

**САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ  
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО**

**Дисциплина:** Бэк-энд разработка

Отчет

Лабораторная работа 2: Реализация REST API на основе  
boilerplate

Выполнил:

Лазебный Всеволод

Группа К3344

Проверил:

Добряков Д. И.

Санкт-Петербург

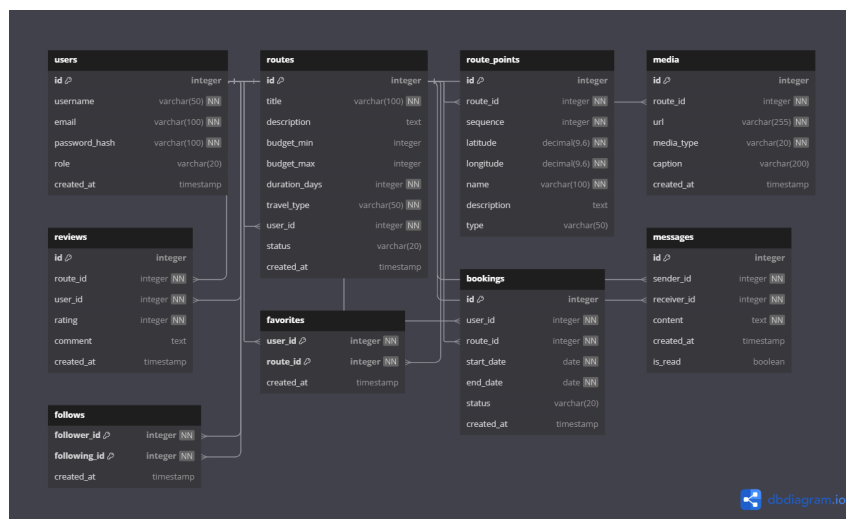
2025 г.

## Задача

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

## Ход работы

В рамках Д31 была спроектирована следующая схема данных:



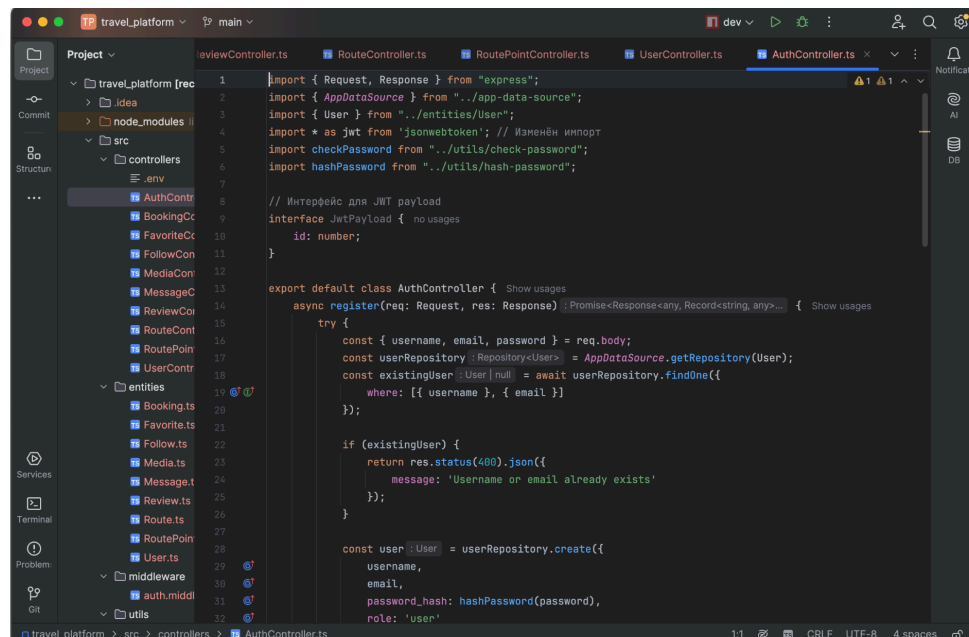
Также в рамках Д33 были успешно протестированы все CRUD-запросы через Postman. Результаты можно найти в отчете по Д33. В рамках лабораторной работы была добавлена логика реализации регистрации, входа, хэширования, проверка пароля и получение токена.

Ниже представлены файлы hashPassword.ts и checkPassword.ts

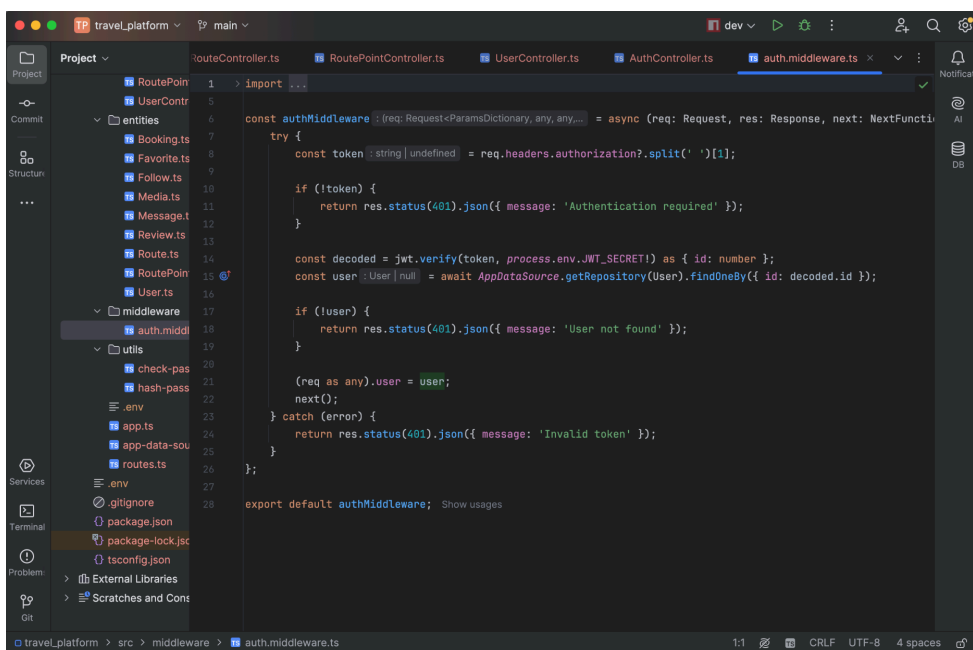
```
1 import bcrypt from 'bcryptjs';
2
3 const hashPassword : (password: string) => string = (password: string): string => { Show usages
4   return bcrypt.hashSync(password, bcrypt.genSaltSync(10));
5 };
6
7 export default hashPassword; Show usages
```

```
1 import bcrypt from 'bcryptjs';
2
3 const checkPassword : (password: string, hashedPassword: string...) => boolean = (password: string, hashedPassword: string): boolean => {
4   return bcrypt.compareSync(password, hashedPassword);
5 };
6
7 export default checkPassword; Show usages
```

Все нужные запросы доступны благодаря входу и токену, там, где нужно требуется вход пользователя



```
1 import { Request, Response } from "express";
2 import { AppDataSource } from "../app-data-source";
3 import { User } from "../entities/User";
4 import * as jwt from 'jsonwebtoken'; // Именован импорт
5 import checkPassword from "../utils/check-password";
6 import hashPassword from "../utils/hash-password";
7
8 // Интерфейс для JWT payload
9 interface JwtPayload {
10   id: number;
11 }
12
13 export default class AuthController {
14   async register(req: Request, res: Response): Promise<Response<any, Record<string, any>...> {
15     try {
16       const { username, email, password } = req.body;
17       const userRepository : Repository<User> = AppDataSource.getRepository(User);
18       const existingUser : User | null = await userRepository.findOne({
19         where: [{ username }, { email }]
20       });
21
22       if (existingUser) {
23         return res.status(400).json({
24           message: 'Username or email already exists'
25         });
26       }
27
28       const user : User = userRepository.create({
29         username,
30         email,
31         password_hash: hashPassword(password),
32         role: 'user'
33       });
```



```
1 import { Request, Response, NextFunction } from "express";
2
3 const authMiddleware : (req: Request<ParamsDictionary, any, any>, res: Response, next: NextFunction) => Promise<void> = async (req: Request, res: Response, next: NextFunction) => {
4   try {
5     const token : string | undefined = req.headers.authorization?.split(' ')[1];
6
7     if (!token) {
8       return res.status(401).json({ message: 'Authentication required' });
9     }
10
11     const decoded = jwt.verify(token, process.env.JWT_SECRET!) as { id: number };
12     const user : User | null = await AppDataSource.getRepository(User).findOneBy({ id: decoded.id });
13
14     if (!user) {
15       return res.status(401).json({ message: 'User not found' });
16     }
17
18     (req as any).user = user;
19     next();
20   } catch (error) {
21     return res.status(401).json({ message: 'Invalid token' });
22   }
23 };
24
25 export default authMiddleware;
```

## Вывод

В ходе этой лабораторной была реализована новая логика: хэширование и проверка пароля.