating user", error });
  }
};

export const updateUser = async (req: Request, res: Response) => {
  try {
    const { id } = req.params;
    const updateData = req.body;

    if (updateData.password) {
      updateData.password = await bcrypt.hash(updateData.password, 10);
```

```
    }

    if (updateData.BirthDate) {
      updateData.BirthDate = new Date(updateData.BirthDate);
    }

    await userRepository.update(id, updateData);
    const updatedUser = await userRepository.findOne({ where: { UserID: parseInt(id)
} });

    res.json(updatedUser);
  } catch (error) {
    res.status(500).json({ message: "Error updating user", error });
  }
};

export const deleteUser = async (req: Request, res: Response) => {
  try {
    const { id } = req.params;
    await userRepository.delete(id);
    res.json({ message: "User deleted successfully" });
  } catch (error) {
    res.status(500).json({ message: "Error deleting user", error });
  }
};

export const login = async (req: Request, res: Response) => {
  try {
    const { username, password } = req.body;

    const user = await userRepository.findOne({ where: { username } });
    if (!user) {
      return res.status(401).json({ message: "Invalid credentials" });
    }

    const isValidPassword = await bcrypt.compare(password, user.password);
    if (!isValidPassword) {
      return res.status(401).json({ message: "Invalid credentials" });
    }

    const expiresIn: string = process.env.JWT_EXPIRES_IN || "24h";
    const jwtSecret: string = process.env.JWT_SECRET || "fallback-secret";
    const token = (jwt.sign as any)(
      { userId: user.UserID, username: user.username },
      jwtSecret,
      { expiresIn }
    );

    res.json({ token, user: { UserID: user.UserID, username: user.username, email:
user.email } });
  } catch (error) {
    res.status(500).json({ message: "Error during login", error });
  }
```

```
};
```

6. Роуты (Routes)

Маршруты организованы по модульному принципу с использованием Express Router

```
/**
 * @swagger
 * tags:
 *   name: Users
 *   description: Управление пользователями
 */

import { Router } from "express";
import {
  getAllUsers,
  getAgents,
  getClients,
  getUserById,
  createUser,
  updateUser,
  deleteUser
} from "../controllers/userController";
import { authMiddleware } from "../middleware/auth";

const router = Router();

router.get("/", getAllUsers);
router.get("/agents", getAgents);
router.get("/clients", getClients);
router.get("/:id", getUserById);
router.post("/", createUser);
router.put("/:id", authMiddleware, updateUser);
router.delete("/:id", authMiddleware, deleteUser);

export default router;
```

7. Middleware аутентификации

Реализован JWT-based middleware для защиты маршрутов:

```
import { Request, Response, NextFunction } from "express";
import jwt from "jsonwebtoken";
import { AppDataSource } from "../config/database";
import { User } from "../entities/User";

export interface AuthRequest extends Request {
  user?: User;
}

export const authMiddleware = async (req: AuthRequest, res: Response, next:
NextFunction) => {
```

```
  try {
    const token = req.header("Authorization")?.replace("Bearer ", "");

    if (!token) {
      return res.status(401).json({ message: "Access denied. No token provided." });
    }

    const decoded = jwt.verify(token, process.env.JWT_SECRET || "fallback-secret") as
any;
    const userRepository = AppDataSource.getRepository(User);
    const user = await userRepository.findOne({ where: { UserID: decoded.userId } });

    if (!user) {
      return res.status(401).json({ message: "Invalid token." });
    }

    req.user = user;
    next();
  } catch (error) {
    res.status(401).json({ message: "Invalid token." });
  }
};

export const optionalAuthMiddleware = async (req: AuthRequest, res: Response, next:
NextFunction) => {
  try {
    const token = req.header("Authorization")?.replace("Bearer ", "");

    if (token) {
      const decoded = jwt.verify(token, process.env.JWT_SECRET || "fallback-secret")
as any;
      const userRepository = AppDataSource.getRepository(User);
      const user = await userRepository.findOne({ where: { UserID: decoded.userId }
});
      req.user = user || undefined;
    }

    next();
  } catch (error) {
    next();
  }
};
```

Вывод

В ходе выполнения лабораторной работы был успешно создан boilerplate на Express.js + TypeORM + TypeScript с четким разделением на модели, контроллеры и роуты. Достигнутые результаты:

1. Архитектурное разделение: Реализовано четкое разделение ответственности между слоями приложения (entities, controllers, routes)
2. TypeORM интеграция: Настроена работа с базой данных SQLite через TypeORM с автоматической синхронизацией схемы
3. Аутентификация: Реализована JWT-based аутентификация с middleware для защиты маршрутов
4. REST API: Создан полный набор CRUD операций для всех сущностей системы
5. Безопасность: Реализовано хеширование паролей с помощью bcryptjs
6. Обработка ошибок: Настроена централизованная обработка ошибок
7. Валидация: Использованы TypeORM декораторы для валидации данных

Технологический стек: Node.js, Express.js, TypeScript, TypeORM, SQLite, JWT для аутентификации.

Проект готов к использованию как основа для разработки веб-приложений с REST API и может быть легко расширен дополнительной функциональностью.