САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Бэк-энд разработка

Отчет

Лабораторная работа №1

Выполнил:

Корчагин Вадим

Группа
К3341

Проверил:
Добряков Д. И.

Санкт-Петербург

2025 г.

Тема

Платформа для фитнес-тренировок и здоровья.

Задача

1. Нужно написать свой boilerplate на express + TypeORM + typescript.
2. Должно быть явное разделение на:
    - модели
    - контроллеры
    - роуты

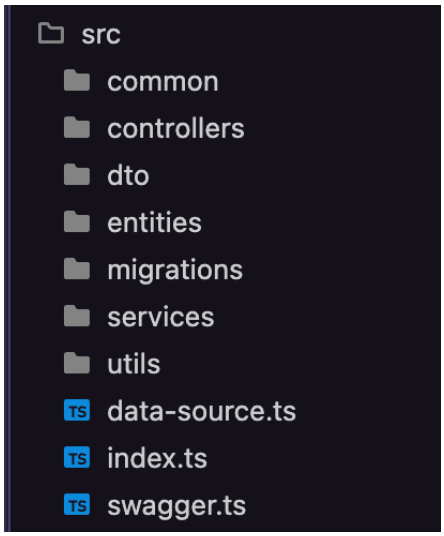
Технологии:

- Express.js
- TypeORM
- TypeScript
- class-validator, class-transformer
- dotenv
- routing-controllers

Ход работы

Обновленная структура проекта

Относительно предыдущей работы “ДЗ2” были добавлены разделы common, dto, utils. Так как были добавлены абстрактные классы в common, были обновлены контроллеры и сервисы.



- **common** — отвечает за абстрактные классы, такие как BaseService и BaseController, которые позволяют переиспользовать общую логику для работы с сущностями и упростить реализацию CRUD в сервисах и контроллерах.
- **dto** — содержит классы для входных и выходных данных API. DTO используются для:
    - Валидации входящих данных (create, update)
    - Формализации структуры ответов (response)
    - Генерации схем в Swagger через аннотации class-validator и class-transformer
- **utils** — включает вспомогательные функции, такие как:
    - hashPassword.ts и checkPassword.ts — безопасное хеширование и сравнение паролей
    - authCheck.ts — реализация JWT-аутентификации для защиты маршрутов

Common

В директории **common/** находятся абстрактные классы, которые позволяют переиспользовать общую бизнес-логику для всех сущностей:

BaseService
Универсальный сервис, работающий с любой сущностью через TypeORM репозиторий. Поддерживает CRUD-операции (findAll, findOne, create, update, remove).

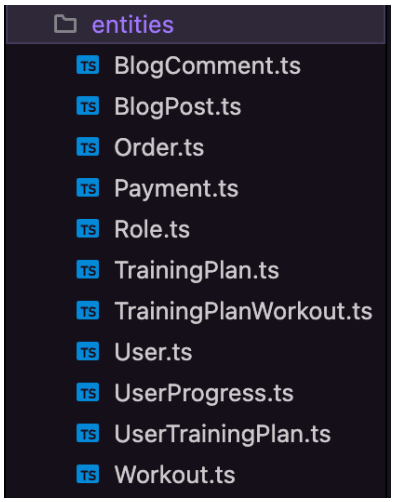
```
export abstract class BaseService<Entity> {
  protected repository: Repository<Entity>;

  constructor(entity: { new (): Entity }) {
    this.repository = AppDataSource.getRepository(entity);
  }
      You, 6 days ago • feat: Lab1, Lab2, HW3, HW4 are done …
  async findAll(relations: string[] = []): Promise<Entity[]> {
    return this.repository.find({ relations });
  }

  async findOne(id: number, relations: string[] = []): Promise<Entity | null> {
    return this.repository.findOne({ where: { id } as any, relations });
  }

  async create(data: DeepPartial<Entity>): Promise<Entity> {
    const entity = this.repository.create(data);
    return this.repository.save(entity);
  }

  async update(id: number, data: QueryDeepPartialEntity<Entity>): Promise<Entity | null> {
    await this.repository.update(id, data);
    return this.findOne(id);
  }

  async remove(id: number): Promise<void> {
    await this.repository.delete(id);
  }
}
```

BaseController
Абстрактный контроллер, реализующий шаблон использования сервисов в конкретных контроллерах. Позволяет сократить дублирование кода.

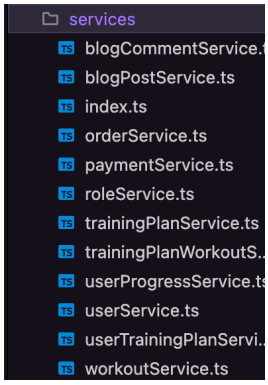
```
You, 6 days ago | 1 author (You)
export abstract class BaseController<Entity> {
  protected service: BaseService<Entity>;

  constructor(service: BaseService<Entity>) {
    this.service = service;
  }

  protected getRelations(): string[] {
    return [];
  }
}       You, 6 days ago • feat: Lab1, Lab2, HW3, HW4 are done
```

Entities

Относительно Домашней работы 2, данный раздел не изменился.



Services



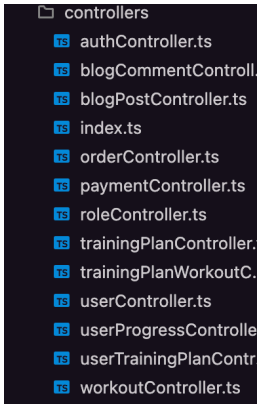
Слой services/ отвечает за реализацию бизнес-логики приложения. Каждый сервис расширяет BaseService, что позволяет использовать универсальные CRUD-методы, а при необходимости — добавлять специфические методы:

- В сервисах реализуется доступ к данным через TypeORM репозитории.
- Связанные данные (relations) подгружаются через методы findOneWithRelations, findAllWithRelations.
- В сервисе UserService реализована хешировка пароля при создании пользователя.
- Сервисы не зависят от Express — они чисто логические и переиспользуемые.

Пример UserService:

```
You, 6 days ago | 1 author (You)
export class UserService extends BaseService<User> {
  constructor() {
    super(User);
  }

  async findAllWithRelations() {
    return this.repository.find({ relations: ["role"] });
  }

  async findOneWithRelations(id: number) {
    return this.repository.findOne({ where: { id }, relations: ["role"] });
  }

  async findByEmail(email: string) {
    return this.repository.findOne({ where: { email }, relations: ["role"] });
  }

  override async create(data: Partial<User>) {
    const user = this.repository.create({
      ...data,
      password_hash: hashPassword(data.password_hash || ""), // Хешируем пароль при создании
    });
    return this.repository.save(user);      You, 6 days ago • feat: Lab1, Lab2, HW3, HW4 are d
  }
}
```

Controllers



В директории **controllers/** реализованы REST-контроллеры для всех сущностей. Каждый контроллер отвечает за приём HTTP-запросов, валидацию входных данных и делегирование работы соответствующему сервису.

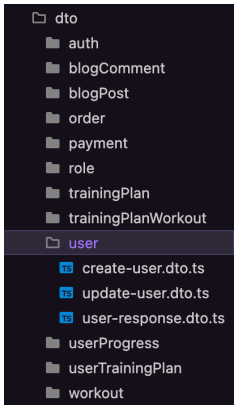
Контроллеры используют декораторы из библиотеки routing-controllers, такие как:

- @Get, @Post, @Patch, @Delete — маршруты
- @Param, @Body — извлечение параметров
- @Authorized — защита эндпоинтов
- @JsonController — привязка к префиксу маршрута
- @OpenAPI, @ResponseSchema — аннотации для Swagger-документации

```
@JsonController("/users")
export class UserController extends BaseController<User> {
  private readonly userService: UserService;

  constructor() {
    super(new UserService());
    this.userService = this.service as UserService;
  }

  @Get("/")
  @OpenAPI({ summary: "Get all users" })
  @ResponseSchema(UserResponseDto, { isArray: true })
  async getAll() {
    return this.userService.findAllWithRelations();
  }

  @Get("/id/:id")
  @OpenAPI({ summary: "Get user by ID" })
  @ResponseSchema(UserResponseDto)
  async getById(@Param("id") id: number) {
    const user = await this.userService.findOneWithRelations(id);
    if (!user) return { error: "User not found" };
    return user;
  }
```

```
  @Get("/email/:email")
  @OpenAPI({ summary: "Get user by email" })
  @ResponseSchema(UserResponseDto)
  async getByEmail(@Param("email") email: string) {
    const user = await this.userService.findByEmail(email);
    if (!user) return { error: "User not found" };
    return user;
  }

  @Post("/")
  @OpenAPI({ summary: "Create a new user" })
  @ResponseSchema(UserResponseDto)
  async create(@Body({ required: true }) userData: CreateUserDto) {
    return this.userService.create(userData);
  }

  @Patch("/:id")
  @OpenAPI({ summary: "Update a user" })
  @ResponseSchema(UserResponseDto)
  async update(@Param("id") id: number, @Body({ required: true }) updateData: UpdateUserDto) {
    return this.userService.update(id, updateData);
  }

  @Delete("/:id")
  @OpenAPI({ summary: "Delete a user" })
  async remove(@Param("id") id: number) {
    await this.userService.remove(id);
    return { message: "User deleted" };
  }
}
```

DTO



Папка **dto/** содержит классы, описывающие структуру входных и выходных данных для API. Необходимость использования DTO обусловлена удобством централизованной валидации данных, отделение слоя API от внутренней бизнес-логики, генерация схем OpenAPI (Swagger) на основе классов. DTO-классы разделены на три категории:

- create-*.dto.ts — для создания новых сущностей (POST)
- update-*.dto.ts — для частичного обновления (PATCH)
- *-response.dto.ts — формат возвращаемых данных (GET)

Каждое поле аннотировано декораторами из class-validator и class-transformer:

- @IsString, @IsInt, @IsOptional, @IsEmail, и т.д.
- @Type(() => ...) для преобразования типов

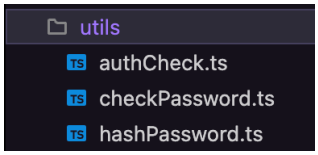
Пример UserDTO:

```ts
export class CreateUserDto {
  @IsString()
    @Type(() => String)
  name: string;

  @IsEmail()
    @Type(() => String)
  email: string;

  @IsString()
    @Type(() => String)
  password_hash: string;

  @IsOptional()
  @IsDateString()      You, 6 days ago • feat: Lab1, Lab2, HW3, HW4 are done …
  @Type(() => String)
  date_of_birth?: Date;

  @IsOptional()
    @Type(() => String)
  gender?: string;

  @IsOptional()
  @IsNumber()
  @Type(() => Number)
  role_id?: number;
}
```

```
You, 6 days ago | 1 author (You)
export class UpdateUserDto {
  @IsOptional()
  @IsString()
    @Type(() => String)
  name?: string;

  @IsOptional()
  @IsEmail()
    @Type(() => String)
  email?: string;

  @IsOptional()
  @IsString()
    @Type(() => String)
  password_hash?: string;

  @IsOptional()
  @IsDateString()
    @Type(() => String)
  date_of_birth?: Date;

  @IsOptional()
  @IsString()
    @Type(() => String)      You, 6 days ago • feat: Lab1, Lab2, HW3, HW4 are done …
  gender?: string;

  @IsOptional()
  role_id?: number;
}
```

```
export class UserResponseDto {
  @IsInt()
    @Type(() => Number)
  id: number;

  @IsString()
    @Type(() => String)
  name: string;

  @IsEmail()
    @Type(() => String)
  email: string;      You, 6 days ago • feat: Lab1, Lab2, HW3, HW4 are done …

  @IsOptional()
  @IsDateString()
    @Type(() => String)
  date_of_birth?: Date;

  @IsOptional()
  @IsString()
    @Type(() => String)
  gender?: string;

  @IsOptional()
  @IsInt()
    @Type(() => Number)
  role_id?: number;
}
```

Utils



Папка **utils/** содержит вспомогательные функции, используемые в разных частях проекта.

authCheck.ts:

- Реализация функции authorizationChecker для routing-controllers
- Проверяет JWT-токен в заголовке Authorization
- Распаковывает payload и прикрепляет пользователя к request.user
- Используется для защиты маршрутов через @Authorized

```
export const authorizationChecker = async (action: Action, roles: string[]) => {
  const authHeader = action.request.headers["authorization"];

  if (!authHeader) return false;

  const [type, token] = authHeader.split(" ");
  if (type !== "Bearer" || !token) return false;

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET || "secret");
    action.request.user = decoded;
    return true;       You, 6 days ago • feat: Lab1, Lab2, HW3, HW4 are done …
  } catch (err) {
    return false;
  }
};
```

hashPassword.ts / checkPassword.ts:

- Используют bcryptjs для безопасного хранения паролей
- hashPassword() применяется в UserService при создании
- checkPassword() — в AuthController при логине

```ts
import * as bcrypt from 'bcryptjs';

export const checkPassword = (hashedPassword: string, password: string): boolean => {
  return bcrypt.compareSync(password, hashedPassword);
};
```

```ts
export const hashPassword = (password: string): string => {
  return bcrypt.hashSync(password, bcrypt.genSaltSync(8));
};
```

Index.ts

Файл **src/index.ts** — это точка входа в приложение. В нём:

- Инициализируется подключение к базе данных через AppDataSource.initialize()
- Настраивается Express с помощью routing-controllers
- Загружаются все контроллеры проекта.
- Устанавливаются настройки CORS, валидации, обработка ошибок.
- Подключается authorizationChecker для @Authorized() маршрутов.
- Запускается сервера

```
const app = express();

AppDataSource.initialize()
  .then(() => {
    console.log("DB connected");

    useExpressServer(app, {
      controllers: [...
      ],
      routePrefix: "",
      cors: true,
      defaultErrorHandler: true,
      validation: {
        whitelist: true,
        forbidNonWhitelisted: true,
      },
      authorizationChecker,
      currentUserChecker: async (action) => {
        return action.request.user;
      },
    });

    useSwagger(app, {...
    });

    app.listen(3000, () => {
      console.log("🚀 Server running at http://localhost:3000");
    });
  })
  .catch((error) => console.log("DB connection error:", error));
```

Выводы

В рамках лабораторной работы №1 был разработан собственный проектный шаблон (boilerplate) на основе стека Express + TypeORM + TypeScript. Проект реализован с чёткой архитектурой и разделением по слоям:

- **Модели (Entities)** описывают структуру и связи таблиц базы данных;
- **Контроллеры (Controllers)** реализуют REST-маршруты и обрабатывают запросы;
- **Сервисы (Services)** инкапсулируют бизнес-логику и взаимодействуют с БД;
- **DTO** обеспечивают валидацию и формализацию входных и выходных данных;
- **Utils** реализуют повторно используемую логику, такую как авторизация и хеширование;
- **Общие компоненты (Common)** позволяют масштабировать проект с помощью абстракций.