**Документация по бэкенду**

Бэкенд представляет собой RESTful API для управления персонажами, чатами и сообщениями в приложении, связанном с генерацией текста с использованием языковых моделей (LLM). API предоставляет функциональность для регистрации и аутентификации пользователей, создания и управления персонажами, а также ведения чатов с персонажами. Взаимодействие между компонентами системы организовано через контроллеры, репозитории, сервисы и мапперы.

При запуске проекта в Debug открывается **Swagger**.

**Основные компоненты системы**

**1. Пользователи (AppUser)**

Функционал:

* Регистрация нового пользователя.
* Аутентификация пользователя (логин) с выдачей JWT токена.
* Управление персонажами, созданными пользователем.

Взаимодействие:

* **AccountController** обрабатывает запросы на регистрацию и аутентификацию.
* **UserManager** (из Identity Framework) управляет пользователями и их ролями.
* **TokenService** генерирует JWT токен для аутентификации.

**2. Персонажи (Character)**

Функционал:

* Создание нового персонажа с указанием имени и системного промпта.
* Получение списка всех персонажей с возможностью фильтрации и пагинации.
* Получение персонажа по идентификатору.
* Обновление данных персонажа.
* Удаление персонажа.

Взаимодействие:

* **CharacterController** обрабатывает запросы, связанные с персонажами.
* **CharacterRepository** взаимодействует с базой данных для выполнения CRUD операций.
* **CharacterMapper** преобразует данные между сущностями и DTO.

**3. Чаты (ChatHistory)**

Функционал:

* Создание нового чата с указанием персонажа.
* Получение списка всех чатов пользователя.
* Получение чата по идентификатору с сообщениями.
* Удаление чата.

Взаимодействие:

* **ChatController** обрабатывает запросы, связанные с чатами.
* **ChatRepository** взаимодействует с базой данных для выполнения CRUD операций.
* **ChatMapper** преобразует данные между сущностями и DTO.

**4. Сообщения (Message)**

Функционал:

* Добавление нового сообщения в чат.
* Обновление существующего сообщения.
* Удаление сообщения.
* Генерация ответа от языковой модели на основе сообщений и системного промпта персонажа.

Взаимодействие:

* **ChatController** обрабатывает запросы, связанные с сообщениями.
* **ChatRepository** управляет сообщениями в базе данных.
* **ChatService** взаимодействует с внешним API языковой модели для генерации ответа.
* **ChatMessageMapper** преобразует данные между сущностями и DTO.

**5. Языковая модель (LLM)**

Функционал:

* Генерация текстового ответа на основе истории сообщений и системного промпта персонажа.

Взаимодействие:

* **LLMController** принимает запросы на генерацию ответа.
* **ChatService** отправляет запросы к внешнему API языковой модели (например, FastAPI) и обрабатывает ответы.

**6. Веб-сокеты (WebSocket)**

Функционал:

* Реальное время в чатах: отправка и получение сообщений через веб-сокеты.

Взаимодействие:

* **ChatWebSocketHandler** управляет соединениями по веб-сокетам.
* **ChatRepository** и **ChatService** используются для обработки сообщений и генерации ответов.

**Описание функционала и взаимодействия**

**1. Регистрация и аутентификация**

* **Функционал**:
  + Пользователь регистрируется, указывая имя, email и пароль.
  + После регистрации пользователь может войти в систему, используя свои учетные данные.
  + При успешной аутентификации пользователь получает JWT токен, который используется для доступа к защищенным ресурсам.
* **Взаимодействие**:
  + **AccountController** принимает запросы на регистрацию и аутентификацию.
  + **UserManager** создает нового пользователя и проверяет учетные данные.
  + **TokenService** генерирует JWT токен.

**2. Управление персонажами**

* **Функционал**:
  + Пользователь может создавать персонажей, указывая имя и системный промпт.
  + Пользователь может просматривать, обновлять и удалять своих персонажей.
  + Доступен поиск и фильтрация персонажей.
* **Взаимодействие**:
  + **CharacterController** обрабатывает запросы.
  + **CharacterRepository** выполняет операции с базой данных.
  + **CharacterMapper** преобразует данные между сущностями и DTO.

**3. Управление чатами**

* **Функционал**:
  + Пользователь может создавать чаты с персонажами.
  + Пользователь может просматривать свои чаты и удалять их.
  + В чатах можно отправлять и получать сообщения.
* **Взаимодействие**:
  + **ChatController** обрабатывает запросы.
  + **ChatRepository** управляет чатами и сообщениями в базе данных.
  + **ChatService** генерирует ответы от языковой модели.

**4. Генерация ответов от языковой модели**

* **Функционал**:
  + Когда пользователь отправляет сообщение в чат, система отправляет запрос к языковой модели.
  + Языковая модель генерирует ответ на основе истории сообщений и системного промпта персонажа.
* **Взаимодействие**:
  + **ChatService** отправляет запрос к внешнему API языковой модели.
  + Ответ модели сохраняется в базе данных как новое сообщение.

**5. Веб-сокеты для реального времени**

* **Функционал**:
  + Пользователь может открыть веб-сокет соединение для чата.
  + Сообщения отправляются и получаются в реальном времени.
* **Взаимодействие**:
  + **ChatWebSocketHandler** управляет соединениями.
  + **ChatRepository** и **ChatService** используются для обработки сообщений.

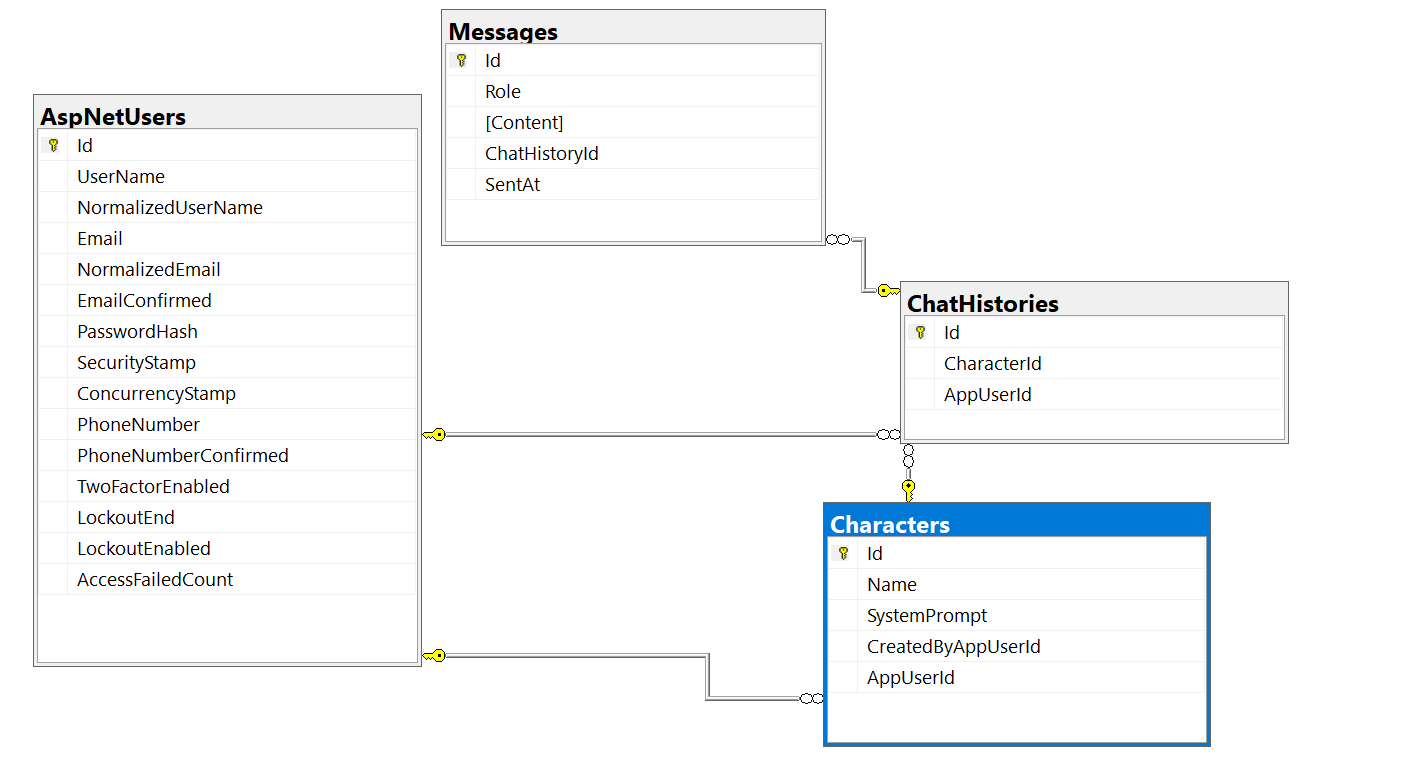
### База данных

#### Обзор

База данных приложения построена на основе **Entity Framework Core** и использует **Microsoft SQL Server** в качестве СУБД. Взаимодействие с базой данных осуществляется через контекст ApplicationDBContext, который наследуется от IdentityDbContext<AppUser>. Это позволяет интегрировать систему аутентификации и авторизации на основе Identity Framework с пользовательскими сущностями, такими как персонажи, чаты и сообщения.

#### Основные сущности базы данных

1. **AppNetUser (Пользователь)**:
   * Наследуется от IdentityUser.
   * Содержит список персонажей (Characters), созданных пользователем.
   * Используется для аутентификации и авторизации.
2. **Character (Персонаж)**:
   * Содержит информацию о персонаже: имя (Name), системный промпт (SystemPrompt), идентификатор создателя (CreatedByAppUserId).
   * Связан с пользователем через внешний ключ CreatedByAppUserId.
3. **ChatHistory (История чата)**:
   * Содержит информацию о чате: идентификатор персонажа (CharacterId), идентификатор пользователя (AppUserId), список сообщений (Messages).
   * Связан с пользователем и персонажем через внешние ключи.
4. **Message (Сообщение)**:
   * Содержит информацию о сообщении: роль (Role), содержание (Content), идентификатор чата (ChatHistoryId), время отправки (SentAt).
   * Связан с историей чата через внешний ключ ChatHistoryId.



#### Конфигурация базы данных

Конфигурация базы данных задается в файле appsettings.json:

|  |
| --- |
| {  "ConnectionStrings": {  "DefaultConnection": "Data Source=DESKTOP-T3ERBIR\\SQLEXPRESS;Initial Catalog=llmchat;Integrated Security=True;Connect Timeout=30;Encrypt=False;TrustServerCertificate=False;ApplicationIntent=ReadWrite;MultiSubnetFailover=False"  }  } |

* **DefaultConnection**: Строка подключения к базе данных SQL Server.
  + Data Source: Имя сервера (например, DESKTOP-T3ERBIR\\SQLEXPRESS).
  + Initial Catalog: Имя базы данных (например, llmchat).
  + Integrated Security: Использование встроенной аутентификации Windows.

#### Контекст базы данных (ApplicationDBContext)

Контекст базы данных ApplicationDBContext наследуется от IdentityDbContext<AppUser> и управляет всеми сущностями приложения:

#### **Миграции**

Entity Framework Core использует миграции для управления изменениями в структуре базы данных. Для создания и применения миграций используются следующие команды:

**1. Применение миграции**:

|  |
| --- |
| dotnet ef migrations add <MigrationName> |

**2. Применение миграции**:

|  |
| --- |
| dotnet ef database update |

### Тестирование

#### Обзор

Тесты для проекта находятся в директории D:\023course4\arx\LlmCharacterChat\llmChatTests. Они написаны с использованием следующих технологий:

* **xUnit** — фреймворк для модульного тестирования.
* **Moq** — библиотека для создания mock-объектов, используемых в тестах.
* **coverlet.collector** — инструмент для измерения покрытия кода тестами.

#### Технологии

* **xUnit**: Основной фреймворк для написания тестов.
* **Moq**: Используется для создания mock-объектов, которые имитируют поведение зависимостей (например, репозиториев и сервисов).
* **coverlet.collector**: Инструмент для измерения покрытия кода тестами. В проекте используется для оценки степени покрытия тестами.

#### Покрытие тестами

Степень покрытия тестами составляет **78%**. Это означает, что 78% кода приложения проверяется автоматическими тестами, что является хорошим показателем для обеспечения стабильности и надежности системы.

#### Структура тестов

Тесты разделены на несколько категорий:

1. **Тесты контроллеров**:
   * Проверяют корректность работы API-методов (например, создание персонажа, добавление сообщения в чат).
   * Используют mock-объекты для имитации работы репозиториев и сервисов.
2. **Тесты сервисов**:
   * Проверяют бизнес-логику, реализованную в сервисах (например, генерация ответа от языковой модели).
3. **Тесты репозиториев**:
   * Проверяют корректность работы с базой данных (например, сохранение и загрузка данных).

И т.д.

#### Примеры тестов

1. **Тест контроллера CharacterController**:
   * Проверяет, что метод CreateCharacter возвращает корректный результат при успешном создании персонажа.
   * Использует mock-объекты для UserManager и ICharacterRepository.
2. **Тест сервиса ChatHistoryService**:
   * Проверяет, что метод GetChatHistoryByIdAsync возвращает историю чата с сообщениями.
   * Использует mock-объект для IChatHistoryRepository.
3. **Тест репозитория ChatHistoryRepository**:
   * Проверяет, что метод SaveChatHistoryAsync корректно сохраняет данные в базу данных.
   * Использует InMemoryDatabase для изоляции тестов от реальной базы данных.