САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Бэк-энд разработка

Отчет

Лабораторная работа №2

Выполнил:

Кадникова Екатерина

Группа К3341

Проверил:
Добряков Д. И.

Санкт-Петербург

2025 г.

Задача

По выбранному варианту необходимо будет реализовать RESTful API средствами express + typescript (используя ранее написанный boilerplate).

Вариант: Сервис для аренды недвижимости

- Вход
- Регистрация
- Личный кабинет пользователя (список арендованных и арендующихся объектов)
- Поиск недвижимости с фильтрацией по типу, цене, расположению
- Страница объекта недвижимости с фото, описанием и условиями аренды
- История сообщений и сделок пользователя

Ход работы

1. Модели

Помимо основных моделей User и Role, реализованных в рамках ЛР1, были реализованы модели в соответствие со схемой БД (см. Рисунок 1):

- Property (Недвижимость);
- Rental (Аренда);
- Favorite (Избранное);
- Message (Сообщение).

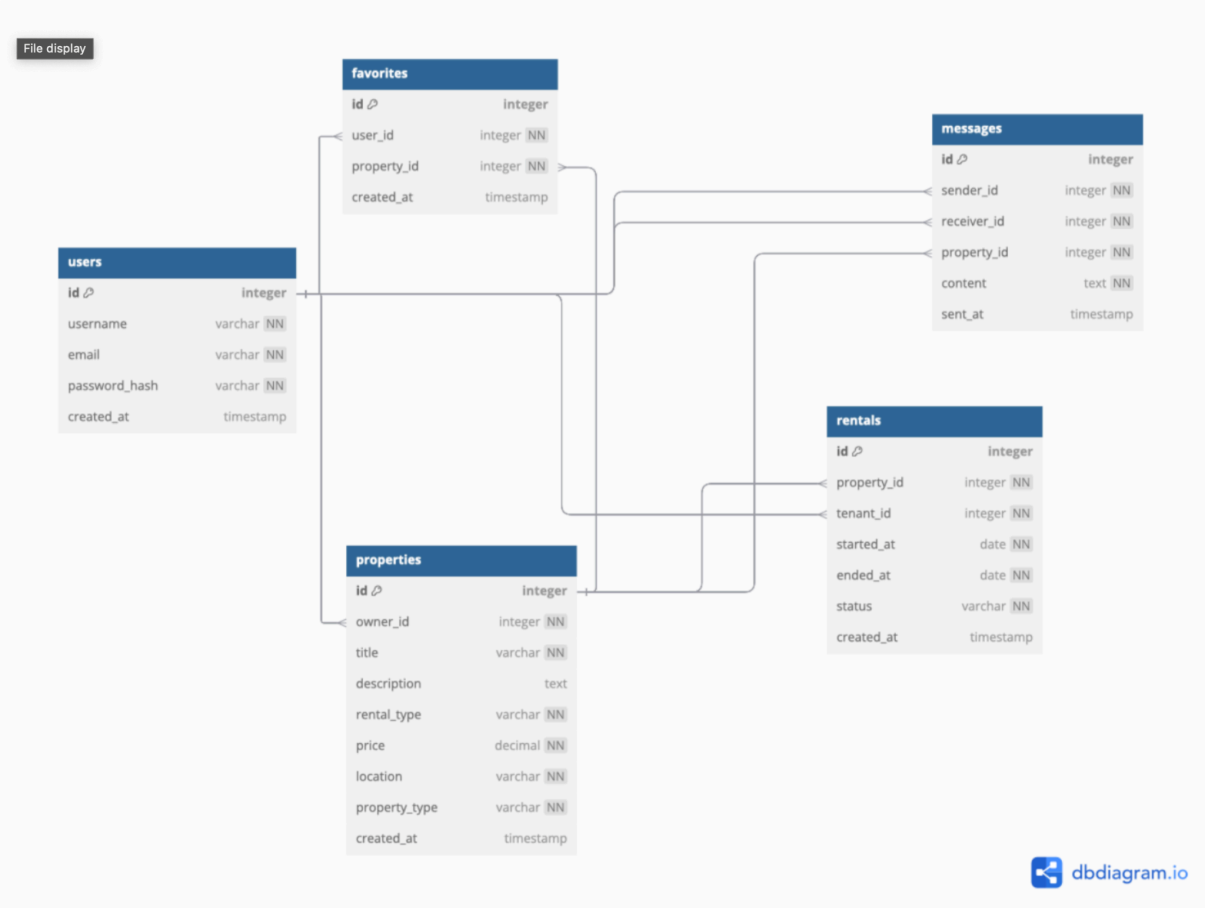

Рисунок 1 - Схема БД

Пример реализованной модели можно увидеть на Листинге 1.

Листинг 1 - Модель Property:

```
import { Entity, PrimaryGeneratedColumn, Column, CreateDateColumn, ManyToOne, OneToMany } from 'typeorm';
import { User } from './user';
import { Rental } from './rental';
import { Message } from './message';
import { Favorite } from './favorite';
import {PropertyType} from "./enums/propertyType";
import {RentalType} from "./enums/rentalType";

@Entity('properties')
export class Property {
    @PrimaryGeneratedColumn()
    id!: number;

    @ManyToOne(() => User, user => user.properties)
    owner!: User;

    @Column()
    title!: string;
```

```
    @Column({ type: 'text', nullable: true })
    description?: string;

    @Column({ type: 'enum', enum: RentalType })
    rental_type!: RentalType;

    @Column({ type: 'decimal', precision: 10, scale: 2 })
    price!: number;

    @Column()
    location!: string;

    @Column({ type: 'enum', enum: PropertyType })
    property_type!: PropertyType;

    @CreateDateColumn({ type: 'timestamp' })
    created_at!: Date;

    @OneToMany(() => Rental, rental => rental.property)
    rentals!: Rental[];

    @OneToMany(() => Message, message => message.property)
    messages!: Message[];

    @OneToMany(() => Favorite, favorite => favorite.property)
    favorites!: Favorite[];
}
```

2. Сервисы

Для работы с каждой сущностью (Property, Rental, Favorite, Message) были реализованы соответствующие сервисы. Эти сервисы содержат бизнес-логику приложения и вызываются из контроллеров для обработки HTTP-запросов.

Рассмотрим структуру реализованных сервисов на примере сервиса для работы с недвижимостью (см. Листинг 2).

Листинг 2 - propertyServicets:

```
import { AppDataSource } from "../data-source";
import { Property } from "../models/property";
import { User } from "../models/user";
import { Role } from "../models/enums/role";
import { CreatePropertyDto, UpdatePropertyDto, SearchPropertyDto } from
"../dto/propertyDto";

const propertyRepository = AppDataSource.getRepository(Property);
const userRepository = AppDataSource.getRepository(User);

class PropertyService {
```

```
        async getAllProperties() {
            return propertyRepository.find({ relations: ["owner"] });
        }

        async getPropertyById(id: number) {
            const property = await propertyRepository.findOne({
                where: { id },
                relations: ["owner"]
            });
            return property;
        }

        async createProperty(dto: CreatePropertyDto, userId: number) {
            const owner = await userRepository.findOneBy({ id: userId });
            if (!owner) {
                throw new Error("User not found");
            }

            const property = propertyRepository.create({ ...dto, owner });
            await propertyRepository.save(property);

            return propertyRepository.findOne({
                where: { id: property.id },
                relations: ["owner"]
            });
        }

        async updateProperty(id: number, dto: UpdatePropertyDto, userId: number,
role: Role) {
            const property = await propertyRepository.findOne({
                where: { id },
                relations: ["owner"]
            });

            if (!property) {
                throw new Error("Property not found");
            }

            if (property.owner.id !== userId && role !== Role.ADMIN) {
                throw new Error("Forbidden");
            }

            Object.assign(property, dto);
            await propertyRepository.save(property);

            return property;
        }

        async deleteProperty(id: number, userId: number, role: Role) {
            const property = await propertyRepository.findOne({
                where: { id },
                relations: ["owner"]
            });
```

```
            if (!property) {
                throw new Error("Property not found");
            }

            if (property.owner.id !== userId && role !== Role.ADMIN) {
                throw new Error("Forbidden");
            }

            await propertyRepository.remove(property);
            return { message: "Property deleted successfully" };
        }

        async searchProperties(dto: SearchPropertyDto) {
            const qb = propertyRepository.createQueryBuilder("property")
                .leftJoinAndSelect("property.owner", "owner");

            if (dto.location) {
                qb.andWhere("property.location ILIKE :location", { location:
`%${dto.location}%` });
            }

            if (dto.minPrice) {
                qb.andWhere("property.price >= :minPrice", { minPrice:
Number(dto.minPrice) });
            }

            if (dto.maxPrice) {
                qb.andWhere("property.price <= :maxPrice", { maxPrice:
Number(dto.maxPrice) });
            }

            if (dto.propertyType) {
                qb.andWhere("property.property_type = :propertyType", {
propertyType: dto.propertyType });
            }

            if (dto.rentalType) {
                qb.andWhere("property.rental_type = :rentalType", { rentalType:
dto.rentalType });
            }

            return qb.getMany();
        }
    }

    export default new PropertyService();
```

Сервисы представляют собой слой бизнес-логики, который:

1. Принимает данные от контроллеров.
2. Взаимодействует с базой данных через репозитории TypeORM.

3. Выполняет проверки (валидация, права доступа и т.д.).
4. Возвращает результат контроллеру для формирования HTTP-ответа.

Некоторые методы со “сложной” структуры вводной информации использую DTO - Data Transfer Objects. Они определяют структуру входных/выходных данных (например, CreatePropertyDto, UpdatePropertyDto - см. Листинг 3) и используются для передачи данных между слоями приложения.

Листинг 3 - propertyDto.ts:

```
import { IsString, IsNumber, IsEnum, IsNotEmpty, IsOptional, IsNumberString
} from "class-validator";
import { RentalType } from "../models/enums/rentalType";
import { PropertyType } from "../models/enums/propertyType";

export class CreatePropertyDto {
   @IsString()
   @IsNotEmpty()
   title?: string;

   @IsString()
   @IsNotEmpty()
   description?: string;

   @IsEnum(RentalType)
   rental_type?: RentalType;

   @IsNumber()
   price?: number;

   @IsString()
   @IsNotEmpty()
   location?: string;

   @IsEnum(PropertyType)
   property_type?: PropertyType;
}

export class UpdatePropertyDto {
   @IsString()
   title?: string;

   @IsString()
   description?: string;

   @IsEnum(RentalType)
   rental_type?: RentalType;

   @IsNumber()
   price?: number;
```

```
    @IsString()
    location?: string;

    @IsEnum(PropertyType)
    property_type?: PropertyType;
}

export class SearchPropertyDto {
    @IsOptional()
    @IsString()
    location?: string;

    @IsOptional()
    @IsNumberString()
    minPrice?: string;

    @IsOptional()
    @IsNumberString()
    maxPrice?: string;

    @IsOptional()
    @IsEnum(PropertyType)
    propertyType?: PropertyType;

    @IsOptional()
    @IsEnum(RentalType)
    rentalType?: RentalType;
}
```

Пример потока данных в PropertyService:

1. Запрос на создание объекта (createProperty):
    - Контроллер передает в сервис CreatePropertyDto и userId.
    - Сервис проверяет, существует ли пользователь через userRepository.
    - Создает новую запись в БД через propertyRepository.create().
    - Сохраняет данные через propertyRepository.save().
2. Поиск объектов (searchProperties):
    - Использует QueryBuilder для гибкого формирования SQL-запроса с фильтрами.
    - Пример фильтрации по минимальной цене:

```
qb.andWhere("property.price >= :minPrice", { minPrice: Number(dto.minPrice) });
```

3. Контроллеры

Для работы с каждой сущностью также были созданы контроллеры, которые отвечают за:

1. Прием HTTP-запросов и извлечение данных (параметры, тело запроса).
2. Валидацию входных данных (через DTO).
3. Вызов сервисов для выполнения бизнес-логики.
4. Формирование HTTP-ответов (успешных или ошибок).

Рассмотрим прицнип работы контроллеров на примере контроллера для работы с недвижимостью (см. Листинг 4).

Листинг 4 - propertyController.ts:

```
import { Request, Response } from "express";
import propertyService from "../services/propertyService";
import { CreatePropertyDto, UpdatePropertyDto, SearchPropertyDto } from
"../dto/propertyDto";
import { validateDto } from "../utils/validateDto";
import { Role } from "../models/enums/role";

export const getAllProperties = async (req: Request, res: Response) => {
    try {
        const properties = await propertyService.getAllProperties();
        res.json(properties);
    } catch {
        res.status(500).json({ message: "Error fetching properties" });
    }
};

export const getPropertyById = async (req: Request, res: Response) => {
    const propertyId = Number(req.params.id);

    try {
        const property = await propertyService.getPropertyById(propertyId);
        if (!property) {
            res.status(404).json({ message: "Property not found" });
            return;
        }
        res.json(property);
    } catch {
        res.status(500).json({ message: "Error fetching property" });
    }
};

export const createProperty = async (req: Request, res: Response) => {
    const dto = await validateDto(CreatePropertyDto, req.body, res);
    if (!dto) return;

    const userId = req.user?.id;
    if (!userId) {
        res.status(401).json({ message: "Unauthorized" });
```

```
            return;
        }

        try {
            const property = await propertyService.createProperty(dto, userId);
            res.status(201).json(property);
        } catch (error: any) {
            res.status(400).json({ message: error.message });
        }
    };

    export const updateProperty = async (req: Request, res: Response) => {
        const dto = await validateDto(UpdatePropertyDto, req.body, res);
        if (!dto) return;

        const userId = req.user?.id;
        const roleRaw = req.user?.role;

          if (!userId || !roleRaw || !Object.values(Role).includes(roleRaw as
Role)) {
            res.status(401).json({ message: "Unauthorized or invalid role" });
            return;
        }

        const role = roleRaw as Role;
        const propertyId = Number(req.params.id);

        try {
              const property = await propertyService.updateProperty(propertyId,
dto, userId, role);
            res.json(property);
        } catch (error: any) {
               res.status(500).json({ message: error.message || "Error updating
property" });
        }
    };

    export const deleteProperty = async (req: Request, res: Response) => {
        const userId = req.user?.id;
        const roleRaw = req.user?.role;

          if (!userId || !roleRaw || !Object.values(Role).includes(roleRaw as
Role)) {
            res.status(401).json({ message: "Unauthorized or invalid role" });
            return;
        }

        const role = roleRaw as Role;
        const propertyId = Number(req.params.id);

        try {
                const result = await propertyService.deleteProperty(propertyId,
userId, role);
```

```
            res.json(result);
        } catch (error: any) {
                res.status(500).json({ message: error.message || "Error deleting
property" });
        }
    };

    export const searchProperties = async (req: Request, res: Response) => {
        const dto = await validateDto(SearchPropertyDto, req.query, res);
        if (!dto) return;

        try {
            const properties = await propertyService.searchProperties(dto);
            res.json(properties);
        } catch {
            res.status(500).json({ message: "Error searching properties" });
        }
    };
```

Основные особенности:

1. DTO Используются для валидации входных данных. Метод validateDto (см. Листинг 5).проверяет соответствие структуры и типов данных.
2. Контроллеры делегируют логику сервисам (например, propertyService).
3. Все операции оборачиваются в try-catch для возврата структурированных ошибок.

Листинг 4 - validateDto.ts:

```
    import { plainToInstance } from "class-transformer";
    import { validate as classValidate } from "class-validator";
    import { Response } from "express";

    export async function validateDto<T extends object>(
        dtoClass: new () => T,
        plain: object,
        res: Response
    ): Promise<T | null> {
        const instance = plainToInstance(dtoClass, plain);
        const errors = await classValidate(instance);

        if (errors.length > 0) {
            res.status(400).json({
                message: "Validation failed",
                errors: errors.map(err => ({
                    property: err.property,
                    constraints: err.constraints
                }))
            });
            return null;
        }
```

```
    return instance;
}
```

4. Роуты

Было реализовано четкое разделение роутов для обработки запросов, связанных с разными сущностями. Все роуты сгруппированы по своей тематике и подключаются через отдельные модули.

Рассмотрим принцип работы роута на примере роутов для работы я недвижимостью (см. Листинг 5).

Листинг 5 - propertyRoutes.ts:

```
import { Router } from "express";
import {
   getAllProperties,
   getPropertyById,
   createProperty,
   updateProperty,
   deleteProperty,
   searchProperties
} from "../controllers/propertyController";
import { authenticateToken } from "../middlewares/authMiddleware";

const router = Router();

router.get("/", getAllProperties);
router.get("/search", searchProperties);
router.get("/:id", getPropertyById);
router.post("/", authenticateToken, createProperty);
router.put("/:id", authenticateToken, updateProperty);
router.delete("/:id", authenticateToken, deleteProperty);

export default router;
```

Роуты используются для:

1. Сопоставления URL-путей с соответствующими контроллерами
2. Определения HTTP-методов (GET, POST, PUT, DELETE и др.)
3. Подключения middleware (например, для аутентификации)
4. Организация структуры API (разделение на логические группы)

Все маршруты в этом файле относятся к /properties, а далее подключаются в файле app.ts как:

```
app.use("/api/properties", propertyRoutes);
```

В итоге сформирована следующая структура:

Метод	**Путь**	**Контроллер**	**Требует аутентификации**
GET	/properties	getAllProperties	Нет
GET	/properties/search	searchProperties	Нет
GET	/properties/:id	getPropertyById	Нет
POST	/properties	createProperty	Да
PUT	/properties/:id	updateProperty	Да
DELETE	/properties/:id	deleteProperty	Да

Вывод

В рамках работы было разработано RESTful API для сервиса аренды недвижимости с использованием Express, TypeORM и TypeScript. Реализована модульная структура проекта с четким разделением на модели, сервисы, контроллеры и роуты, а также базовые функции для работы с недвижимостью, включая поиск, бронирование и управление объявлениями.