

Аутентификация и авторизация в NestJS с сохранением данных в базу данных с помощью Prisma

Мандронов Антон Андреевич

Приамурский государственный университет имени Шолом-Алейхема

Студент

Аннотация

В данной работе рассматривается разработка и анализ механизма аутентификации и авторизации в NestJS с использованием JWT-токенов, Prisma ORM и MySQL. Исследование направлено на создание безопасной и масштабируемой системы управления доступом на основе ролевой модели. Применяются современные инструменты, такие как Passport.js, bcrypt и class-validator, для обеспечения защиты данных и валидации запросов. Проведено тестирование API через Postman, подтвердившее корректность работы системы. Результаты демонстрируют эффективность предложенного подхода для веб-приложений и микросервисных архитектур.

Ключевые слова: NestJS, аутентификация, авторизация, JWT, Prisma, MySQL, Passport.js, bcrypt, RBAC, Postman, API-безопасность, ORM, микросервисы.

Authentication and authorization in NestJS with saving data to the database using Prisma

Mandronov Anton Andreevich

Sholom-Aleichem Priamursky State University

Student

Abstract

This study explores the development and analysis of an authentication and authorization mechanism in NestJS using JWT tokens, Prisma ORM, and MySQL. The research focuses on creating a secure and scalable access control system based on a Role-Based Access Control model. Modern tools such as Passport.js, bcrypt, and class-validator are employed to ensure data protection and request validation. API testing via Postman confirmed the system's correct functionality. The results demonstrate the effectiveness of the proposed approach for web applications and microservice architectures.

Keywords: NestJS, authentication, authorization, JWT, Prisma, MySQL, Passport.js, bcrypt, RBAC, Postman, API security, ORM, microservices.

1 Введение

1.1 Актуальность

С развитием веб-приложений и микросервисных архитектур вопросы безопасности доступа к API становятся критически важными. Аутентификация (подтверждение личности пользователя) и авторизация (проверка прав доступа) — ключевые механизмы защиты данных от несанкционированного доступа. Фреймворк NestJS набирает популярность благодаря своей модульности, поддержке TypeScript и интеграции с современными инструментами, такими как Prisma (ORM для работы с БД). Однако, несмотря на обилие документации, многие разработчики сталкиваются с трудностями при реализации надежной системы аутентификации и авторизации, особенно при работе с базами данных. Использование JWT (JSON Web Tokens) для управления сессиями и ролевой модели доступа (RBAC) позволяет создавать масштабируемые и безопасные API. Однако некорректная реализация может привести к уязвимостям, таким как утечки паролей из-за отсутствия хеширования, недостаточная валидация токенов, отсутствие контроля доступа на уровне маршрутов.

1.2 Обзор исследований

В работе А. Н. Мытникова и В. Р. Анохина анализируются встроенные механизмы информационной безопасности в NestJS, предлагаются методы их эффективного применения. Их подходы помогают разработчикам повысить защищенность веб-приложений [1]. В. А. Долженко изучает архитектуру микросервисов в финансовых системах, построенных на NestJS, выделяя преимущества этого фреймворка. В исследовании рассматриваются способы оптимизации взаимодействия между сервисами [2]. А. В. Лакшинских и Д. К. Давыденко разрабатывают веб-приложение для работы с историческими материалами, используя React и NestJS. Они описывают ключевые этапы проектирования и реализации [3]. В работе С. Н. Девицыной, П. В. Пилькевича и Е. В. Удада исследуются уязвимости JWT-токенов и предлагают способы усиления их защиты. В работе представлены практические рекомендации по предотвращению атак [4].

1.3 Цель исследования

Целью данной работы является разработка и анализ надежного механизма аутентификации и авторизации в NestJS с сохранением данных в базе данных через Prisma.

2 Материалы и методы

Исследование проводилось в среде разработки Visual Studio Code (VS Code) с использованием следующих технологий: Node.js – среда выполнения JavaScript; NestJS – фреймворк для построения серверных приложений; Prisma – ORM для работы с реляционными базами данных; MySQL – СУБД для хранения пользовательских данных; JWT (JSON Web Tokens) – механизм аутентификации через токены, Passport.js — промежуточное программное

обеспечение для авторизации, bcrypt – библиотека для хеширования паролей. Запросы на авторизацию и аутентификацию проверялись в сервисе для тестирования API Postman. Для запуска сервера использовалась портативная программная среда для веб-разработки Open Server Panel.

3 Результаты и обсуждения

Перед созданием проекта на NestJS необходимо установить NodeJS и npm с официального сайта nodejs.org. Node.js — это кроссплатформенная среда выполнения JavaScript, которая позволяет разработчикам запускать код на сервере, а не только в браузере. Используется при разработке высоконагруженных веб-приложений, стриминговых онлайн-сервисов, видеоигр. npm (Node Package Manager) — это система управления пакетами для Node.js. Предоставляет огромную экосистему библиотек и инструментов с открытым исходным кодом, которые разработчики могут легко интегрировать в свои проекты. Упрощает процесс управления зависимостями, обмена кодом и сборки приложений Node.js. Чтобы проверить, что установка прошла успешно, в терминале vscode введём команды `node -v` и `npm -v` (рис.1).

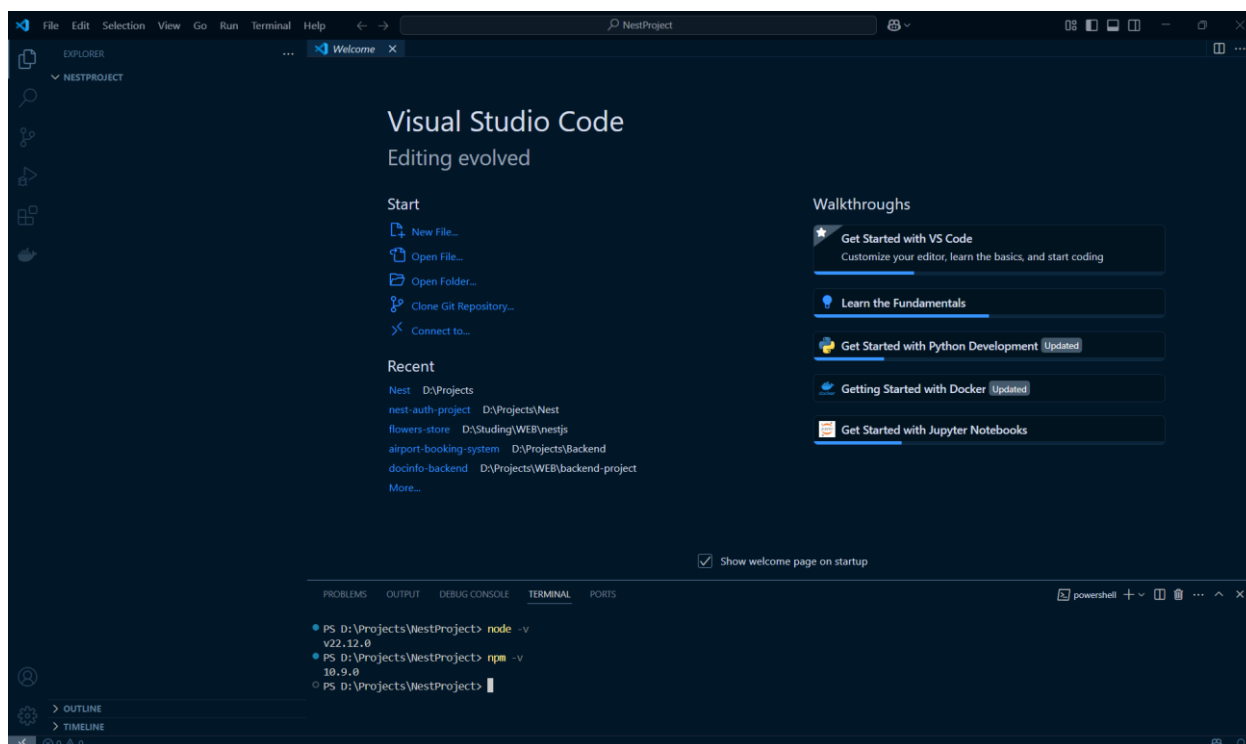


Рис 1. Проверка установки NodeJS и npm

Установим фреймворк NestJS через команду «*npm install -g @nestjs/cli*». Ключ *-g* означает глобальную установку, которая позволяет использовать команды *nest* не только в текущем проекте, но и в любом другом, использующим NodeJS. Создадим проект с помощью *nest new nest-auth-prisma*. На вопрос «Which package manager would you like to use?» отвечаем «*npm*». После создания проекта переместимся в него через «*cd nest-auth-*

prisma» и установим все необходимые зависимости «*npm install @nestjs/passport passport passport-jwt bcrypt @nestjs/config*». Для работы с базой данных была использована Prisma. Prisma — это ORM нового поколения для Node.js и TypeScript. Это инструмент, позволяющий работать с реляционными (PostgreSQL, MySQL, SQL Server, SQLite) и нереляционной (MongoDB) базами данных с помощью JavaScript или TypeScript без использования SQL. Устанавливаем *prisma* «*npm install @prisma/client prisma -D*» и инициализируем в проект «*npx prisma init*». В созданном файле *prisma/schema.prisma* определяем модель пользователя (рис. 2). В модели будет четыре основных поля: email, пароль, роль (обычный пользователь или админ), дата и время создания профиля. Сменим также тип используемой базы данных с *postgres* на *mysql*. Также в файле *.env* необходимо поменять *DATABASE_URL* с «*postgresql://johndoe:randompassword@localhost:5432/mydb?schema=public*» на «*mysql://root:@localhost:3306/mydb?schema=public*».

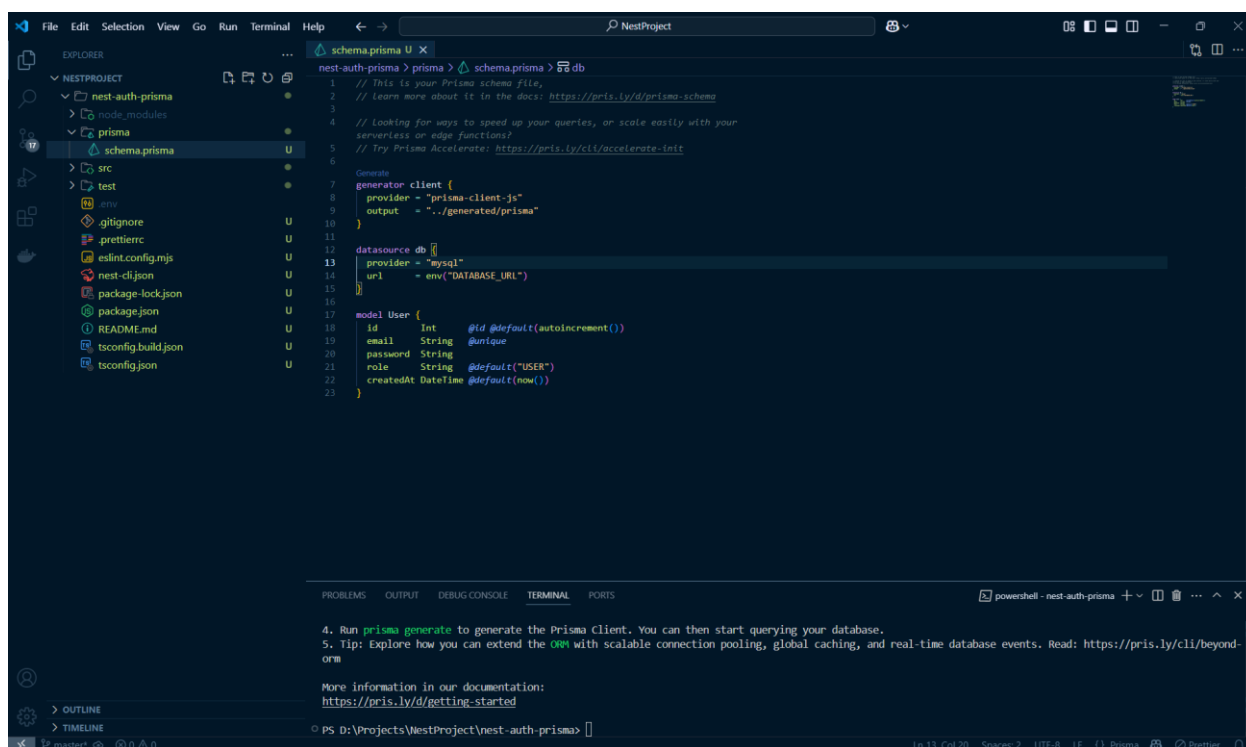


Рис 2. Модель пользователя в *schema.prisma*

Запускаем сервер на Open Server Panel, выполняем миграцию «*npx prisma migrate dev --name init*» и создаём клиента «*npx prisma generate*». Выполнение миграции в Prisma необходимо для управления изменениями схемы базы данных. Создание клиента Prisma необходимо, чтобы предоставить программный интерфейс приложения (API) для работы с моделями и типы для TypeScript. После генерации клиента его можно импортировать в код и использовать для отправки запросов. Создаём файлы *prisma.service.ts* (рис. 3) для подключения к базе данных и *prisma.module.ts* (рис. 4) для возможности использования подключения в других скриптах.

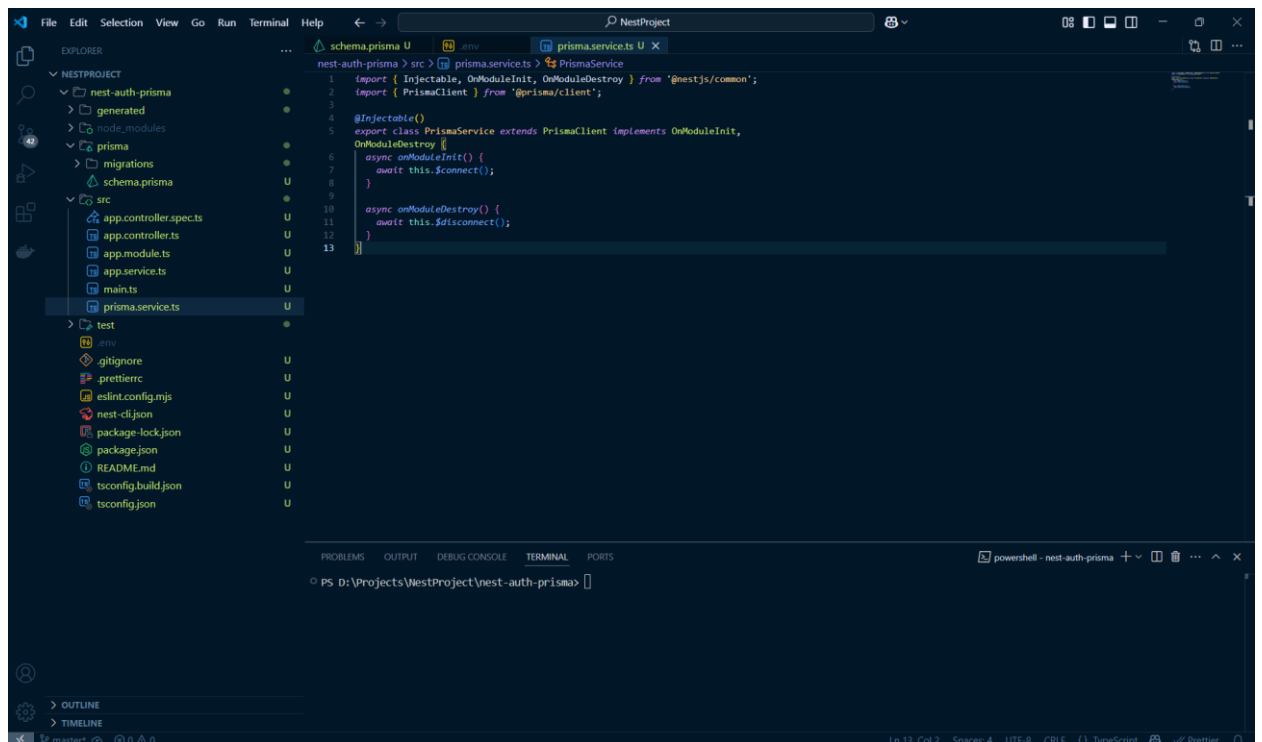


Рис 3. Prisma.service.ts

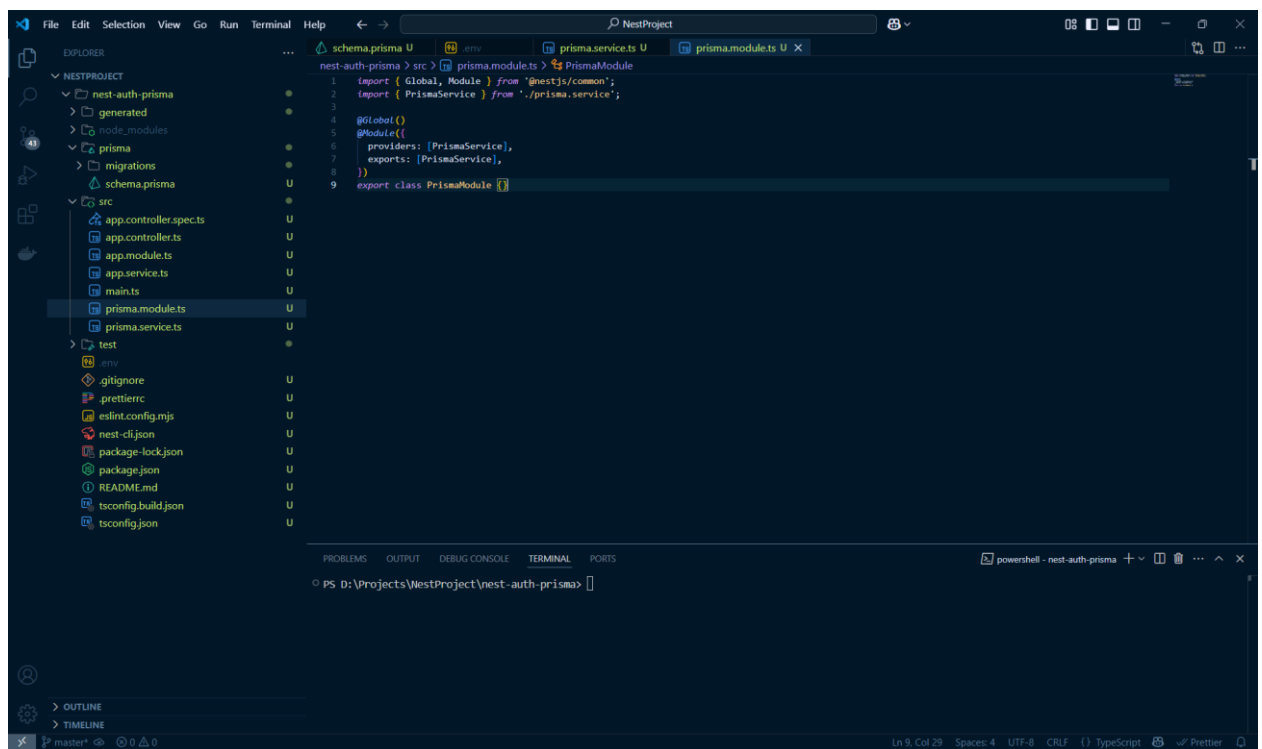


Рис 4. Prisma.module.ts

Создаём папку src/auth, где будут храниться скрипты для авторизации и аутентификации. Теперь в этой папке создаём папку dto, где будут храниться login.dto.ts (рис. 5) и register.dto.ts (рис. 6). DTO-файлы в NestJS нужны для контроля и проверки данных при создании или обновлении ресурсов в приложении, гарантируя, что данные соответствуют определённым правилам

и ограничениям. Для обоих файлов импортируем class-validator для проверки типов полей через «*npm install class-validator*».

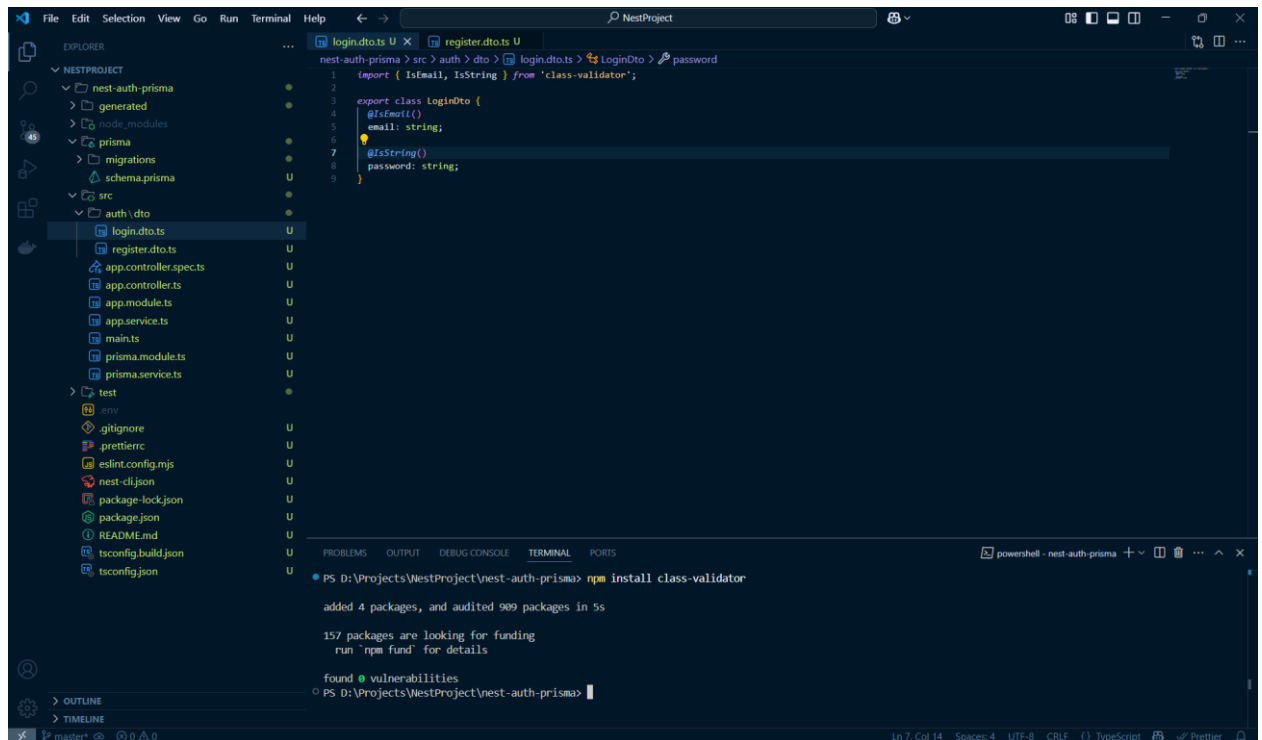


Рис 5. Login.dto.ts

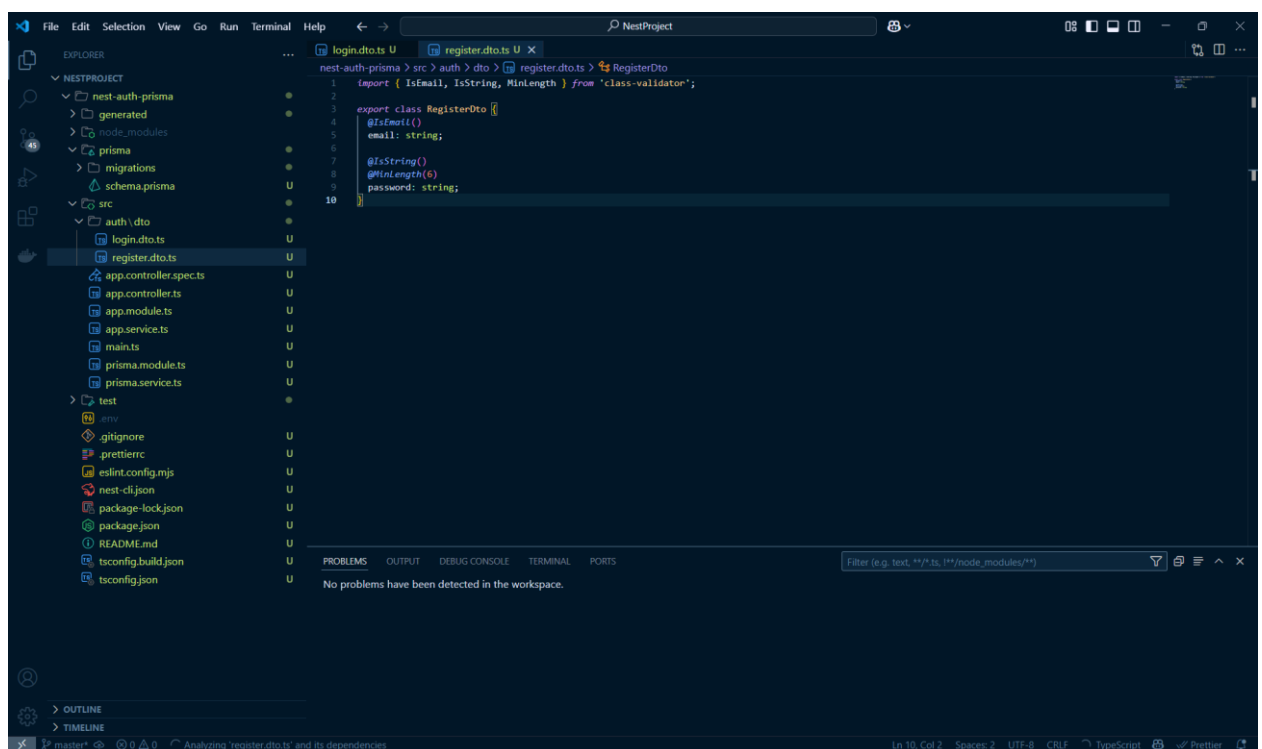


Рис 6. Register.dto.ts

Создаём папку `auth/guards`, где будут храниться `jwt-auth.guard.ts` (рис. 7), который проверяет JWT-токен пользователя и пропускает дальше только авторизованных, `roles.decorator.ts` (рис. 8), с помощью которого указывается,

какие роли требуются для доступа к конкретному endpoint'у, и roles.guard.ts (рис. 9), проверяющий права доступа пользователя в зависимости от его роли. Для аутентификации будет использоваться Passport.js.

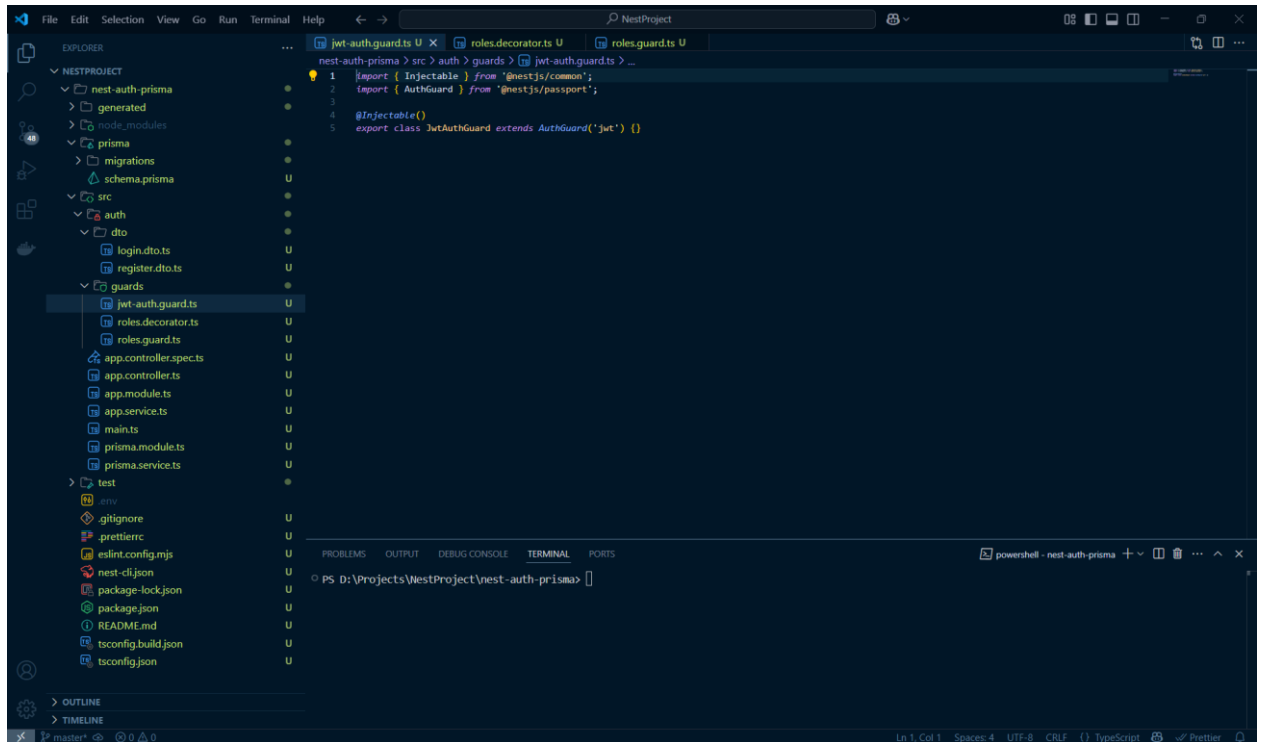


Рис 7. Jwt-auth.guard.ts

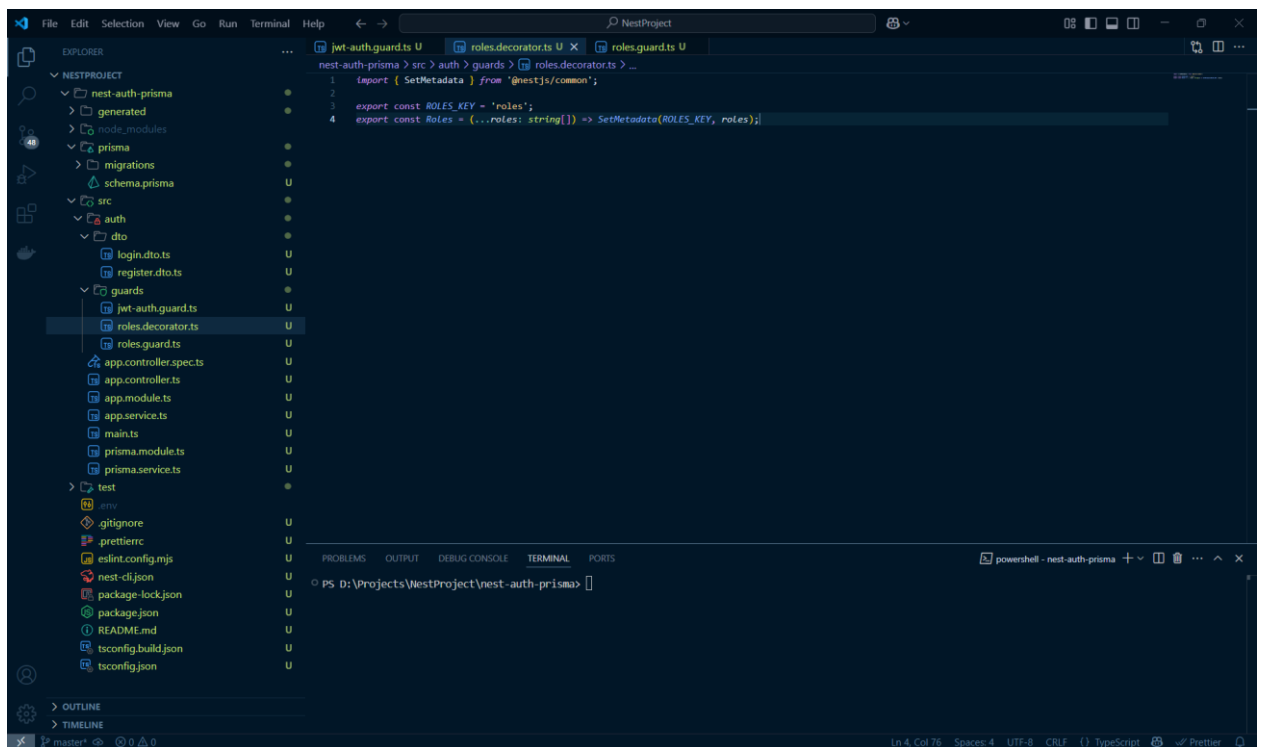


Рис 8. Roles.decorator.ts

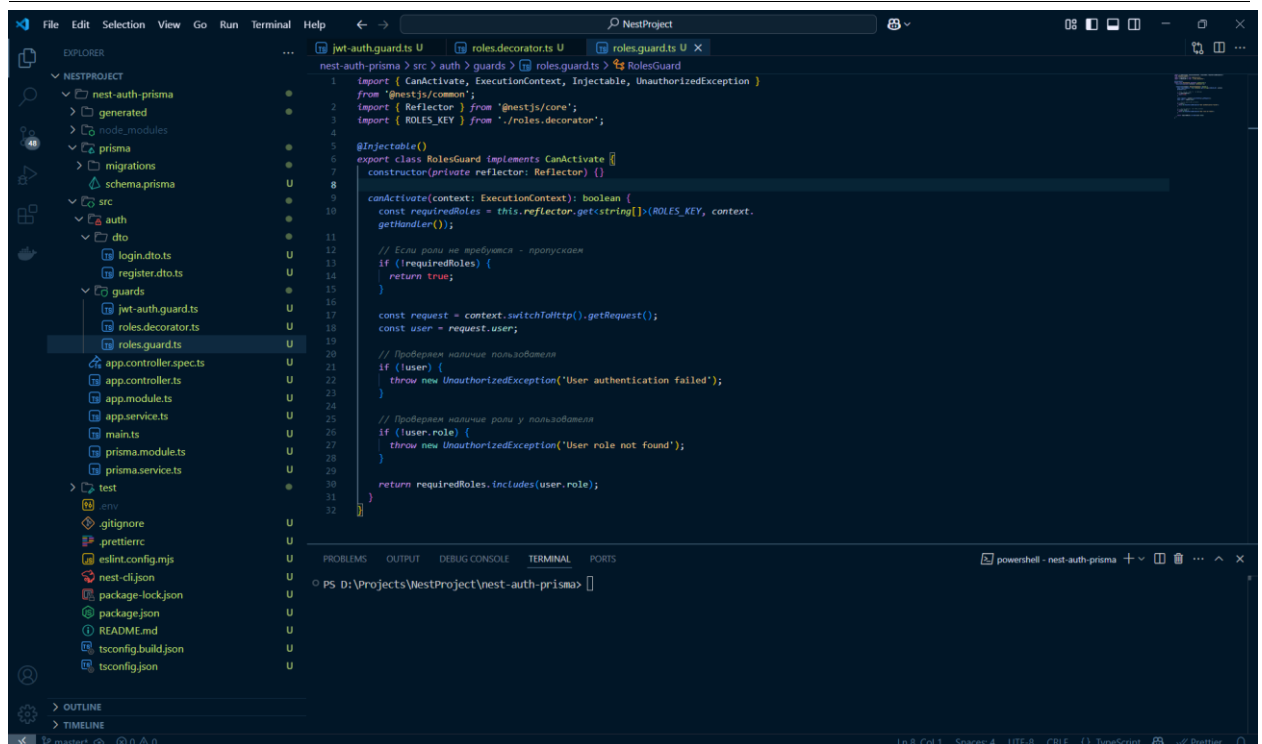


Рис 9. Roles.guard.ts

Создаём папку auth/strategies, где будет храниться ключевой компонент системы аутентификации - jwt.strategy.ts (рис. 10). Данный файл будет получать из запроса токен, по которому получит данные о пользователе и его роли. Для проверки токена необходимо добавить в файл .env переменную JWT_SECRET и присвоить ей любое значение, достаточно сложное для угадывания.

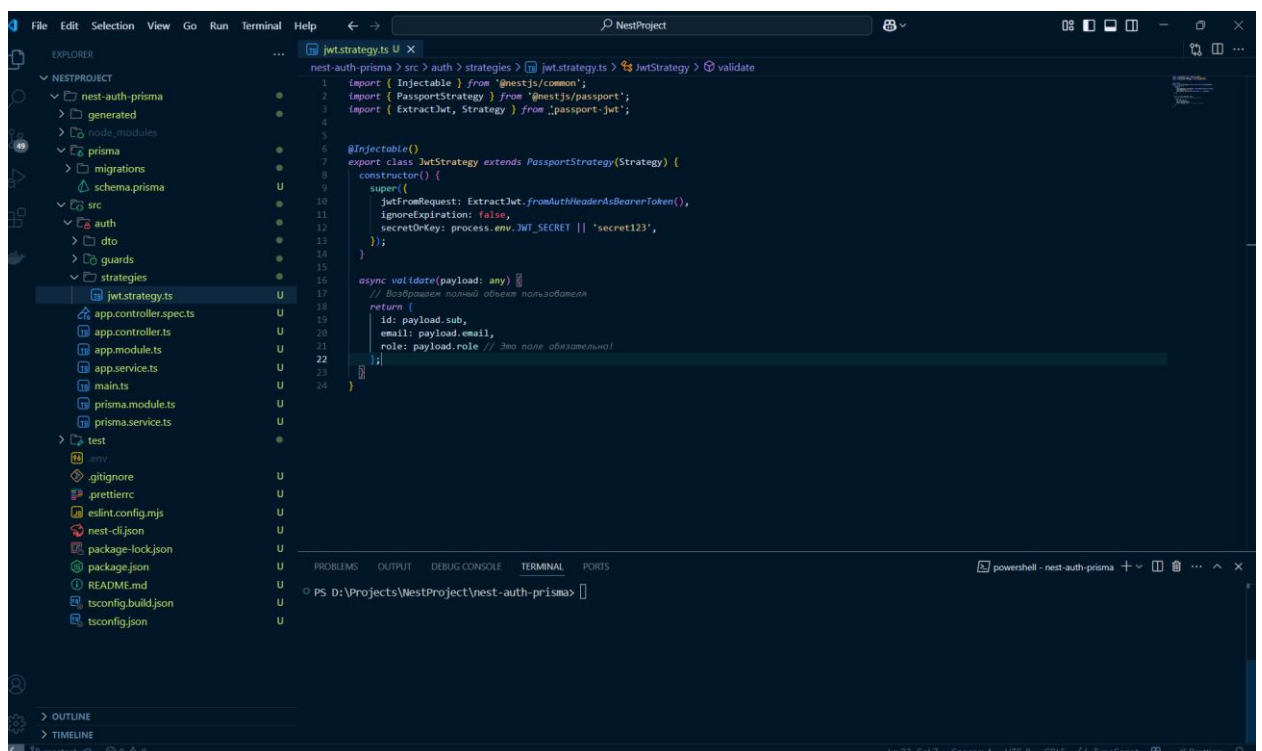


Рис 10. Jwt.strategy.ts

Теперь создадим три основных файла:

1. `auth.controller.ts` (рис. 11), проверяющий валидность входных данных и возвращающий пользователю JWT-токен или его данные;
2. `auth.service.ts` (рис 12), который будет хешировать пароли при помощи `bcrypt`, проверять авторизованность пользователя, генерировать JWT-токен и работать с базой данных;
3. `auth.module.ts` (рис 13), импортирующий необходимые зависимости и связующий сервисы с контроллером.

Чтобы сгенерировать все три файла сразу, достаточно ввести в терминал «`nest generate resource auth --no-spec`». Ключ `--no-spec` позволяет отменить создание файлов `.spec.ts`, поскольку в данном проекте они будут лишними. После введения команды, терминал спросит «*What transport layer do you use?*», на что отвечаем «*REST API*». На вопрос «*Would you like to generate CRUD entry points?*» отвечаем «*n*», что означает «*no*». Также для работы с JWT-токенами в сервисе и модуле необходимо импортировать пакет `@nestjs/jwt` через «`npm install @nestjs/jwt`».

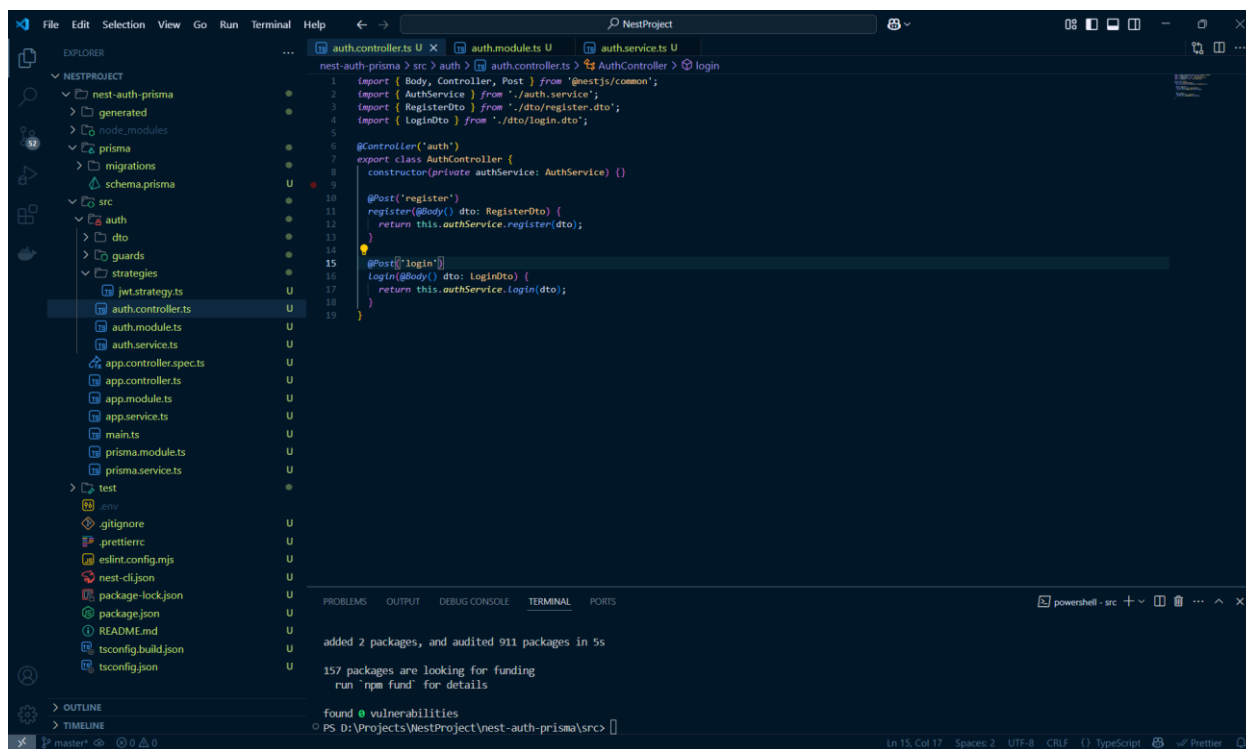


Рис 11. `Auth.controller.ts`

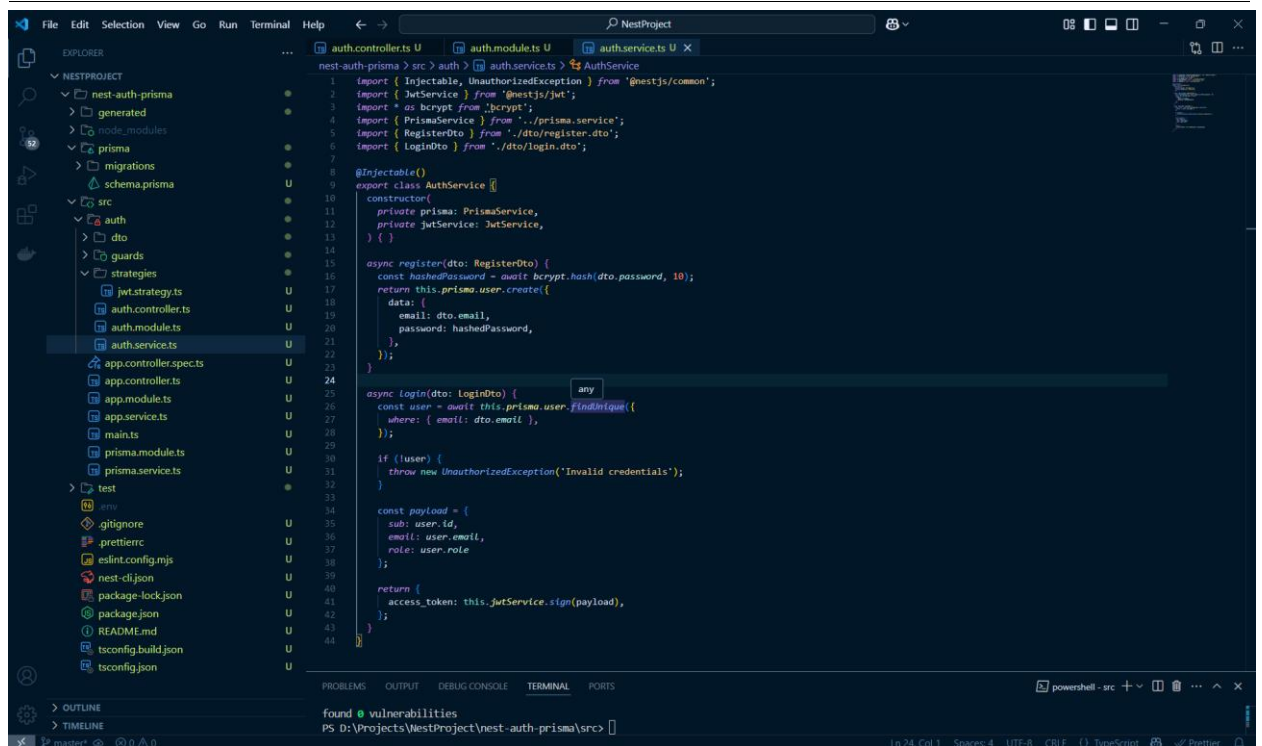


Рис 12. Auth.service.ts

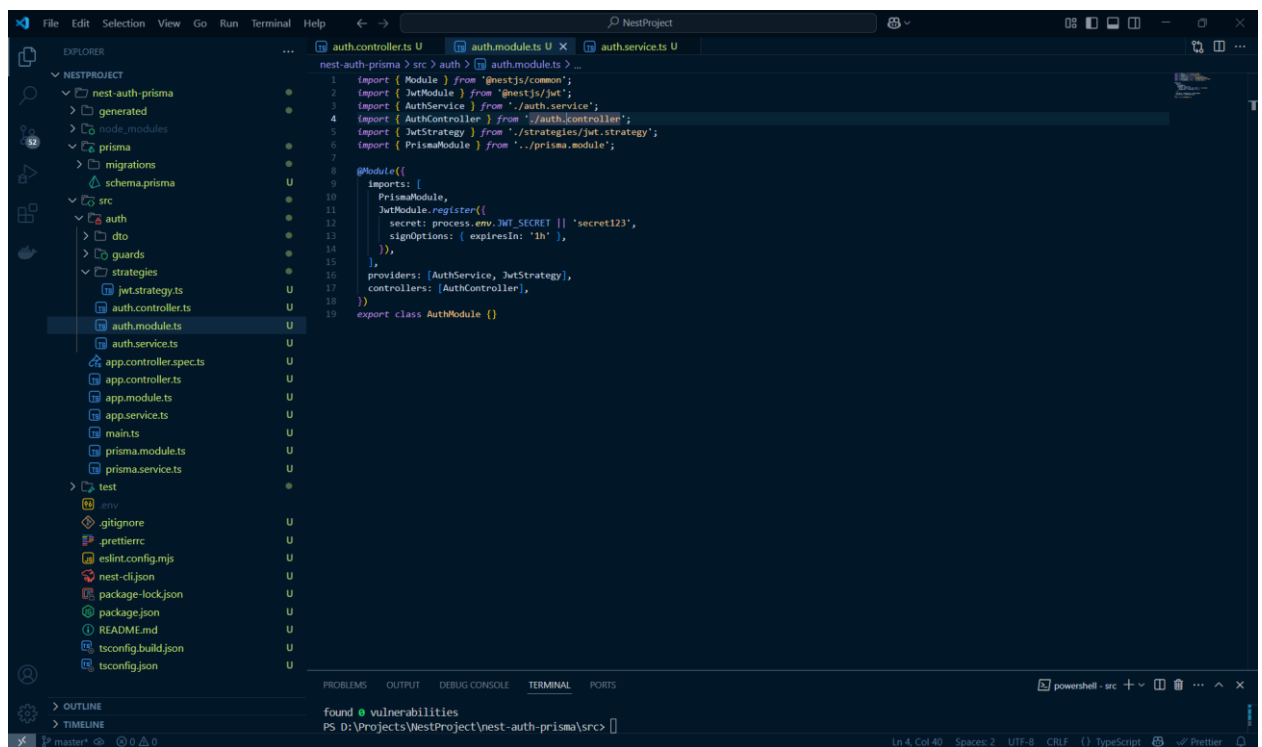


Рис 13. Auth.module.ts

В папке src нужно создать папку profile. В ней будут храниться profile.controller.ts (рис. 14) и profile.module.ts (рис. 15), которые в отличие от auth.controller.ts и auth.module.ts будут работать с только аутентифицированными пользователями.

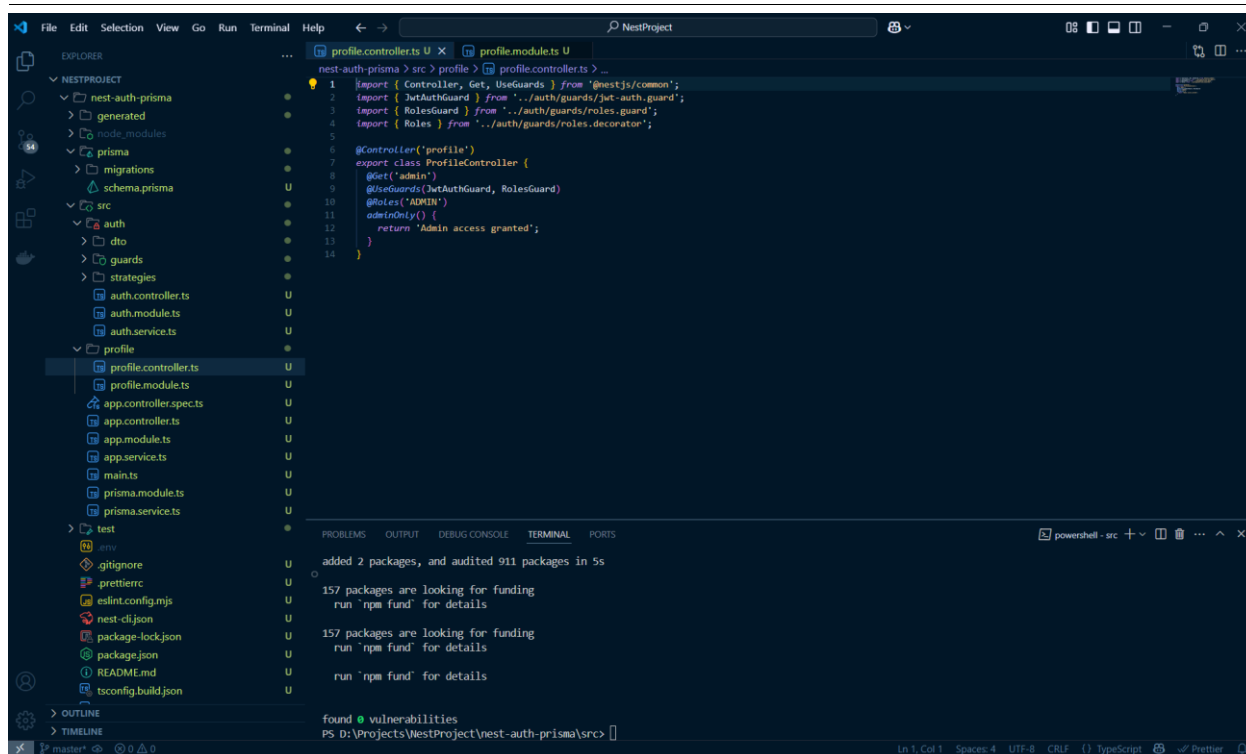


Рис 14. Profile.controller.ts

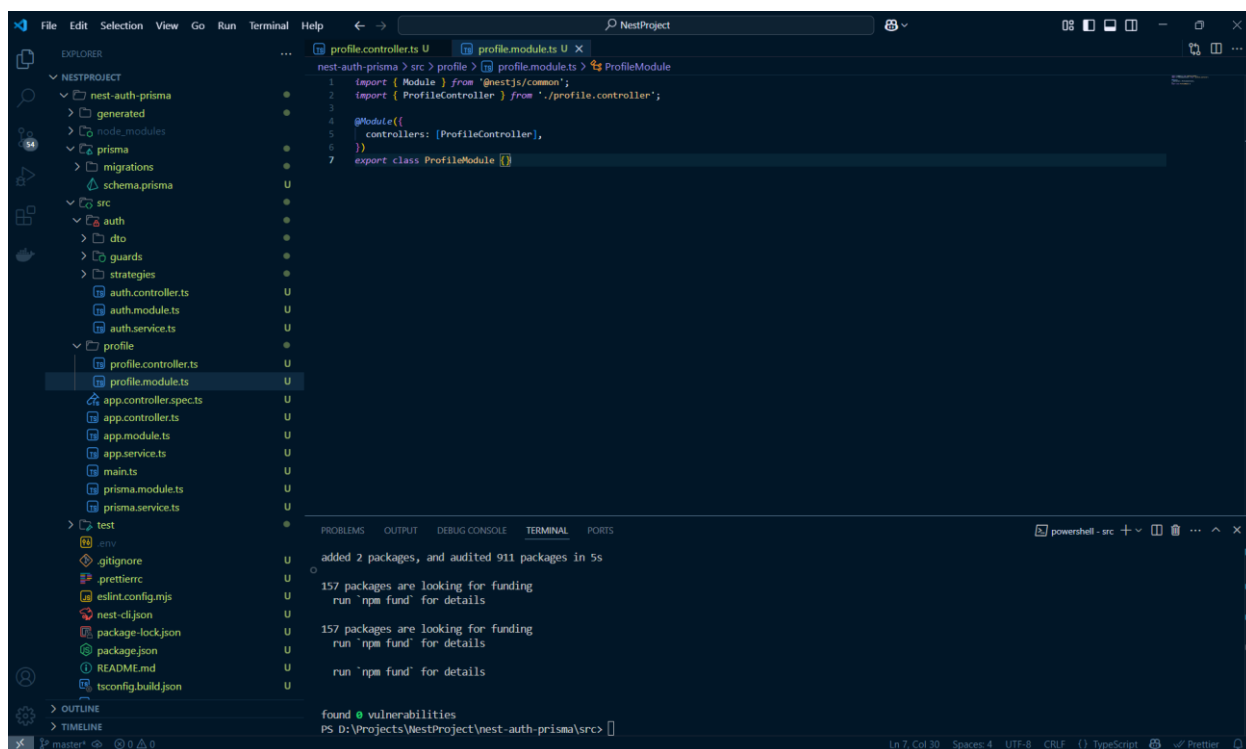


Рис 15. Profile.module.ts

В терминале перемещаемся в папку nest-auth-prisma и запускаем сервер через «*npm run start:dev*». При запуске получаем ошибку об отсутствии клиента (рис. 16). Связано это с тем, что сервер не видит клиента в пути, указанном в schema.prisma.

```
[Nest] 9956 - 03.05.2025, 20:01:10 LOG [NestFactory] Starting Nest application...
[Nest] 9956 - 03.05.2025, 20:01:10 ERROR [ExceptionHandler] Error: @prisma/client did not initialize yet. Please run "prisma generate" and try to import it again.
    at new PrismaClient (D:\Projects\WestProject\nest-auth-prisma\node_modules\prisma\client\default.js:43:11)
    at new PrismaService (D:\Projects\WestProject\nest-auth-prisma\src\prisma.service.ts:5:8)
    at Injector.instantiateClass (D:\Projects\WestProject\nest-auth-prisma\node_modules\@nestjs\core\injector\injector.js:373:19)
    at callback (D:\Projects\WestProject\nest-auth-prisma\node_modules\@nestjs\core\injector\injector.js:65:45)
    at async Injector.resolveConstructorParams (D:\Projects\WestProject\nest-auth-prisma\node_modules\@nestjs\core\injector\injector.js:145:24)
    at async Injector.loadInstance (D:\Projects\WestProject\nest-auth-prisma\node_modules\@nestjs\core\injector\injector.js:70:13)
    at async Injector.loadProvider (D:\Projects\WestProject\nest-auth-prisma\node_modules\@nestjs\core\injector\injector.js:98:9)
    at async Injector.lookupComponentInImports (D:\Projects\WestProject\nest-auth-prisma\node_modules\@nestjs\core\injector\injector.js:297:17)
    at async Injector.lookupComponentInParentModules (D:\Projects\WestProject\nest-auth-prisma\node_modules\@nestjs\core\injector\injector.js:260:33)
    at async Injector.resolveComponentInstance (D:\Projects\WestProject\nest-auth-prisma\node_modules\@nestjs\core\injector\injector.js:215:33)
```

Рис 16. Ошибка сервера

Меняем адрес с «../generated/prisma» на «../node_modules/.prisma/client», регенерируем клиента и перезапустим сервер. Теперь сервер работает исправно (рис. 17).

```
[Nest] 12432 - 03.05.2025, 20:30:20 LOG [NestFactory] Starting Nest application...
[Nest] 12432 - 03.05.2025, 20:30:20 LOG [InstanceLoader] PrismaModule dependencies initialized +26ms
[Nest] 12432 - 03.05.2025, 20:30:20 LOG [InstanceLoader] JwtModule dependencies initialized +2ms
[Nest] 12432 - 03.05.2025, 20:30:20 LOG [InstanceLoader] AppModule dependencies initialized +1ms
[Nest] 12432 - 03.05.2025, 20:30:20 LOG [InstanceLoader] AuthModule dependencies initialized +0ms
[Nest] 12432 - 03.05.2025, 20:30:20 LOG [RoutesResolver] AppController {}: +5ms
[Nest] 12432 - 03.05.2025, 20:30:20 LOG [RouterExplorer] Mapped {/, GET} route +4ms
[Nest] 12432 - 03.05.2025, 20:30:20 LOG [RoutesResolver] AuthController {/auth}: +0ms
[Nest] 12432 - 03.05.2025, 20:30:20 LOG [RouterExplorer] Mapped {/auth/register, POST} route +1ms
[Nest] 12432 - 03.05.2025, 20:30:20 LOG [RouterExplorer] Mapped {/auth/login, POST} route +1ms
[Nest] 12432 - 03.05.2025, 20:30:20 LOG [NestApplication] Nest application successfully started +17ms
```

Рис 17. Рабочий сервер

Чтобы проверить, что система авторизации и аутентификации работает, воспользуемся Postman. Postman — это универсальная платформа для работы с API, которая сочетает в себе мощные инструменты для разработки, тестирования и документирования интерфейсов. Через него тестировщики отправляют различные HTTP-запросы к приложениям, получают и проверяют ответы. Это позволяет проверять, правильно ли работает приложение. Проверять будем три запроса:

1. POST-запрос на регистрацию пользователя (рис. 18)

Выбираем тип запроса POST, вставляем ссылку «http://localhost:3000/auth/register», в разделе Body выбираем raw, тип текста ставим JSON и в этом формате вставляем придуманные email и пароль. После отправки запроса получаем ответ в формате JSON. Чтобы проверить, что пользователь сохранился в базу данных, создадим в vscode отдельный терминал и напишем «*npx prisma studio*» для просмотра в браузере (рис. 19).

2. POST-запрос на вход в профиль и получение JWT-токена (рис. 20)

С предыдущего запроса необходимо изменить только ссылку на «http://localhost:3000/auth/login». В результате формируется JWT-токен

3. GET-запрос на проверку прав пользователя по токenu (рис. 21)

Меняем тип запроса на GET, вставляем ссылку «http://localhost:3000/profile/admin», в разделе Headers в качестве ключа вставляем «Authorization», а в качестве значения «Bearer <сгенерированный JWT-токен>». Чтобы запрос сработал, нужно в коде app.module.ts (рис. 22) импортировать ProfileModule, иначе маршрут profile/admin не будет найден сервером. В результате получаем ошибку 403. Это означает, что в доступе

отказано, поскольку это обычный пользователь. Если в Prisma Studio поменять роль с «USER» на «ADMIN» и выполнить запрос на вход, сгенерируется новый токен, который в этом запросе даст доступ (рис. 23).

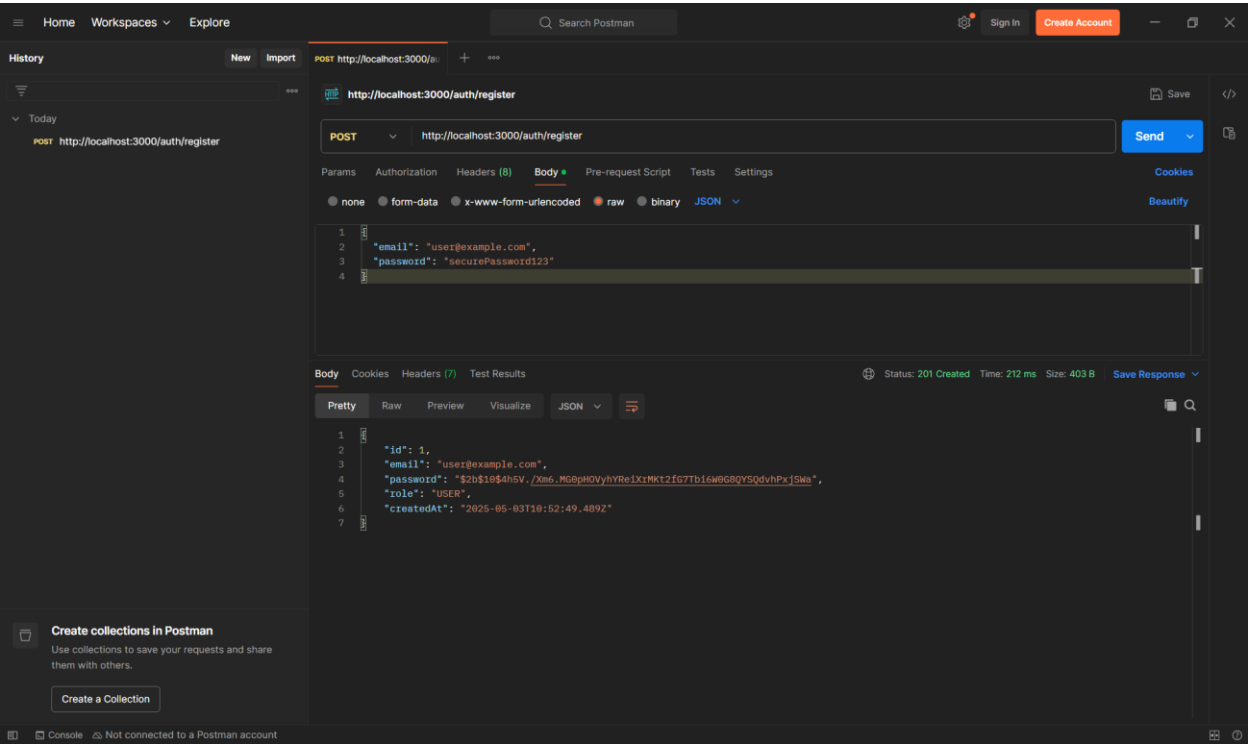


Рис 18. Запрос на регистрацию

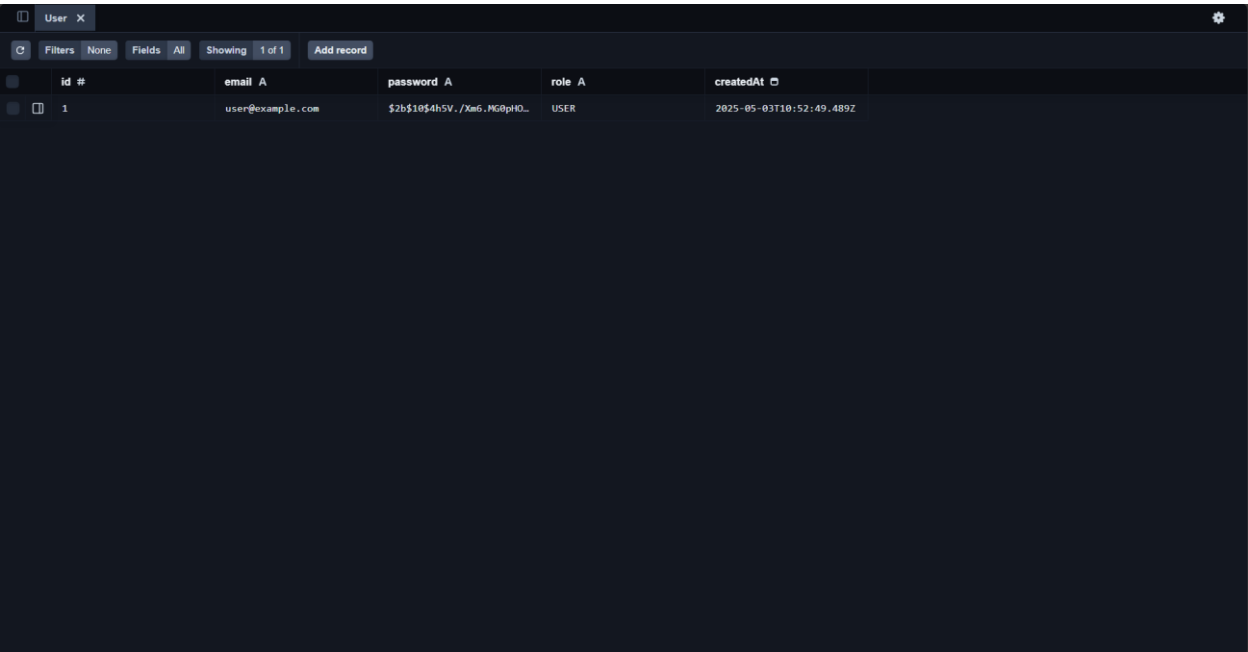


Рис 19. База данных в Prisma Studio

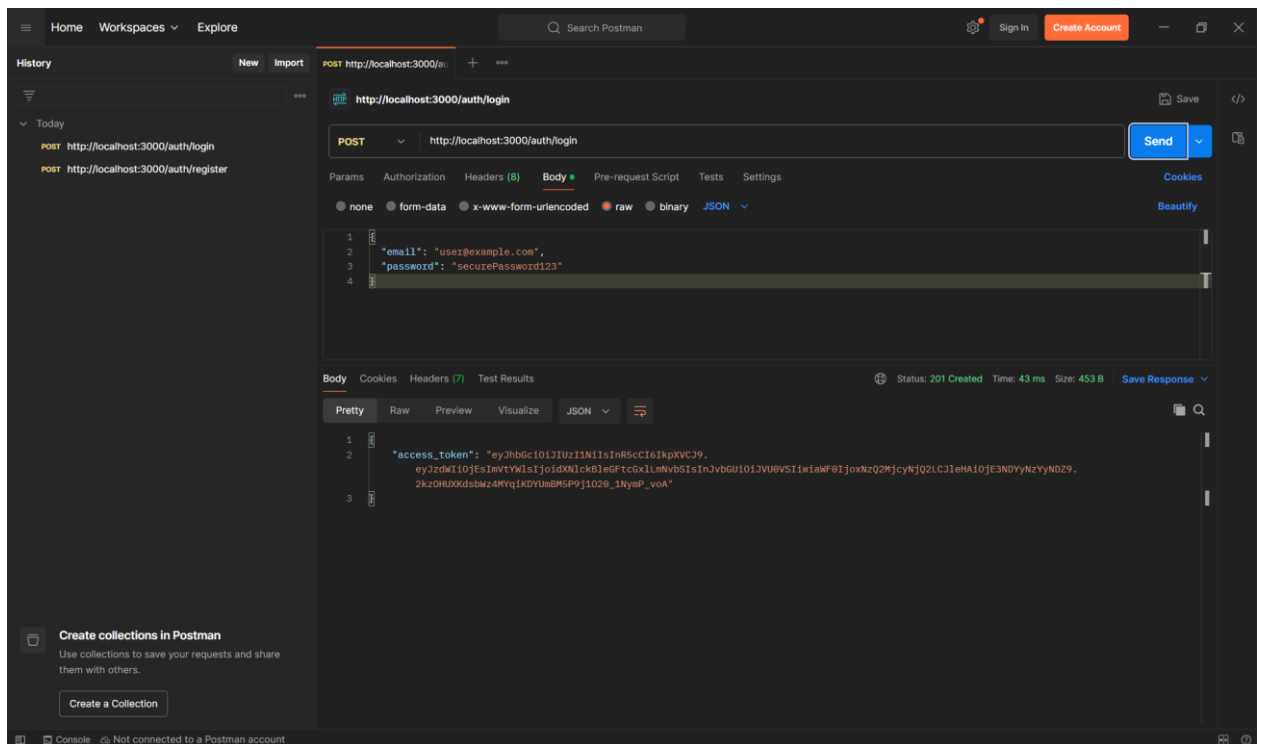


Рис 20. Запрос на вход

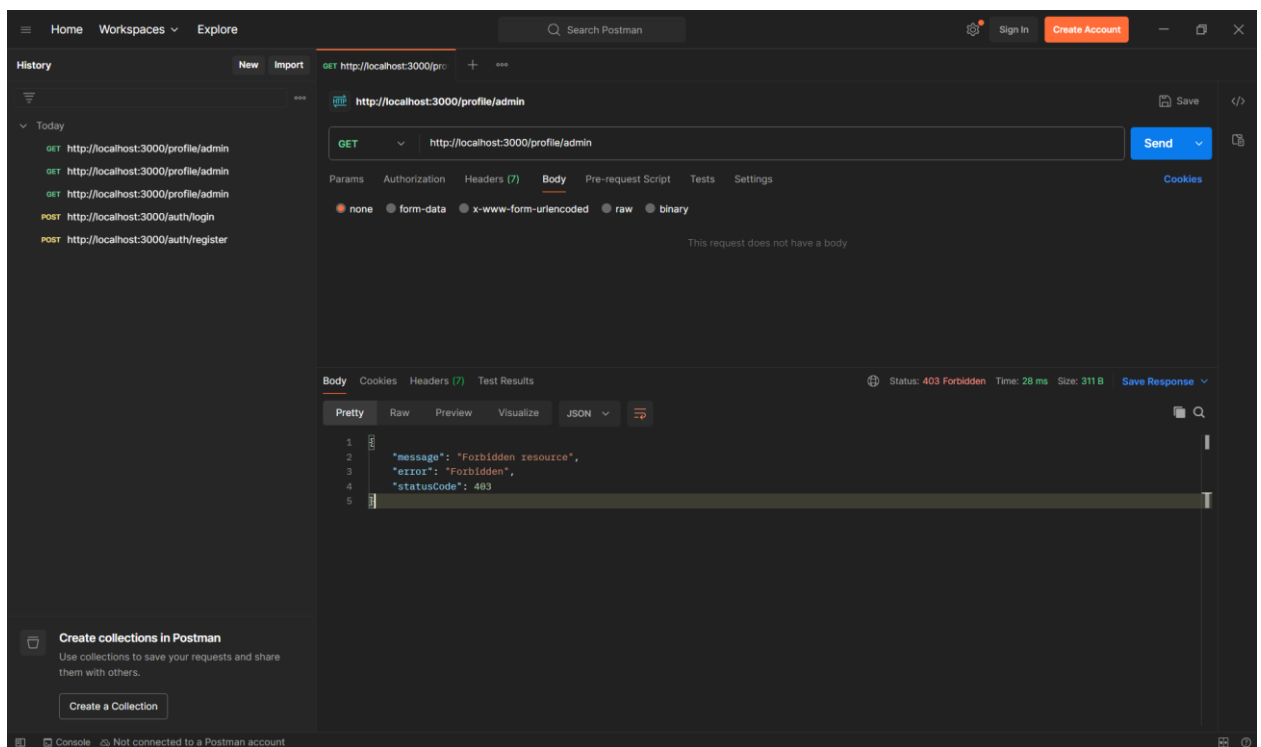


Рис 21. Запрос на проверку прав обычного пользователя

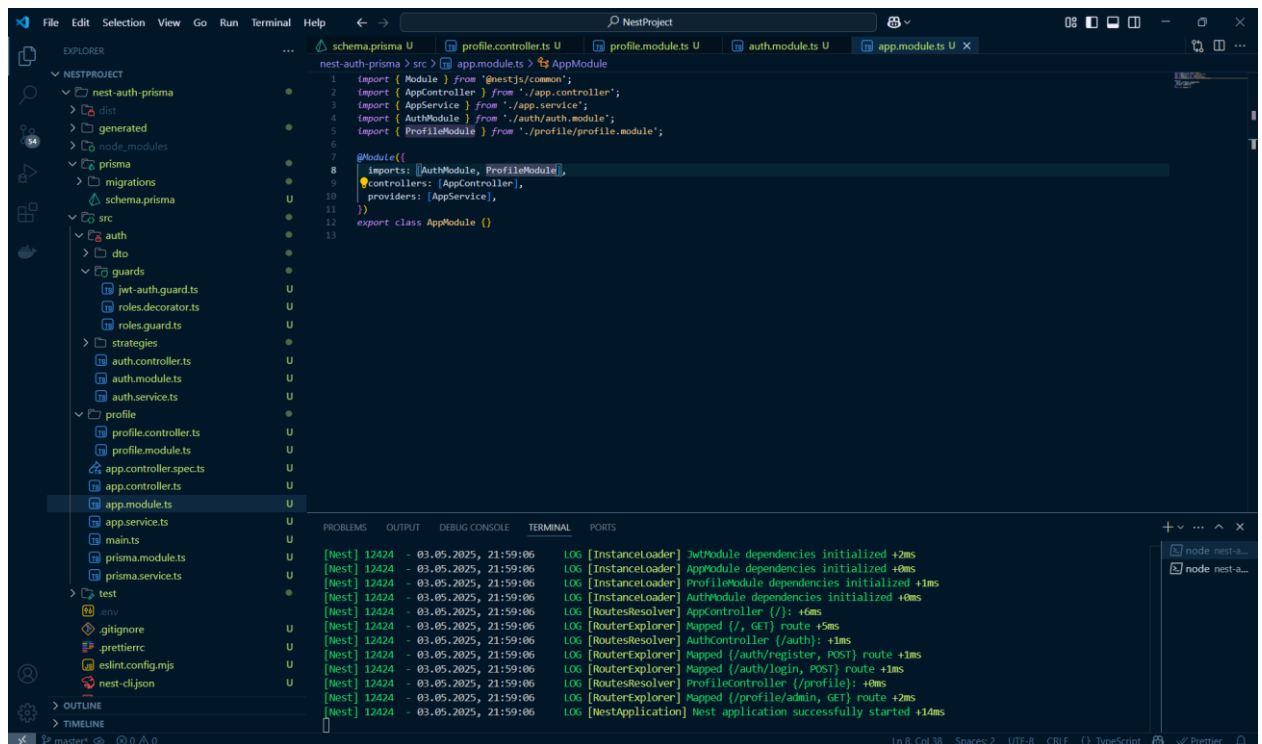


Рис 22. Импорт ProfileModule в app.module.ts

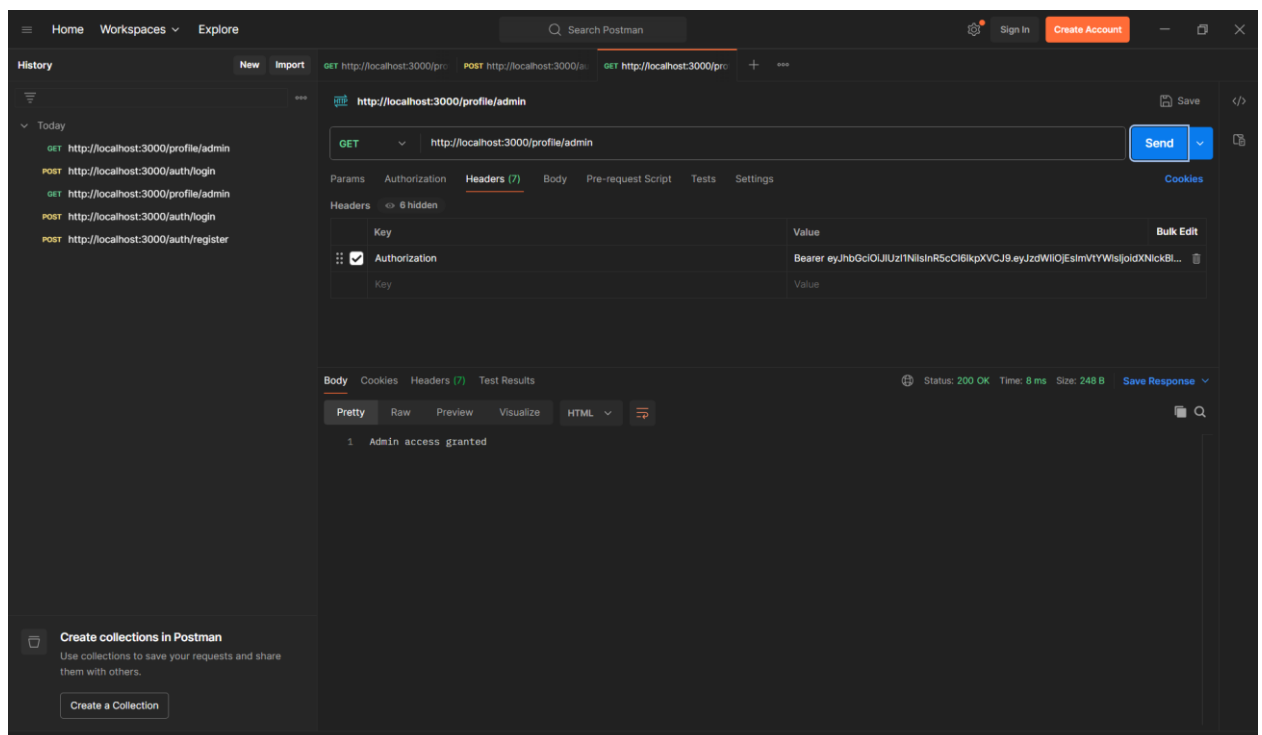


Рис 23. Запрос на проверку прав админа

4 Выводы

В ходе исследования была разработана надежная система аутентификации и авторизации в NestJS с использованием JWT-токенов, Prisma ORM и MySQL. Реализована ролевая модель (RBAC), позволяющая разграничивать доступ между обычными пользователями (USER) и администраторами (ADMIN). Это обеспечивает гибкость в управлении

правами доступа к API. Использование Prisma значительно упростило работу с базой данных, обеспечив типобезопасность и удобный API для выполнения запросов. Миграции позволили эффективно управлять изменениями схемы данных. Тестирование API через Postman подтвердило корректность работы системы.

Библиографический список

1. Мытников А. Н., Анохин В. Р. Механизмы обеспечения информационной безопасности в фреймворке Nestjs //Современные вопросы естествознания и экономики. 2023. С. 574-579.
2. Долженко В. А. Взаимодействие микросервисов в сложных финансовых системах с использованием NestJS //Актуальные исследования. 2023. №. 10 (140).
3. Лакшинских А. В., Давыденко Д. К. Разработка web-приложения для исторических материалов с использованием react и nestjs //Современные достижения молодых исследователей : сборник статей II Международного научно-исследовательского конкурса. Пенза: МЦНС «Наука и Просвещение», 2023. С. 19.
4. Девицына С. Н., Пилькевич П. В., Удод Е. В. Способы улучшения защищённости сервисов, использующих JWT-токены //Экономика. Информатика. 2023. Т. 50. №. 1. С. 144-151.