САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Бэк-энд разработка

Отчет

Лабораторная работа №1

Выполнил:

Кадникова Екатерина

Группа К3341

Проверил:
Добряков Д. И.

Санкт-Петербург

2025 г.

Задача

Нужно написать свой boilerplate на express + TypeORM + typescript.

Должно быть явное разделение на:

- модели
- контроллеры
- роуты

Ход работы

1. Модели

Были выделены две основные модели - User и Role. Модель Role представляет собой перечисление (enum), определяющее роли пользователей в системе (выделены роли - пользователь и админ). Модель User (см. Листинг 1) описывает сущность пользователя в базе данных и содержит информацию:

- id: Уникальный идентификатор пользователя (первичный ключ, автоинкрементируемый).
- username: Имя пользователя.
- email: Email пользователя, должен быть уникальным.
- password: Пароль пользователя.
- role: Роль пользователя в системе (по умолчанию присваивается роль USER).

Листинг 1 - Модель User:

```
import { Entity, PrimaryGeneratedColumn, Column } from 'typeorm';
import { Role } from './role';

@Entity('users')
export class User {
   @PrimaryGeneratedColumn()
   id!: number;

   @Column()
   username!: string;

   @Column({ unique: true })
   email!: string;

   @Column()
   password!: string;

   @Column({
```

```
            type: 'enum',
            enum: Role,
            default: Role.USER,
        })
        role!: Role;
    }
```

2. Контроллеры

В проекте реализованы два основных контроллера: AuthController и UserController.

Контроллер AuthController (см. Листинг 2) отвечает за регистрацию и авторизацию пользователей. Он взаимодействует с сервисом авторизации AuthService (см. раздел 3. Сервис авторизации).

- register(req: Request, res: Response) — регистрация нового пользователя.
- login(req: Request, res: Response) — вход в систему для зарегистрированного пользователя.

Листинг 2 - authController:

```
    import { Request, Response } from "express";
    import AuthService from "../services/authService";

    class AuthController {
        async register(req: Request, res: Response) {
            try {
                const { username, email, password } = req.body;
                const user = await AuthService.register(username, email,
password);
                res.status(201).json(user);
            } catch (error) {
                console.error(error);
                res.status(400).json({ message: (error as Error).message });
            }
        }

        async login(req: Request, res: Response) {
            try {
                const { email, password } = req.body;
                const data = await AuthService.login(email, password);
                res.status(200).json(data);
            } catch (error) {
                console.error(error);
```

```
                res.status(400).json({ message: (error as Error).message });
            }
        }
    }

    export default new AuthController();
```

Контроллер UserController (см. Листинг 3) отвечает за базовые операции работы с пользователями:

- getAllUsers(req, res) — получить всех пользователей.
- getUserById(req, res) — получить пользователя по ID.
- getUserByUsernameOrEmail(req, res) — получить пользователя по username или email.
- updateUser(req, res) — обновить данные пользователя по ID.
- deleteUser(req, res) — удалить пользователя по ID.

Листинг 3 - userController:

```
    import { Request, Response } from "express";
    import { AppDataSource } from "../data-source";
    import { User } from "../models/user";

    const userRepository = AppDataSource.getRepository(User);

    export const getAllUsers = async (req: Request, res: Response):
Promise<Response> => {
        try {
            const users = await userRepository.find();
            return res.json(users);
        } catch (error) {
            return res.status(500).json({ message: "Error fetching users" });
        }
    };

    export const getUserById = async (req: Request, res: Response):
Promise<Response> => {
        try {
            const { id } = req.params;
            const user = await userRepository.findOneBy({ id: Number(id) });

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

            return res.json(user);
        } catch (error) {
             return res.status(500).json({ message: "Error fetching user by ID"
});
        }
    };
```

```
      export  const  getUserByUsernameOrEmail  =  async  (req:  Request,  res:
Response): Promise<Response> => {
         try {
             const { username, email } = req.query;

             if (!username && !email) {
                    return res.status(400).json({ message: "Provide username or
email" });
             }

             const user = await userRepository.findOneBy(
                 username
                     ? { username: String(username) }
                     : { email: String(email) }
             );

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

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

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

             const user = await userRepository.findOneBy({ id: Number(id) });

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

             userRepository.merge(user, data);
             const updatedUser = await userRepository.save(user);

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

      export  const  deleteUser  =  async  (req:  Request,  res:  Response):
Promise<Response> => {
         try {
             const { id } = req.params;
```

```
            const result = await userRepository.delete(Number(id));

            if (result.affected === 0) {
                return res.status(404).json({ message: "User not found" });
            }

            return res.json({ message: "User deleted successfully" });
        } catch (error) {
            return res.status(500).json({ message: "Error deleting user" });
        }
    };
```

3. Сервис авторизации

Для реализации логики регистрации и авторизации пользователей был создан отдельный сервис AuthService, а также утилита для работы с JWT (см. Листинг 5) и middleware для аутентификации.

AuthService (см. Листинг 4) реализует бизнес-логику, связанную с созданием новых пользователей и их последующей авторизацией.

- register(username: string, email: string, password: string)

— Регистрирует нового пользователя:

  - Проверяет наличие пользователя с данным email.
  - Хеширует пароль с помощью bcrypt.
  - Сохраняет нового пользователя в базу данных.
  - Возвращает основные данные о пользователе без пароля.

- login(email: string, password: string)

— Выполняет вход пользователя:

  - Проверяет наличие пользователя с указанным email.
  - Сверяет переданный пароль с сохраненным хешем.
  - В случае успеха генерирует JWT-токен для авторизации.

Листинг 4 - authService:

```
    import { AppDataSource } from "../data-source";
    import { User } from "../models/user";
    import bcrypt from "bcryptjs";
    import { generateToken } from "../utils/jwt";

    class AuthService {
        private userRepository = AppDataSource.getRepository(User);

        async register(username: string, email: string, password: string) {
            const existingUser = await this.userRepository.findOne({ where: {
email } });
            if (existingUser) {
```

```
                throw new Error("User already exists");
            }

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

            const user = this.userRepository.create({
                username,
                email,
                password: hashedPassword,
            });

            await this.userRepository.save(user);

            return {
                id: user.id,
                username: user.username,
                email: user.email,
                role: user.role,
            };
        }

        async login(email: string, password: string) {
             const user = await this.userRepository.findOne({ where: { email } });

            if (!user) {
                throw new Error("Invalid credentials");
            }

                    const isPasswordValid = await bcrypt.compare(password, user.password);

            if (!isPasswordValid) {
                throw new Error("Invalid credentials");
            }

            const token = generateToken(user);

            return { token };
        }
    }

    export default new AuthService();
```

Для управления JWT-токенами используется модуль jsonwebtoken. В проекте реализованы функции:

- generateToken(user: User) — создает токен, содержащий id, email и роль пользователя. Срок действия токена указывается в .env файле через переменные JWT_SECRET и JWT_EXPIRES_IN.

- verifyToken(token: string) — проверяет валидность токена и возвращает его содержимое.

Листинг 5 - Утилита jwt:

```
import jwt, { SignOptions } from "jsonwebtoken";
import { User } from "../models/user";
import * as dotenv from "dotenv";

dotenv.config();

const JWT_SECRET = process.env.JWT_SECRET as string;
const JWT_EXPIRES_IN = process.env.JWT_EXPIRES_IN ?
parseInt(process.env.JWT_EXPIRES_IN) : 3600;

if (!JWT_SECRET) {
   throw new Error("JWT_SECRET is not defined in environment variables");
}

interface TokenPayload {
   id: number;
   email: string;
   role: string;
}

export const generateToken = (user: User): string => {
   const payload: TokenPayload = {
       id: user.id,
       email: user.email,
       role: user.role,
   };

   const options: SignOptions = {
       expiresIn: JWT_EXPIRES_IN,
   };

   return jwt.sign(payload, JWT_SECRET, options);
};

export const verifyToken = (token: string): TokenPayload => {
   try {
       return jwt.verify(token, JWT_SECRET) as TokenPayload;
   } catch (error) {
       throw new Error("Invalid or expired token");
   }
};
```

Для проверки наличия и валидности токена на защищенных маршрутах реализован middleware authenticateToken.

- Извлекает токен из заголовка Authorization.
- Проверяет его валидность через verifyToken.

- При успешной проверке добавляет информацию о пользователе (id, email, role) в объект запроса (req.user).
- В случае ошибок отправляет ответ с кодом 401 Unauthorized.

Листинг 6 - authMiddleware:

```
import { Request, Response, NextFunction } from "express";
import { verifyToken } from "../utils/jwt";

export interface AuthenticatedRequest extends Request {
   user?: {
       id: number;
       email: string;
       role: string;
   };
}

export const authenticateToken = (
   req: AuthenticatedRequest,
   res: Response,
   next: NextFunction
) => {
   const authHeader = req.headers.authorization;

   if (!authHeader || !authHeader.startsWith("Bearer ")) {
            return res.status(401).json({ message: "Authorization header
missing or malformed" });
   }

   const token = authHeader.split(" ")[1];

   try {
       const user = verifyToken(token);
       req.user = user;
       next();
   } catch (error) {
        return res.status(401).json({ message: "Invalid or expired token"
});
   }
};
```

4. Роуты

Было реализовано четкое разделение роутов для обработки запросов, связанных с аутентификацией пользователей и управлением данными пользователей. Все роуты сгруппированы по своей тематике и подключаются через отдельные модули.

Файл authRoutes.ts (см. Листинг 7) отвечает за регистрацию новых пользователей и вход в систему.

Основные маршруты:

- POST /api/auth/register — регистрация нового пользователя.
- POST /api/auth/login — авторизация пользователя и получение токена.

Листинг 7 - authRoutes:

```
import { Router } from "express";
import AuthController from "../controllers/authController";

const authRouter = Router();

authRouter.post("/register", AuthController.register);
authRouter.post("/login", AuthController.login);

export default authRouter;
```

Файл userRoutes.ts (см. Листинг 8) реализует набор маршрутов для работы с пользователями: получение списка пользователей, поиск, обновление и удаление.

Основные маршруты:

- GET /api/users/ — получить список всех пользователей.
- GET /api/users/find — найти пользователя по username или email (через query параметры).
- GET /api/users/:id — получить пользователя по ID.
- PUT /api/users/:id — обновить данные пользователя по ID.
- DELETE /api/users/:id — удалить пользователя по ID.

Листинг 8 - userRoutes:

```
import { Router } from "express";
import {
   getAllUsers,
   getUserById,
   getUserByUsernameOrEmail,
   updateUser,
   deleteUser
} from "../controllers/userController";
import { asyncHandler } from "../utils/asyncHandler";

const router = Router();

router.get("/", asyncHandler(getAllUsers));
router.get("/find",   asyncHandler(getUserByUsernameOrEmail));   //   через
query ?username=... или ?email=...
router.get("/:id", asyncHandler(getUserById));
```

```
router.put("/:id", asyncHandler(updateUser));
router.delete("/:id", asyncHandler(deleteUser));

export default router;
```

5. Middleware

В проекте реализованы дополнительные middleware-функции для проверки конфигурации окружения, обработки ошибок и логирования запросов.

Для корректной работы приложения требуется наличие определенного набора переменных окружения (.env файл). Middleware checkEnvVariables автоматически проверяет их наличие при старте приложения.

- Задает список обязательных переменных.
- Проверяет, присутствуют ли они в process.env.
- Если каких-то переменных не хватает, приложение выводит ошибку и аварийно завершает работу (process.exit(1)).

Middleware errorMiddleware перехватывает все необработанные ошибки, возникшие в процессе обработки запроса, и отправляет клиенту стандартный ответ.

- Логирует ошибку в консоль.
- Отправляет клиенту JSON-ответ с кодом 500 Internal Server Error.
- Если ошибка является экземпляром класса Error, возвращает её сообщение; в противном случае — общее сообщение об ошибке.

Middleware logRequestMiddleware предназначен для удобного отслеживания всех приходящих запросов.

- Выводит в консоль информацию о каждом запросе: метод, URL, тело запроса и параметры запроса (query).
- Форматирует лог с привязкой к времени выполнения.

Вывод

В рамках работы был собран базовый проект на Express, TypeORM и TypeScript с правильной структурой — отдельно модели, контроллеры, роуты и сервисы. Реализована регистрация, авторизация пользователей и базовые

CRUD-операции с пользователями. Также добавлены полезные middleware для логирования, проверки окружения и обработки ошибок.