САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Бэк-энд разработка

Отчет

Лабораторная работа №1

Выполнила:
Лапшина Екатерина
Алексеевна
Группа К3340

Проверил:
Добряков Д. И.

Санкт-Петербург

2025 г.

Задача

Создать boilerplate на Express.js + TypeORM + TypeScript с явным разделением на:
- Модели (entities)
- Контроллеры (controllers)
- Роуты (routes)

Ход работы

1. Структура проекта

Проект организован по принципу разделения ответственности (Separation of Concerns) со следующей структурой:

```
src/
├── config/
│   └── database.ts          # Конфигурация TypeORM
├── controllers/
│   ├── BaseController.ts    # Базовый контроллер
│   └── userController.ts    # Логика работы с пользователями
├── entities/
│   ├── Base.ts             # Базовая
│   └── User.ts             # Модель пользователя
├── middleware/
│   └── auth.ts             # Middleware для аутентификации
├── routes/
│   ├── auth.ts             # Маршруты аутентификации
│   ├── users.ts            # Маршруты пользователей
└── index.ts                # Точка входа приложения
```

2. Настройка зависимостей

В `package.json` определены основные зависимости:

```json
{
  "dependencies": {
    "express": "^4.18.2",
    "typeorm": "^0.3.17",
    "sqlite3": "^5.1.6",
    "typescript": "^5.3.2",
    "jsonwebtoken": "^9.0.2",
    "bcryptjs": "^2.4.3",
    "class-validator": "^0.14.0",
    "swagger-jsdoc": "^6.2.8",
    "swagger-ui-express": "^5.0.1"
  }
}
```

3. Конфигурация базы данных

В файле `src/config/database.ts` настроено подключение к БД через TypeORM:

```
export const AppDataSource = new DataSource({
  type: "sqlite",
  database: process.env.DATABASE_PATH || "./database.sqlite",
  synchronize: process.env.NODE_ENV !== "production", // Auto-sync in development
  logging: process.env.NODE_ENV === "development",
  entities: [
    // Add your models here
    User,
    // Example,
  ],
  subscribers: [],
  migrations: [],
});

export const initializeDatabase = async () => {
  try {
    await AppDataSource.initialize();
    console.log("✅ Database connection established successfully");

    if (process.env.NODE_ENV === "development") {
      console.log("📊 Database synchronized");
    }
  } catch (error) {
    console.error("❌ Error connecting to database:", error);
    process.exit(1);
  }
};

export const closeDatabase = async () => {
  try {
    await AppDataSource.destroy();
    console.log("🔌 Database connection closed");
  } catch (error) {
    console.error("❌ Error closing database connection:", error);
  }
};
```

4. Модели (Entities)

С помощью TypeORM реализована базовая сущность, в которой настроены сервисные поля (id, createdAt, updatedAt, deletedAt)

```
export abstract class BaseModel {
  @PrimaryGeneratedColumn()
  id!: number;

  @CreateDateColumn({ name: 'created_at' })
  createdAt!: Date;

  @UpdateDateColumn({ name: 'updated_at' })
  updatedAt!: Date;

  @DeleteDateColumn({ name: 'deleted_at', nullable: true })
  deletedAt?: Date;
}
```

На её основе подготовлена модель пользователя с использованием TypeORM декораторов:

```
@Entity("users")
export class User extends BaseModel {
  @Column({ length: 100 })
  @Index()
  firstName!: string;

  @Column({ length: 100 })
  @Index()
  lastName!: string;

  @Column({ unique: true, length: 255 })
  @Index()
  @IsEmail()
  email!: string;

  @Column({ length: 255 })
  @MinLength(6)
  password!: string;

  @Column({
    type: "varchar",
    length: 20,
    default: "user",
    enum: ["admin", "user", "moderator"]
  })
  role!: string;

  @Column({ default: true })
  isActive!: boolean;

  @Column({ nullable: true })
  lastLoginAt?: Date;

  @BeforeInsert()
  @BeforeUpdate()
  async hashPassword() {
    if (this.password && this.password.length < 60) {
      this.password = await bcrypt.hash(this.password, 12);
    }
  }

  async comparePassword(candidatePassword: string): Promise<boolean> {
    return bcrypt.compare(candidatePassword, this.password);
  }

  get fullName(): string {
    return `${this.firstName} ${this.lastName}`;
  }
}
```

5. Контроллеры (Controllers)

Реализован базовый контроллер, отвечающий за работу с БД, пагинацию и обработку ошибок. На его основе реализован CRUD-контроллер для пользователей.

```
// Get all with pagination
protected async getAll(
  req: Request,
  res: Response,
  next: NextFunction,
  where?: FindOptionsWhere<T>
): Promise<void> {
  try {
    const { page = 1, limit = 10, sortBy = 'id', sortOrder = 'DESC' } = req.query as PaginationParams;

    const skip = (Number(page) - 1) * Number(limit);

    const [data, total] = await this.repository.findAndCount({
      where,
      skip,
      take: Number(limit),
      order: { [sortBy]: sortOrder } as any
    });

    const totalPages = Math.ceil(total / Number(limit));

    const response: PaginatedResponse<T> = {
      success: true,
      message: 'Data retrieved successfully',
      data,
      pagination: {
        page: Number(page),
        limit: Number(limit),
        total,
        totalPages
      }
    };

    res.status(200).json(response);
  } catch (error) {
    next(new AppError('Failed to retrieve data', 500, ErrorType.DATABASE_ERROR));
  }
}

// Get by ID
protected async getById(
  req: Request,
  res: Response,
  next: NextFunction
): Promise<void> {
  try {
    const { id } = req.params;

    const data = await this.repository.findOne({
      where: { id: Number(id) } as any
    });

    if (!data) {
      throw new AppError('Record not found', 404, ErrorType.NOT_FOUND_ERROR);
    }

    const response: ApiResponse<T> = {
      success: true,
      message: 'Data retrieved successfully',
      data
    };

    res.status(200).json(response);
  } catch (error) {
    if (error instanceof AppError) {
      next(error);
    } else {
      next(new AppError('Failed to retrieve data', 500, ErrorType.DATABASE_ERROR));
    }
  }
}
```

На его основе реализован контроллер CRUD-операций модели пользователя.

6. Роуты (Routes)

Маршруты организованы по модульному принципу с использованием Express Router

```
const router = Router();
const userController = new UserController();

router.get('/', authenticateToken, requireAdmin, userController.getAllUsers);
router.get('/:id', authenticateToken, requireUser, userController.getUserById);
router.post('/', authenticateToken, requireAdmin, userController.createUser);
router.put('/:id', authenticateToken, requireAdmin, userController.updateUser);
router.delete('/:id', authenticateToken, requireAdmin, userController.deleteUser);
router.get('/profile/me', authenticateToken, userController.getProfile);

export default router;
```

7. Middleware аутентификации

Реализован JWT-based middleware для защиты маршрутов:

```
export const authenticateToken = async (
  req: AuthenticatedRequest,
  res: Response,
  next: NextFunction
): Promise<void> => {
  try {
    const authHeader = req.headers.authorization;
    const token = authHeader && authHeader.split(' ')[1]; // Bearer TOKEN

    if (!token) {
      throw new AppError('Access token required', 401, ErrorType.AUTHENTICATION_ERROR);
    }

    const decoded = jwt.verify(token, process.env.JWT_SECRET || 'fallback-secret') as any;

    const userRepository = AppDataSource.getRepository(User);
    const user = await userRepository.findOne({
      where: { id: decoded.userId, isActive: true }
    });

    if (!user) {
      throw new AppError('User not found or inactive', 401, ErrorType.AUTHENTICATION_ERROR);
    }

    req.user = {
      id: user.id,
      email: user.email,
      role: user.role
    };

    next();
  } catch (error) {
    if (error instanceof jwt.JsonWebTokenError) {
      next(new AppError('Invalid token', 401, ErrorType.AUTHENTICATION_ERROR));
    } else if (error instanceof AppError) {
      next(error);
    } else {
      next(new AppError('Authentication failed', 401, ErrorType.AUTHENTICATION_ERROR));
    }
  }
};
```

Вывод

В ходе выполнения лабораторной работы был успешно создан boilerplate на Express.js + TypeORM + TypeScript с четким разделением на модели, контроллеры и роуты.

Достигнутые результаты:

1. Архитектурное разделение: Реализовано четкое разделение ответственности между слоями приложения (entities, controllers, routes)
2. TypeORM интеграция: Настроена работа с базой данных SQLite через TypeORM с автоматической синхронизацией схемы
3. Аутентификация: Реализована JWT-based аутентификация с middleware для защиты маршрутов
4. REST API: Создан полный набор CRUD операций для всех сущностей системы
5. Безопасность: Реализовано хеширование паролей с помощью bcryptjs
6. Обработка ошибок: Настроена централизованная обработка ошибок
7. Валидация: Использованы TypeORM декораторы для валидации данных

Технологический стек: - Node.js + Express.js - TypeScript - TypeORM - SQLite - JWT для аутентификации

Проект готов к использованию как основа для разработки веб-приложений с REST API и может быть легко расширен дополнительной функциональностью.