**САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО**

**Дисциплина:** Бэк-энд разработка

Отчет

# Лабораторная работа №2

Выполнил:

Корчагин Вадим

Группа
К3341

Проверил:
Добряков Д. И.

Санкт-Петербург

2025 г.

# Тема

Платформа для фитнес-тренировок и здоровья.

# Задача

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

# Технологии:

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

# Ход работы

## CRUD’s

### Контроллеры с простым CRUD (без дополнительной логики)

Для этих сущностей реализованы стандартные операции:

- GET / — получить все записи
- GET /:id — получить по ID
- POST / — создать новую запись
- PATCH /:id — обновить запись
- DELETE /:id — удалить запись

Список моделей:

- Role (/roles)
- Workout (/workouts)
- TrainingPlan (/training-plans)
- TrainingPlanWorkout (/training-plan-workouts)
- UserProgress (/user-progress)
- Order (/orders)
- Payment (/payments)
- BlogPost (/blog-posts)
- BlogComment (/blog-comments)
- UserTrainingPlan (/user-training-plans)

### Контроллеры с дополнительной логикой

1. User (/users)

- GET /users/email/:email — получить пользователя по email
- POST /users — при создании пароль автоматически хешируется
- GET /users и GET /users/:id возвращают вложенные данные о роли
- Реализована проверка авторизации в отдельных эндпоинтах (@Authorized)

2. Auth (/auth)

- Реализован только один маршрут:
- POST /auth/login — вход по email и паролю
- Возвращается JWT-токен при успешной аутентификации
- Проверка пароля производится через bcrypt (checkPassword)
- Используется в связке с @Authorized() для защиты приватных маршрутов

## Особенности архитектуры

Проект построен по принципам слоистой архитектуры, с чётким разделением ответственности между слоями данных, логики и представления. Это позволяет добиться читаемости, переиспользуемости кода и лёгкой масштабируемости.

Разработанный API строго следует принципам REST и обеспечивает полный набор CRUD-операций для каждой бизнес-сущности.

1. Разделение логики между слоями

- Контроллеры обрабатывают входящие запросы и вызывают соответствующие методы сервисов.
- Сервисы реализуют логику работы с базой данных и связями между сущностями.
- DTO-классы используются для валидации данных в POST и PATCH запросах.

2. Единый стиль ответов

- Все ответы соответствуют REST-стандарту:
- При успешном выполнении возвращаются данные (200 OK, 201 Created)
- При удалении — сообщение об успешном удалении ({ message: "Deleted successfully" })
- В случае ошибки — объект с ключом error и описанием проблемы ({ error: "Not found" })

3. Аутентификация и защита маршрутов

- Вход по логину реализован через POST /auth/login
- Возвращается JWT-токен
- Защищённые маршруты (например, /payments, /orders) требуют Authorization: Bearer <token>

- Авторизация обрабатывается через декоратор @Authorized() и кастомный authorizationChecker

## Реализация REST API в коде

1. Контроллеры с routing-controllers

Контроллеры объявляются декларативно с помощью аннотаций @JsonController, @Get, @Post, @Patch, @Delete. Это упрощает конфигурацию маршрутов, устраняет необходимость вручную регистрировать каждый маршрут и обеспечивает единый стиль для всех HTTP-обработчиков.

2. Сервисы (src/services)

Сервисы содержат бизнес-логику и работают с базой данных. Все сервисы наследуются от BaseService, где реализованы базовые методы findAll, findOne, create, update, remove.

3. DTO (src/dto)

Все входные и выходные данные оформлены через DTO, которые:

- Валидируют структуру запроса (class-validator)
- Преобразуют типы (class-transformer)

Пример контроллера:

```
@JsonController("/user-progress")
export class UserProgressController extends BaseController<UserProgress> {
  private readonly userProgressService: UserProgressService;

  constructor() {
    super(new UserProgressService());
    this.userProgressService = this.service as UserProgressService;      You, 6 days ago • fea
  }

  @Get("/")
  @OpenAPI({ summary: "Get all user progress records" })
  @ResponseSchema(UserProgressResponseDto, { isArray: true })
  async getAll() {
    return this.userProgressService.findAllWithRelations();
  }

  @Get("/:id")
  @OpenAPI({ summary: "Get user progress by ID" })
  @ResponseSchema(UserProgressResponseDto)
  async getById(@Param("id") id: number) {
    const progress = await this.userProgressService.findOneWithRelations(id);
    if (!progress) {
      return { error: "Not found" };
    }
    return progress;
  }

  @Post("/")
  @OpenAPI({ summary: "Create user progress" })
  @ResponseSchema(UserProgressResponseDto)
  async create(@Body({ required: true }) data: CreateUserProgressDto) {
    return this.userProgressService.create(data);
  }
```

```
  @Patch("/:id")
  @OpenAPI({ summary: "Update user progress" })
  @ResponseSchema(UserProgressResponseDto)
  async update(@Param("id") id: number, @Body({ required: true }) data: UpdateUserProgressDto) {
    const updated = await this.userProgressService.update(id, data);
    if (!updated) {
      return { error: "Not found" };
    }
    return updated;
  }

  @Delete("/:id")
  @OpenAPI({ summary: "Delete user progress" })
  async remove(@Param("id") id: number) {
    await this.userProgressService.remove(id);
    return { message: "Deleted successfully" };
  }
}
```

# Выводы

REST API реализован по всем стандартам:

- Эндпоинты логично структурированы
- Ответы соответствуют статус-кодам
- Реализована защита приватных маршрутов
- API покрывает все сущности и операции