МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ «ВИТЕБСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ П.М. МАШЕРОВА»

Факультет математики и информационных технологий

Кафедра прикладного и системного программирования

КУРСОВОЙ ПРОЕКТ

по дисциплине «Системное программирование»

РАЗРАБОТКА \_\_\_

Панин Дмитрий Сергеевич,

3-й курс, 21ПОИТ1д

Руководитель:

Ермоченко Сергей Александрович,

Доцент кафедры прикладного и системного программирования, кандидат физико-математических наук

Витебск, 2023

Реферат

Курсовая работа 31 с., 8 рис., 20 источников, 2 прил., 4 листинга

CLIENT-SERVER, FRONT END, BACK END, REACT, REDUX, HTTP, NODE.JS, EXPRESS, NESTJS, TYPESCRIPT, БАЗЫ ДАННЫХ.

Объект исследования – архитектура и устройство серверных приложений, система их взаимодействия.

Предмет исследования – возможность создания функционирующего веб-приложения онлайн-чата.

Цель работы – изучить и продемонстрировать возможности современных технологий создания веб-приложений на основе средств разработки NestJS, апробировать их в разработке веб-приложения.

Методы исследования: теоретическое изучение и анализ информации о средствах и способах разработки серверных веб-приложений и их архитектуры, сравнительно-сопоставительный анализ инструментов разработки, изучение и анализ технической литературы, применение полученной информации для создания приложения.

Элементы новизны: возможность использования современных и востребованных технологий в веб-разработке.

Теоретическая и практическая ценность: работа помогает выявить и оценить актуальные средства разработки серверных веб-приложений, освоить практические навыки их использования.

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 4](#_Toc155320543)

[1 Теоретическая часть 5](#_Toc155320544)

[1.1 Актуальные подходы проектирования и реализации крупных приложений 5](#_Toc155320545)

[1.2 Обзор NestJS 8](#_Toc155320546)

[1.3 NestJS как расширение Express, сравнение фреймворков 14](#_Toc155320547)

[2 Практическая часть 17](#_Toc155320548)

[2.1 Проектирование, используемые библиотеки 17](#_Toc155320549)

[2.2 Реализация серверного приложения 20](#_Toc155320550)

[2.3 Расширение инструментария приложения 28](#_Toc155320551)

[Заключение 30](#_Toc155320552)

[Список использованных источников 32](#_Toc155320553)

# ВВЕДЕНИЕ

В наше время трудно представить крупный сервис, который не использовал бы в своей реализации разделение зон ответственности и масштабируемую архитектуру.

В наше время существует огромное количество серверных технологий, которые выполняют одни и те же функции, имеют свою специфику. В практике реализации сервисов появились общие понятия и паттерны, общие для многих библиотек и фреймворков языков программирования, реализующих межсерверное и клиент-серверное взаимодействие. Из мира технологий серверного JavaScript вот уже несколько лет набирает обороты фреймворк NestJS, который позволяет погрузиться в мир крупных систем и приложений, совмещая в себе возможность использования современных и актуальных подходов конструирования архитектуры систем, спроектированных для решения проблем как конечного пользователя, так и бизнеса.

Актуальность работы обусловлена необходимостью понимания разработчиком взаимодействия удалённых устройств внутри информационный сети и владения знаниями о современных средствах разработки для реализации программного обеспечения, обслуживающего разные виды взаимодействия.

Цель работы - изучить и продемонстрировать возможности современных технологий создания серверных приложений на основе фреймворка NestJS.

Задачи работы:

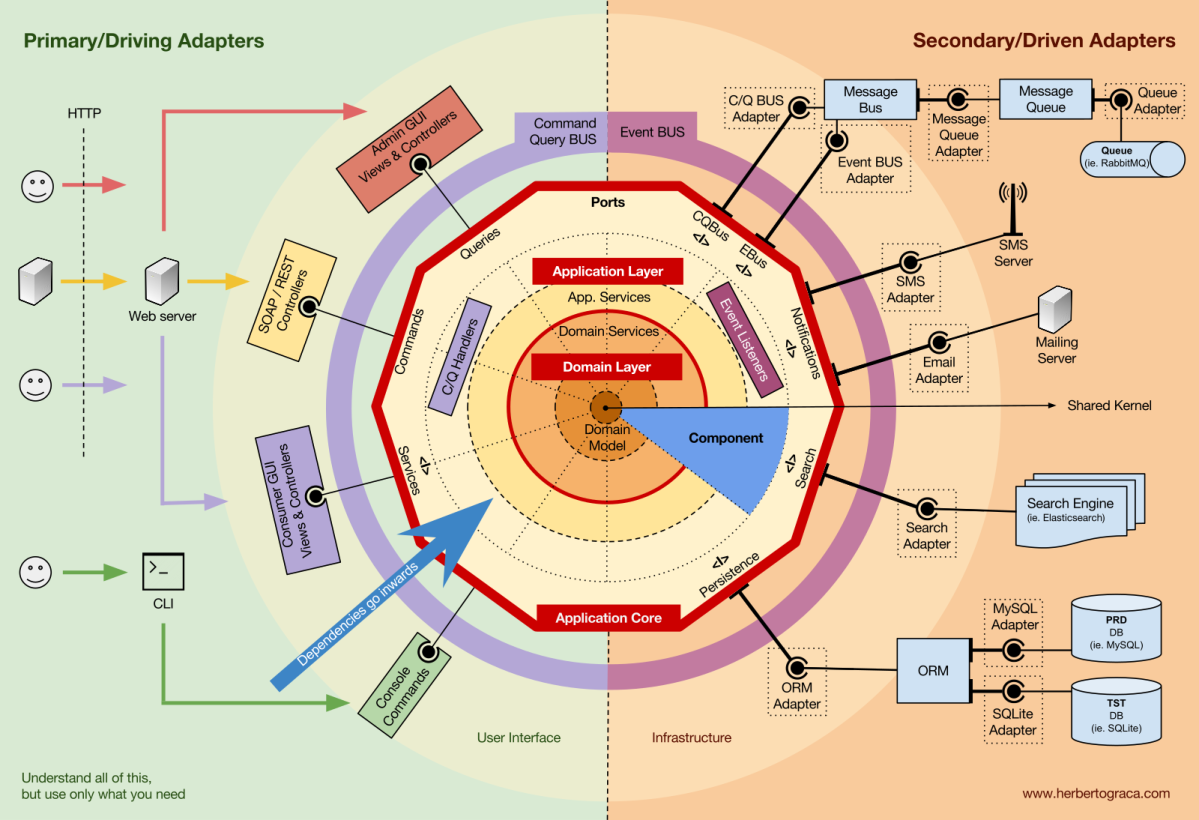
1. Выявить наиболее актуальные подходы для проектирования архитектуры крупных приложений;
2. Рассмотреть специфику и отличия использования фреймворков Express и NestJS, проанализировать простоту расширения инфраструктуры.
3. Адаптировать и переработать структуру исходного приложения в рамках фреймворка NestJS.

# 1 Теоретическая часть

## 1.1 Актуальные подходы проектирования и реализации крупных приложений

Крупные приложения характеризуются задействованием большого количества ролей в доведении конечного продукта до использования, требованием к построению стабильного решения и возможностью быстро адаптироваться под переменчивые требования бизнес-логики в совокупности с длительным использованием продукта и масштабируемости приложения на разные предметные области. Эти требования приводят к целям снижения сложности и стоимости поддержки за счёт стоимости первоначальной разработки, производительности потребляемых ресурсов.

Помимо внедрения гибких методологий разработки программного обеспечения (Agile, SCRUM), конкретная архитектура и реализация перенимает на себя ответственность за соблюдения вышеперечисленных требований. Унификация процесса разработки породила много подходов для улучшения технологического процесса разработки программного обеспечения. Появилось понятие Enterprise разработки. Enterprise разработка - это разработка, направленная на решение проблем бизнеса. В отличии от разработки для решения проблем конечных пользователей. А в подходы к организации архитектуры информационных систем крепко внедрились понятия объектно-ориентированного программирования и проектирования, предметно-ориентированного проектирования (англ. DDD, Domain-Driven Design), разработки через тестирование (англ. TDD, Test-Driven Design), архитектурный шаблон источника событий (англ. Event sourcing), парадигмы разделения назначения запросов (англ. CQRS, The Command and Query Responsibility Segregation), микросервисная архитектура и т. д. (рисунок 1.1)

Рисунок 1.1 – Пример схемы приложения с внедрением DDD, CQRS 

Предметно-ориентированное проектирование — набор принципов и схем, направленных на создание оптимальных систем объектов. Сводится к созданию программных абстракций, которые называются моделями предметных областей. В эти модели входит бизнес-логика, устанавливающая связь между реальными условиями области применения продукта и кодом.

DDD — это подход, который нацелен на изучение предметной области предприятия в целом или каких-то отдельных бизнес-процессов. Это отличный подход для проектов, в которых сложность (запутанность) бизнес-логики достаточно велика. Его применение призвано снизить эту сложность, насколько возможно.

Основной принцип DDD — разделение приложения на домены.

Домен — предметная область, которая описывает совокупность проблем и целей бизнеса. Можно сказать, что это — отрасль деятельности, которую необходимо автоматизировать с помощью IT-решения. Например, если компания занимается грузовыми перевозками, в домен будут входить все связанные с этим проблемы: выбрать машину, построить маршрут и другие.

Ограниченный Контекст (Bounded Context) — ключевой инструмент DDD, это явная граница, внутри которой существует модель предметной области. Она отображает единый язык в модель программного обеспечения. Именно на основании контекстов можно разделить код на модули/пакеты/компоненты таким образом, чтобы изменения в каждом из них оказывали минимальное влияние на других.

Для разработчиков такой подход позволяет вносить изменения в код не опасаясь, что где-то в другом модуле что-то начнёт работать некорректно. Также такой подход помогает распараллеливать задачи, что сокращает время общих усилий.

Event sourcing (источники событий, регистрация событий, генерация событий) — это мощный архитектурный шаблон, при котором все изменения, вносимые в состояние приложения, сохраняются в той последовательности, в которой они происходили. Эти записи служат как источником для получения текущего состояния, так и журналом аудита того, что происходило в приложении за время его существования. Event sourcing способствует децентрализованному изменению и чтению данных. Такая архитектура хорошо масштабируется и подходит для систем, которые уже работают с обработкой событий или хотят перейти на такую архитектуру.

Event sourcing, используя события в качестве основной архитектурной концепции, также является парадигмой моделирования предметной области, лучше отражающей представление о системе.

CQRS — это стиль архитектуры, в котором операции чтения отделены от операций записи. Одна из естественных причин развития CQRS — не симметричное распределение нагрузки и сложности бизнес-логики на читающие и записывающие подсистемы Большинство бизнес-правил и сложных проверок находится во записывающей подсистеме. При этом читают данные зачастую в разы чаще, чем изменяют.

Также из-за стандартизации подхода к проектированию программного обеспечения, возрос спрос на инструменты, позволяющие легко внедрять эти подходы. Рассмотрим поддержку применения всего обилия подходов в фреймворке NestJS.

## 1.2 Обзор NestJS

Nest (NestJS) — это платформа для создания эффективных и масштабируемых серверных приложений Node.js. Она использует прогрессивный JavaScript, создана на основе TypeScript и полностью его поддерживает (но при этом позволяет разработчикам писать код на чистом JavaScript) и сочетает в себе элементы ООП (объектно-ориентированное программирование), FP (функциональное программирование) и FRP (функциональное реактивное программирование).

В своей реализации Nest использует надежные платформы HTTP-сервера, такие как Express (по умолчанию), и при желании его также можно настроить на использование Fastify. Nest обеспечивает уровень абстракции по сравнению с обычными платформами Node.js (Express/Fastify), но также предоставляет их API непосредственно разработчику. Это дает разработчикам свободу использовать множество сторонних модулей, доступных для базовой платформы.

В последние годы, благодаря Node.js, JavaScript стал «лингва-франка» в сети как для клиентских, так и для серверных приложений. Это привело к появлению таких проектов, как Angular, React и Vue, которые повышают производительность разработчиков и позволяют создавать быстрые, тестируемые и расширяемые интерфейсные приложения. Однако, хотя для Node.js (и серверного JavaScript) существует множество превосходных библиотек, помощников и инструментов, ни одна из них не решает эффективно главную проблему — архитектуру.

Nest предоставляет готовую архитектуру приложений, которая позволяет разработчикам и командам создавать легко тестируемые, масштабируемые, слабосвязанные и легко поддерживаемые приложения.

Провайдеры — это фундаментальная концепция в Nest. Многие базовые классы Nest можно рассматривать как поставщики — сервисы, репозитории, фабрики, помощники и т. д. Основная идея провайдера заключается в том, что его можно внедрить как зависимость; это означает, что объекты могут создавать различные отношения друг с другом, а функцию «связывания» этих объектов можно в значительной степени делегировать системе времени выполнения Nest.

Масштабируемость и удобство использования Nest во многом зависит от реализованной системы внедрения зависимостей.

Nest построен на основе строгого шаблона проектирования, широко известного как внедрение зависимостей (англ. DI, Dependency Injection). Внедрение зависимостей — это стиль настройки объекта, при котором поля объекта задаются внешней сущностью. Экземпляры классов настраиваются внешними объектами. DI — это альтернатива самонастройке объектов.

Внедрение зависимостей в Nest — это метод инверсии управления (IoC), при котором вы делегируете создание экземпляров зависимостей контейнеру IoC (в нашем случае системе времени выполнения NestJS), а не обязательно делаете это в своем собственном коде.

Когда контейнер Nest IoC создает экземпляр класса провайдера, он сначала ищет любые зависимости в нём. Когда он находит зависимость, он выполняет поиск токена этой зависимости, который возвращает класс провайдера на этапе регистрации. Предполагая область действия SINGLETON (поведение по умолчанию), Nest затем либо создаст новый экземпляр провайдера, кэширует его и вернет, либо, если он уже кэширован, вернет существующий экземпляр.

Это объяснение немного упрощено, чтобы проиллюстрировать суть. Одна важная область, которую мы упустили, заключается в том, что процесс анализа кода на наличие зависимостей очень сложен и происходит во время начальной загрузки приложения. Одной из ключевых особенностей является то, что анализ зависимостей (или «создание графа зависимостей») является транзитивным. Граф зависимостей гарантирует, что зависимости разрешаются в правильном порядке — по существу «снизу вверх». Этот механизм освобождает разработчика от необходимости управлять такими сложными графами зависимостей.

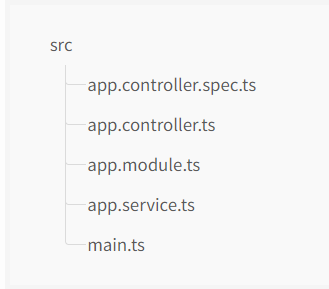
Самой базовой архитектурой приложения является паттерн контроллер – сервис – репозиторий, который уже предоставляет достаточно большую возможность разделения зон ответственности приложения.

Контроллер отвечает за обработку HTTP (gRPC, GraphQL, и т. д.) запроса. Целью контроллера является получение конкретных запросов для приложения. Механизм маршрутизации контролирует, какой контроллер какие запросы получает. Часто каждый контроллер имеет более одного маршрута, и разные маршруты могут выполнять разные действия.

Чтобы создать базовый контроллер, мы используем классы и декораторы. Декораторы связывают классы с необходимыми метаданными и позволяют Nest создавать карту маршрутизации (привязывать запросы к соответствующим контроллерам).

Cервис обеспечивает уровень бизнес-логики, а репозиторий отвечает за хранение данных, взаимодействия с хранилищами (рисунок 1.2).

Рисунок 1.2 – Стандартная файловая структура домена в NestJS



Модуль в Nest — это класс, помеченный декоратором @Module(). Декоратор @Module() предоставляет метаданные, которые Nest использует для организации структуры приложения.

Каждое приложение имеет как минимум один модуль — корневой модуль. Корневой модуль — это отправная точка, которую Nest использует для построения графа приложения — внутренняя структура данных, которую Nest использует для разрешения отношений и зависимостей модуля и поставщика. Хотя очень маленькие приложения теоретически могут иметь только корневой модуль, это не типичный случай. Модули настоятельно рекомендуются как эффективный способ организации ваших компонентов. Таким образом, для большинства приложений результирующая архитектура будет использовать несколько модулей, каждый из которых инкапсулирует тесно связанный набор возможностей.

По умолчанию модуль инкапсулирует провайдеры. Это означает, что невозможно внедрить провайдеры, которые не являются ни непосредственно частью текущего модуля, ни экспортированы из импортированных модулей. Таким образом, можно рассматривать экспортированные провайдеры из модуля как общедоступный интерфейс модуля или API.

Middleware — это функция, которая вызывается перед обработчиком маршрута. Функции промежуточного программного обеспечения имеют доступ к объектам запроса и ответа, а также к функции промежуточного программного обеспечения next() в цикле запрос-ответ приложения. Следующая middleware обычно обозначается переменной с именем next.

Middleware в Nest по умолчанию эквивалентно middleware из Nest.

Middleware могут выполнять следующие задачи:

* выполнить любой код.
* вносить изменения в объекты запроса и ответа.
* завершить цикл запрос-ответ.
* вызвать следующую функцию промежуточного программного обеспечения в стеке.
* если текущая middlware не завершает цикл запрос-ответ, она должна вызвать next(), чтобы передать управление следующей middleware.

Nest поставляется со встроенным слоем исключений, который отвечает за обработку всех необработанных исключений в приложении. Если исключение не обрабатывается кодом вашего приложения, оно перехватывается этим уровнем, который затем автоматически отправляет соответствующий удобный для пользователя ответ.

Pipe — это класс, помеченный декоратором @Injectable(), который реализует интерфейс PipeTransform.

Pipe имеет два типичных варианта использования:

* преобразование: преобразует входные данные в желаемую форму (например, из строки в целое число).
* валидация: оценить входные данные и, если они действительны, просто передать их без изменений; в противном случае выдайте исключение

Guard — это класс, помеченный декоратором @Injectable(), который реализует интерфейс CanActivate.

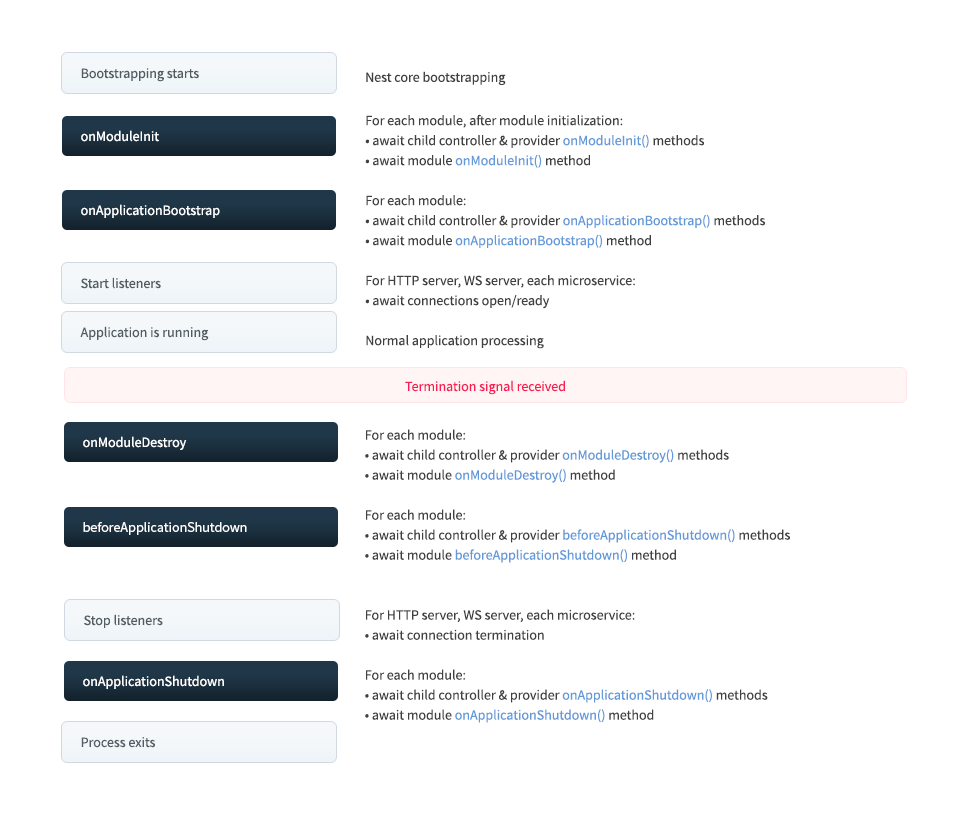
Guard несёт единственную ответственность. Он определяет, будет ли данный запрос обрабатываться обработчиком маршрута или нет, в зависимости от определенных условий (например, разрешений, ролей, списков управления доступом и т. д.), присутствующих во время выполнения. Это часто называют авторизацией. Авторизация (и ее родственник, аутентификация, с которой она обычно взаимодействует) обычно обрабатывается middleware в традиционных приложениях Express. Middlware — хороший выбор для аутентификации, поскольку такие вещи, как проверка токена и прикрепление свойств к объекту запроса, не сильно связаны с конкретным контекстом маршрута (и его метаданными).

Nest предоставляет несколько служебных классов, которые помогают упростить написание приложений, которые работают в нескольких контекстах приложений (например, на основе HTTP-сервера Nest, микросервисов и контекстов приложений WebSockets). Эти утилиты предоставляют информацию о текущем контексте выполнения, которую можно использовать для создания универсальных средств защиты, фильтров и перехватчиков, которые могут работать с широким набором контроллеров, методов и контекстов выполнения.

У приложения Nest, как и у каждого элемента приложения, есть жизненный цикл, которым управляет Nest. Nest предоставляет перехватчики жизненного цикла, которые обеспечивают видимость ключевых событий жизненного цикла и возможность действовать (запускать зарегистрированный код на ваших модулях, поставщиках или контроллерах), когда они происходят.

Ниже (рисунок 1.3) показана последовательность ключевых событий жизненного цикла приложения с момента загрузки приложения до завершения процесса узла. Мы можем разделить общий жизненный цикл на три фазы: инициализация, запуск и завершение. Используя этот жизненный цикл, вы можете планировать соответствующую инициализацию модулей и служб, управлять активными соединениями и корректно завершать работу приложения при получении сигнала завершения.

Рисунок 1.3 – Жизненный цикл Nest



Nest — это платформа-независимый фреймворк. Это означает, что вы можете разрабатывать логические части многократного использования, которые можно использовать в приложениях разных типов. Например, большинство компонентов можно повторно использовать без изменений в различных базовых платформах HTTP-серверов (например, Express и Fastify) и даже в разных типах приложений (например, в платформах HTTP-серверов, микросервисах с разными транспортными уровнями и веб-сокетах).

Кроме того, Nest поставляется со специальным модулем GraphQL. Вы можете использовать GraphQL в качестве уровня API, взаимозаменяемо с предоставлением REST API.

Кроме того, функция контекста приложения помогает создавать любые приложения Node.js, включая такие вещи, как CRON-джобы и приложения CLI, поверх Nest.

Nest стремится стать полноценной платформой для приложений Node.js, которая обеспечит более высокий уровень модульности и возможности повторного использования приложений и вне веб-разработки.

Таким образом Nest предоставляет готовые решения для распространённых паттернов устройства и организации архитектуры, решения для реализации клиент-серверного и межсерверного взаимодействия. Достаточно декларативно описывают своё назначения пакеты основных модулей: @nestjs/graphql, @nestjs/websockets, @nestjs/microservices, @nestjs/passport, @nestjs/cqrs и т. д. Официальная документация содержит большое количество рецептов и примеров внедрения паттернов и архитектурных решений.

## 1.3 NestJS как расширение Express, сравнение фреймворков

Прежде чем углубляться в подробности, важно понять, что сравнивается. И NestJS, и Express.js представляют собой платформы, специально предназначенные для создания серверных веб-приложений.

Фреймворк — это продуманный набор инструментов, который служит основой или отправной точкой для создания чего-либо. Формально: программная среда — это абстракция, обеспечивающая общую функциональность. Его можно выборочно изменять с помощью написанного пользователем кода, обеспечивая тем самым программное обеспечение, ориентированное на конкретное приложение.

Express — это то, что мы называем легкой и непредвзятой структурой. Это означает, что, только что выпущенный из коробки, он настолько прост, насколько это возможно, предоставляя очень мало функциональности. Чтобы начать с нуля, потребуется больше работы, поскольку многие базовые вещи придется создавать с нуля.

Во многих отношениях NestJS является противоположностью Express. Это чрезвычайно самоуверенная структура, которая придерживается парадигмы проектирования «соглашение важнее конфигурации». Другими словами, мы жертвуем гибкостью ради стандартизации и скорости.

Таблица 1.1 – Сравнительная таблица NestJS и Express

|  |  |  |
| --- | --- | --- |
| Критерий | NestJS | Express |
| Типизация | Статическая | Динамическая |
| Основные приложения, разрабатываемые на фреймворке | Веб-приложения, микросервисы, API, приложения реального времени. | Веб-приложения, RESTful API, одностраничные приложения, промежуточное ПО. |
| Производительность | Отличная производительность благодаря использованию TypeScript и компилируемому характеру. | Быстрый и легкий, подходит для работы с высокими нагрузками и быстрым откликом. |
| Стабильность | Стабильная версия с регулярными обновлениями и активной разработкой. | Стабильный и хорошо зарекомендовавший себя, активно поддерживается сообществом. |
| Поддержка сообщества | Активное и растущее сообщество с обширной документацией и официальной поддержкой. | Большое и хорошо зарекомендовавшее себя сообщество с множеством онлайн-ресурсов. |
| Время разработки | Время разработки может быть короче благодаря встроенным функциям. | Более медленный процесс разработки с простым синтаксисом и понятной архитектурой. |

Nest также подходит для крупных проектов с разветвлённой архитектурой и заделом на долгую поддержку.

С другой стороны, есть причина, почему Express так популярен. Трудно сравниться с такой гибкостью, и она идеально подходит для команд, которые хотят экспериментировать или опробовать нестандартизированные решения. Легкие фреймворки, такие как Express, способствуют порождению плохой архитектуры, но очень удобны в узких и специфических решениях из-за своей гипкости, жертвуя масштабируемостью.

С точки зрения производительности оба варианта одинаково эффективны, а что касается масштабируемости, хотя NestJS немного легче масштабировать, Express может предложить то же самое, но с затратой большого количество ресурсов и сил.

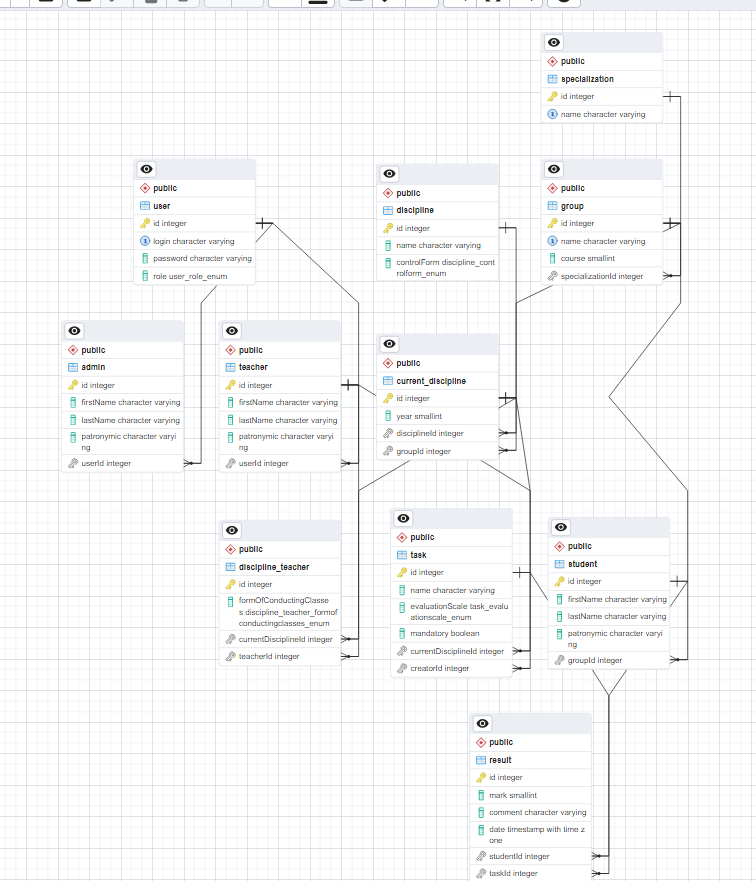
Nest больше подходит для enterprise сервисов, уже успешно используется большими корпорациями.

# 2 Практическая часть

## 2.1 Проектирование, используемые библиотеки

Одна из целей курсового проекта – мигрировать существующее Express приложение на Nest.

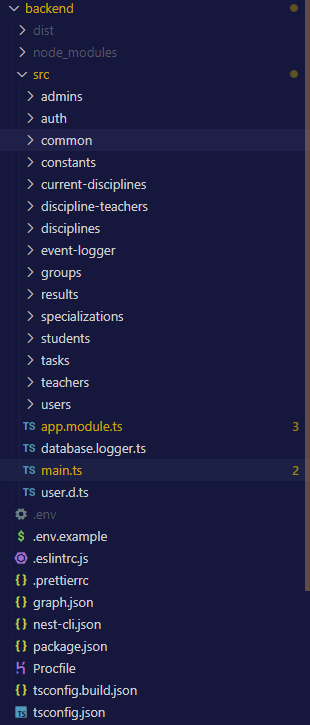
Рисунок 2.1 – ER-диаграмма базы данных



Для начала опишем модель базы данных, исходя из структуры предыдущего приложения (Рисунок 2.1). В исходном приложении использовалась ORM под названием TypeORM. Поддержка Nest взаимодействия с TypeORM позволяет переиспользовать уже реализованные модели данных.

Файловая структура Nest состоит из модулей. В нашем случае мы можем говорить, что модуль является представлением домена. Каждый модуль будет минимально содержать основной файл модуля, контроллер для обработки запросов и сервис для взаимодействия с репозиториями.

Рисунок 2.2 – Файловая структура серверного приложения



Каждый контроллер модуля будет реализовывать REST API, позволяющий производить CRUD операции над данными, некоторые модули в дополнение должны предоставлять возможность более продвинутой выборки данных. Авторизация будет происходить посредством Bearer-токена и HTTP заголовков запроса. Внутреннюю реализацию берёт на себя библиотека passport. Доступ к выборке и модификации данных будет контролироваться подходом управления доступом на основе ролей (англ. RBAC, управление доступом на основе ролей). Также в исходном приложении был реализован процесс размещения на удалённом сервере, настроены процессы CI/CD, для этого будем использовать модуль @nestjs/config, который позволяет манипулировать и взаимодействовать с переменными окружения.

Библиотека bcrypt позволит нам использовать криптографическую хеш-функцию для предварительного шифрования пароля перед внесением в базу данных, сравнивать результаты шифрования. Встроенная работа Nest CORS упрощают работу и конфигурацию HTTP CORS-запросов соответственно. Модуль @nestjs/jwt позволит использовать токены формата JSON Web Token в соответствии со стандартом RFC-7519, также он позволяет подписывать, расшифровывать и проверять токены, которые используются для авторизации и аутентификации в веб-приложениях.

## 2.2 Реализация серверного приложения

Точкой входа в приложение является файл main.ts, в корне директории src. Он определяет основную конфигурацию процесса Node.js, корневой модуль для инициализации экземпляра приложения, при необходимости подключения микросервисов и т. д. Корневой модуль в свою очередь является местом описания конфигурации переменных окружения, подключений к базам данных, импортирования остальных модулей приложения.

Рассмотрим реализацию одного из модулей. Чёткая архитектура позволяет сэкономить время при написании кода, чёткая архитектура позволяет нам придерживаться одной реализации взаимодействия во многих модулях.

Для начала стоит рассмотреть модели данных и их представление внутри нашего приложения.

Листинг 2.1 – Пример описания модели данных TypeORM. Файл group.entity.ts

@Entity()

export class Group {

@PrimaryGeneratedColumn()

@ApiProperty({

example: 1,

description: 'Group id',

})

id: number;

@Column({ unique: true })

name: string;

@Column({ type: 'smallint' })

course: number;

@ManyToOne(() => Specialization, (specialization) => specialization.groups)

specialization: Specialization;

@OneToMany(() => Student, (student) => student.group)

students: Array<Student>;

@OneToMany(

() => CurrentDiscipline,

(currentDiscipline) => currentDiscipline.group,

)

currentDisciplines: Array<CurrentDiscipline>;

};

Второй шаг, после создания модели данных – инициализация сервисов для взаимодействия с нашими доменами. Для этого был реализован базовый сервис (Приложение А.1), который предоставляет возможность использования CRUD операций над данными, инжектируя репозиторий взаимодействия, который делегирует обязанность разработчика менеджмента соединения с базами данных и написания запросов. Класс BaseTypeORMSevice является провайдером, его методы:

* BaseTypeORMSevice.create – создание новой сущности;
* BaseTypeORMSevice.findAll – осуществляет выборку сущностей;
* BaseTypeORMSevice.find – осуществляет выборку сущностей, отобранных по переданным критериям;
* BaseTypeORMSevice.findOne – осуществляет выборку сущности, соответствующую переданным критериям;
* BaseTypeORMSevice.findById – осуществляет выборку сущности, уникальный идентификатор которой соответствует переданному;
* BaseTypeORMSevice.update – осуществляет обновление переданных полей сущности;
* BaseTypeORMSevice.delete – удаляет сущность, уникальный идентификатор которой передан в метод;
* BaseTypeORMSevice.getRepository – возвращает экземпляр репозитория сущности для прямого взаимодействия.

Так как BaseTypeORMSevice реализует базовые CRUD операции и использует дженерики, позволяя просто передать в конструктор инжектированный репозиторий, остальные сервисы модулей расширяют его, расширяя и сужая определённые сервисы под необходимую типизацию, либо расширяя специфическими для модуля операциями. При этом сохраняя возможность использования строго типизированных методов базового класса, что сокращает дубликацию кода разительно, поддерживая принципы DRY (Don’t repeat yourself).

Листинг 2.2 – Сервис модуля группы.

@Injectable()

export class GroupsService extends BaseTypeORMService<Group> {

constructor(

@InjectRepository(Group)

private readonly groupRepository: Repository<Group>,

) {

super(groupRepository);

}

async create(

createGroupInputDto: CreateGroupInputDto,

specialization: Specialization,

) {

const group = this.groupRepository.create({

...createGroupInputDto,

specialization,

});

return this.groupRepository.save(group);

}

}

Третий шаг реализации модуля заключается в создании класса контроллера для обработки HTTP-запросов (Приложение А.2). В контроллер мы инжектируем сервис группы. С помощью декораторов можно указать пути конечных путей, определить параметры эндпоинтов, вернуть ответ ошибки при получении некосистентных данных.

Тела запросов типизированы с помощью DTO (Data Transfer Object), это позволяет определить валидационные схемы в удобном формате (листинг 2.3).

Листинг 2.3 – Описание DTO создания группы студентов

export class CreateGroupInputDto {

@IsString()

@IsNotEmpty()

@ApiProperty()

name: string;

@IsNumber()

@IsInt()

@IsPositive()

@ApiProperty()

course: number;

@IsNumber()

@IsInt()

@IsPositive()

@ApiProperty()

specializationId: number;

}

Декораторы скрывают реализацию валидации данных, но мы можем создать собственные декораторы, если функционал предоставленных не избыточен.

Также декораторы используются в определении необходимых ролей для доступа к функционалу. А проверка этих ролей осуществляется пользовательским guard (Листинг 2.4), который позволяет создать и внедрить Nest из коробки.

Листинг 2.4 – Пример пользовательского гварда

@Injectable()

export class UserRoleGuard implements CanActivate {

constructor(private reflector: Reflector) {}

canActivate(context: ExecutionContext): boolean {

const requiredUserRoles = this.reflector.getAllAndOverride<UserRole[]>(

USER\_ROLE\_KEY,

[context.getHandler(), context.getClass()],

);

if (!requiredUserRoles) {

return true;

}

const request = context.switchToHttp().getRequest();

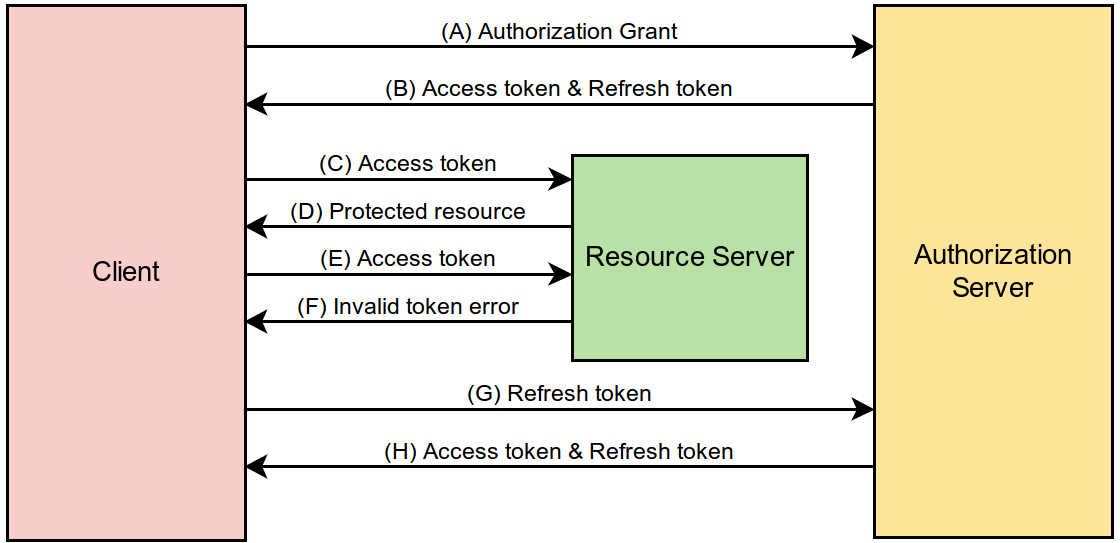
return requiredUserRoles.includes(request.user.role);

}

}

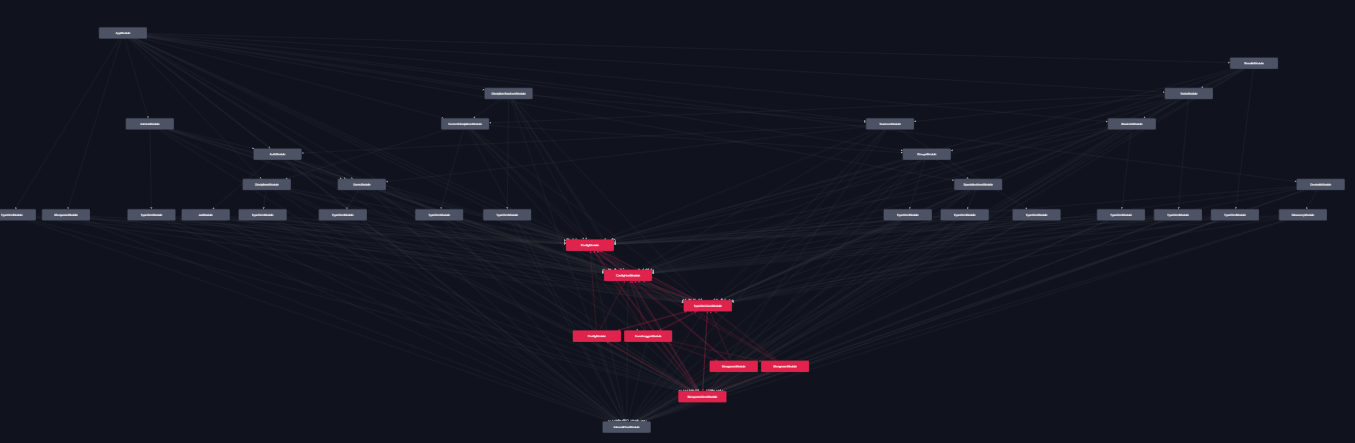
Если верификация не прошла успешно, то серверное приложение вернёт ответ ошибки с кодом 401, что сигнализирует о том, что пользователь не может получить доступ к исполнению запрашиваемых услуг из-за недействительной аутентификации. Если пользователь обладает токеном обновления, который всё ещё является актуальным, то он может избежать повторного процесса аутентификации, обновив свой токен доступа (Рисунок 2.4)

Рисунок 2.4 – Алгоритм авторизации пользователя через токены доступа и обновления



Последний шаг реализации состоит из подключения контроллера и сервиса в модуль группы и импортирования модуля группы в корневой модуль приложения.

Подобным образом реализуются все остальные модули приложения

Рисунок 2.3 – Граф связей модулей приложения

## 2.3 Расширение инструментария приложения

После реализации базовых контроллеров для REST API, можно добавить функционал для тестирования наших запросов. Спецификация OpenAPI позваляет использовать автодокументирование структур взаимодействия с API.

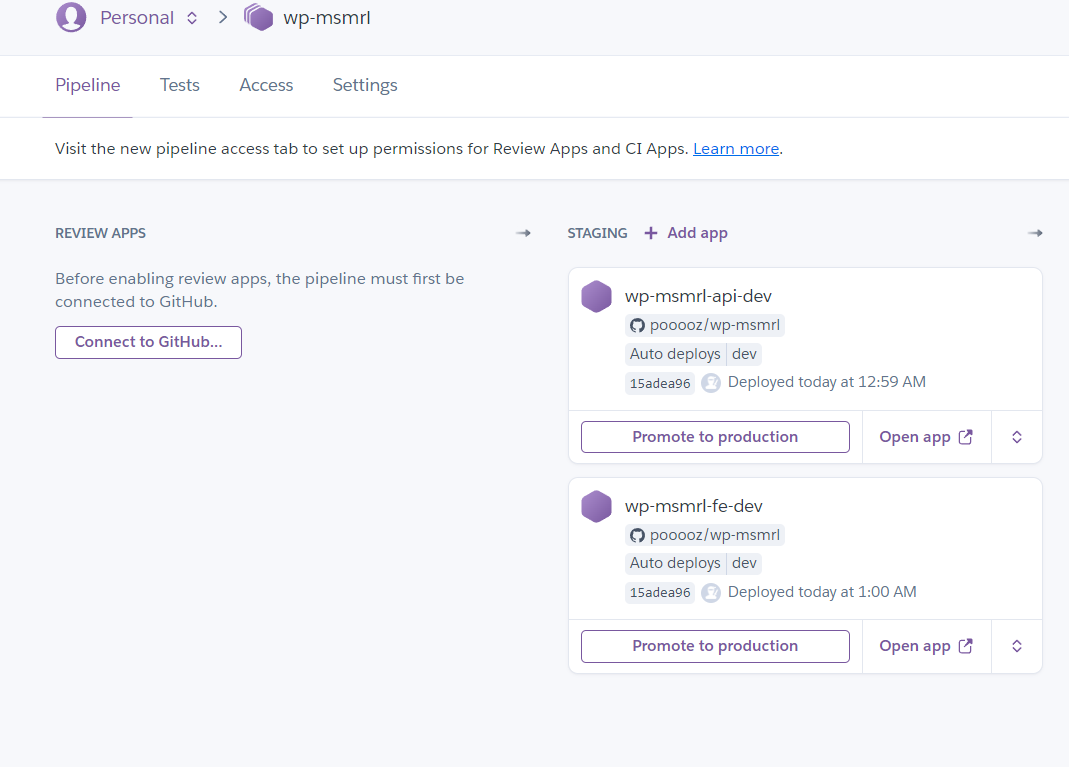
Спецификация OpenAPI — это независимый от языка формат определения, используемый для описания RESTful API. Nest предоставляет специальный модуль, который позволяет генерировать такую спецификацию с помощью декораторов

DocumentBuilder помогает структурировать базовый документ, соответствующий спецификации OpenAPI. Он предоставляет несколько методов, которые позволяют задавать такие свойства, как заголовок, описание, версия и т. д. Чтобы создать полный документ (со всеми определенными маршрутами HTTP), мы используем метод createDocument() класса SwaggerModule. Этот метод принимает два аргумента: экземпляр приложения и объект параметров Swagger. В качестве альтернативы мы можем предоставить третий аргумент, который должен иметь тип SwaggerDocumentOptions. Описание конфигурации происходит в входной точке приложения.

Описанию поддаются любые передаваемые данные. Добавим описание возвращаемых типов, DTO, краткое описание назначение эндпоинтов (Приложение А.3).

После отправки изменений в удалённый репозиторий можем видеть, что приложение успешно размещено и может быть доступно по внешнему адресу. Раннее настроенная конфигурация успешно завершила свою работу.

Рисунок 2.4 – Размещение приложения на удалённом сервере



# Заключение

Данная работа посвящена изучению устройства и архитектуры веб-приложений, возможности конструирования и создания собственных программных продуктов.

В ходе работы были проанализированы актуальные подходы проектирования архитектуры крупных веб-приложений. Изучение платформ, предоставляющих нативную реализацию архитектурных подходов, позволило собрать набор статистических данных. В ходе сравнительного анализа полученных данных были выявлены результаты, определяющие наиболее актуальные технологии, что является обоснованием их выбора для изучения и использования. Опираясь на изученное устройство отобранных технологий, была разработана архитектура и структура собственного приложения, выбранные технологии использованы в миграции существующего приложения на NestJS.

Необходимо отметить результативность и эффективность использования актуальных средств разработки в реализации веб-приложений, широкий спектр их применения, активное развитие и поддержку. Использование фреймворка позволило реализовать приложение в соответствии с масштабируемой архитектурой. Такое проектирование позволит раздробить монолитное приложение на микросервисы, задействовать внешние сервисы, интегрировать разделение назначения запросов и команд с помощью предоставляемого функционала. Улучшить производительность приложения под нагрузкой, подключив автоматическое расширение и создание нескольких экземпляров приложение с балансировкой трафика, интегрировать существующие домены в другие сервисы и приложения.

Итогом работы стали результаты обработки данных, теоретические знания, их апробирование в миграции веб-приложения. Сравнительная таблица особенностей технологий. Продуктом работы является функционирующее веб-приложение (Приложение Б). Также была проведена работа по систематизации знаний, результаты которой применимы в дальнейшем улучшении программной реализации приложения.

Таким образом, задачи, поставленные в курсовом проекте выполнены в полной мере. Установленные цели достигнуты.

# Список использованных источников

1. NestJS documentation [Электронный ресурс]. – Режим доступа: https://docs.nestjs.com/ - Дата доступа: 03.01.2024.
2. HTTP – Режим доступа: <https://developer.mozilla.org/en-US/docs/Web/HTTP> [Электронный ресурс]. - Дата доступа: 03.01.2024.
3. TypeORM [Электронный ресурс]. – Режим доступа: https://typeorm.io/ – Дата доступа: 03.01.2024.
4. Эванс, Э. Предметно-ориентированное проектирование (DDD). Структуризация сложных программных систем / Э. Эванс. – 2020. – 448 с.
5. CQRS [Электронный ресурс]. – Режим доступа: https://bool.dev/blog/detail/pattern-cqrs-i-event-sourcing – Дата доступа: 03.01.2024.
6. Test-Driven design [Электронный ресурс]. – Режим доступа: <https://www.browserstack.com/guide/what-is-test-driven-development> - Дата доступа: 03.01.2024.
7. State of JS 2021 [Электронный ресурс]. – Режим доступа: <https://2021.stateofjs.com/en-US/> - Дата доступа: 03.01.2024.
8. Что такое NoSQL? [Электронный ресурс]. – Режим доступа: <https://aws.amazon.com/ru/nosql/> - Дата доступа: 03.01.2024.
9. RFC-7519 [Электронный ресурс]. – Режим доступа: <https://www.rfc-editor.org/rfc/rfc7519> - Дата доступа: 03.01.2024.
10. PostgreSQL Documentation [Электронный ресурс]. – Режим доступа: https://www.postgresql.org/docs/ - Дата доступа: 03.01.2024.
11. Node.js Documentation [Электронный ресурс]. – Режим доступа: <https://nodejs.org/en/docs/> - Дата доступа: 03.01.2024.
12. Express Documentation [Электронный ресурс]. – Режим доступа: <https://expressjs.com/> - Дата доступа: 03.01.2024.
13. Cross-Origin Resource Sharing (CORS) [Электронный ресурс]. – Режим доступа: <https://developer.mozilla.org/ru/docs/Web/HTTP/CORS> - Дата доступа: 03.01.2024.
14. Passport.js Documentation [Электронный ресурс]. – Режим доступа: <https://www.passportjs.org/docs/> - Дата: 03.01.2024.
15. TypeScript Documentation [Электронный ресурс]. – Режим доступа: <https://www.typescriptlang.org/docs/> - Дата доступа: 03.01.2024.
16. OpenAPI Specification [Электронный ресурс]. – Режим доступа: https://swagger.io/specification/ - Дата доступа: 03.01.2024.
17. Axios Documentation [Электронный ресурс]. – Режим доступа: <https://axios-http.com/ru/docs/intro> - Дата доступа: 03.01.2024.
18. Heroku documentation [Электронный ресурс]. – Режим доступа: https://devcenter.heroku.com/categories/reference - Дата доступа: 03.01.2024.
19. Heroku Procfile [Электронный ресурс]. – Режим доступа: https://devcenter.heroku.com/articles/procfile - Дата доступа: 03.01.2024.
20. Passport auth in NestJS [Электронный ресурс]. – Режим доступа: https://ownid.com/blog/how-to-implement-passport-js-in-nest-js-an-easy-tutorial/ - Дата доступа: 03.01.2024.

Приложение А

Листинг А.1 – Базовый сервис приложения

@Injectable()

export abstract class BaseTypeORMService<

Entity extends { id: number; [key: string]: any },

> {

constructor(private readonly entityRepository: Repository<Entity>) {}

async create(createInputDto: DeepPartial<Entity>, ...rest: unknown[]) {

const entity = this.entityRepository.create(createInputDto);

return this.entityRepository.save(entity);

}

async findAll(

relations?: FindOptionsRelations<Entity>,

select?: FindOptionsSelect<Entity>,

) {

return this.entityRepository.find({

relations,

select,

});

}

async findOne(

where?: FindOptionsWhere<Entity>,

relations?: FindOptionsRelations<Entity>,

select?: FindOptionsSelect<Entity>,

) {

return this.entityRepository.findOne({

where,

relations,

select,

});

}

async find(

where?: FindOptionsWhere<Entity>,

relations?: FindOptionsRelations<Entity>,

select?: FindOptionsSelect<Entity>,

) {

return this.entityRepository.find({

where,

relations,

select,

});

}

async findById(

id: number,

relations?: FindOptionsRelations<Entity>,

select?: FindOptionsSelect<Entity>,

): Promise<Entity | null> {

return this.entityRepository.findOne({

where: {

id,

} as FindOptionsWhere<Entity>,

relations,

select,

});

}

async update(id: number, updateInputDto: DeepPartial<Entity>) {

return this.entityRepository.update(id, {

...updateInputDto,

id,

});

}

async remove(id: number) {

return this.entityRepository.delete(id);

}

public getRepository() {

return this.entityRepository;

}

}

Листинг А.2 – Контроллер модуля группы

@ApiBearerAuth()

@UseGuards(UserRoleGuard)

@ApiTags('groups')

@Controller('groups')

export class GroupsController {

constructor(

private readonly groupsService: GroupsService,

private readonly specializationsService: SpecializationsService,

) {}

@Post()

@RequiredUserRoles(UserRole.Admin)

@ApiOperation({ summary: 'Create group' })

@ApiResponse({

status: 201,

description: 'Group created',

type: Group,

})

async create(

@Body() createGroupInputDto: CreateGroupInputDto,

): Promise<Group> {

const specialization = await this.specializationsService.findById(

createGroupInputDto.specializationId,

);

if (!specialization) {

throw new BadRequestException(

`Specialization with id (${createGroupInputDto.specializationId}) does not exist`,

);

}

return this.groupsService.create(createGroupInputDto, specialization);

}

@Get()

@RequiredUserRoles(UserRole.Admin, UserRole.Teacher)

@ApiOperation({ summary: 'Find all groups' })

@ApiResponse({

status: 200,

description: 'All groups',

type: Array<Group>,

})

findAll() {

return this.groupsService.findAll({ specialization: true });

}

@Get(':id')

@RequiredUserRoles(UserRole.Admin, UserRole.Teacher)

@ApiOperation({ summary: 'Find group' })

@ApiResponse({

status: 200,

description: 'Find group by id',

type: Group,

})

findOne(@Param('id') id: string) {

return this.groupsService.findById(Number(id), {

specialization: true,

students: true,

});

}

@Patch(':id')

@RequiredUserRoles(UserRole.Admin)

async update(

@Param('id') id: string,

@Body() updateGroupInputDto: UpdateGroupInputDto,

) {

const { specializationId, ...updateGroupInputDtoRest } =

updateGroupInputDto;

const updateDto: DeepPartial<Group> = { ...updateGroupInputDtoRest };

if (specializationId) {

const specialization = await this.specializationsService.findById(

specializationId,

);

if (!specialization) {

throw new BadRequestException(

`Specialization with id (${specializationId}) does not exist`,

);

}

updateDto.specialization = specialization;

}

return this.groupsService.update(Number(id), updateDto);

}

@Delete(':id')

@RequiredUserRoles(UserRole.Admin)

async remove(@Param('id') id: string) {

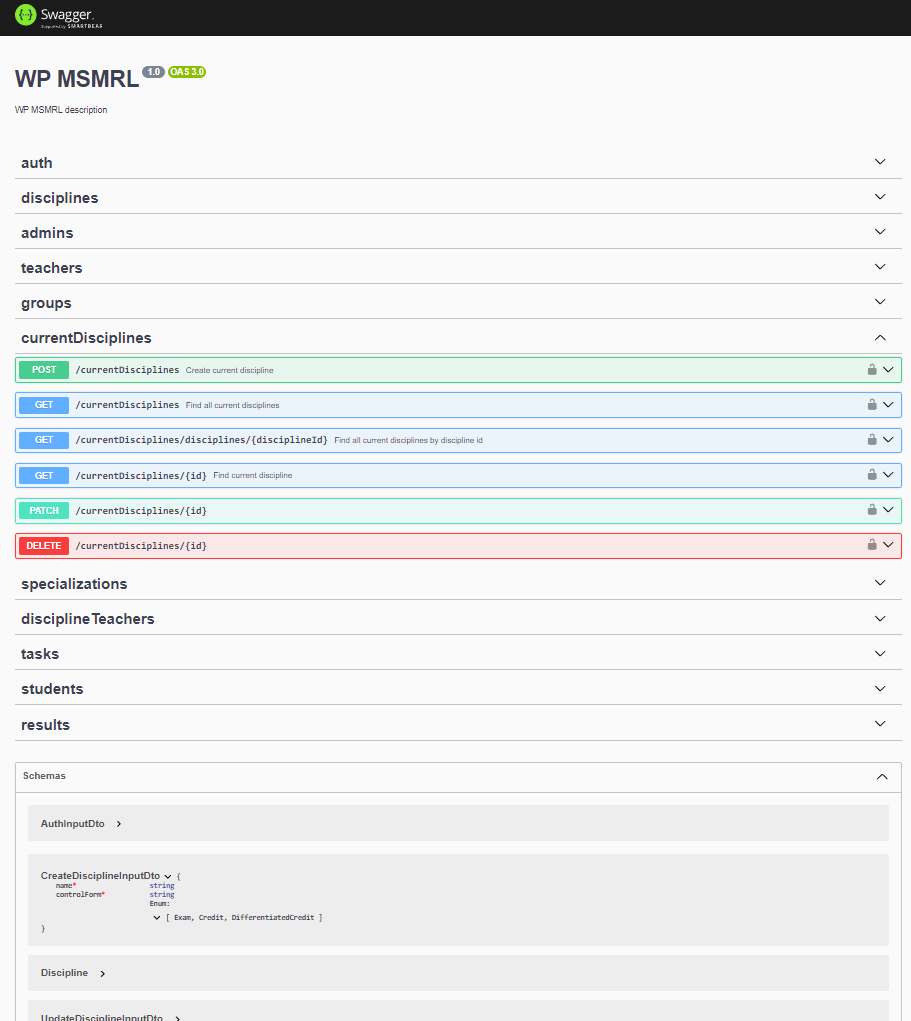
return this.groupsService.remove(Number(id));

}

});

};

Рисунок А.3 – Описание Swagger API



Приложение Б

Размещение исходного кода веб-приложения

Размещение исходного кода клиентского и серверного приложения: https://github.com/pooooz/wp-msmrl

Размещение исходного кода серверного приложения: https://wp-msmrl-fe-dev-c96200e1855b.herokuapp.com/sign-in