Министерство просвещения Приднестровской Молдавской Республики

ГОУ СПО «Тираспольский техникум информатики и права»

**КУРСОВАЯ РАБОТА**

по МДК.01.01 Технология разработки и защиты баз данных

на тему: «Разработка системы электронной почты»

**Выполнил обучающийся**:

Редкозубов Владислав Романович

**Специальность** 2.09.02.07

Технология разработки и защиты баз данных

**Руководитель:**

Преподаватель

информационных дисциплин

Надькин Леонид Юрьевич

\_\_\_\_\_\_\_\_\_\_\_\_\_

(оценка)

«\_\_\_\_» «\_\_\_\_\_\_\_\_\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_

дата подпись руководителя

Тирасполь, 2024

**ОГЛАВЛЕНИЕ**

[ВВЕДЕНИЕ 3](#_30j0zll)

[Глава 1. Теоретическая часть разработки программного обеспечения 5](#_1fob9te)

[1.1. Описание предметной области 5](#_3znysh7)

[1.2. Анализ существующих разработок 6](#_2et92p0)

[1.3. Описание выявленных достоинств и недостатков 9](#_tyjcwt)

[1.4. Используемый язык программирования C# 10](#_3dy6vkm)

[1.5. Общая постановка задачи 11](#_1t3h5sf)

[ЗАКЛЮЧЕНИЕ 14](#_4d34og8)

# **ВВЕДЕНИЕ**

**Актуальность** данной темы заключается в необходимости повышения эффективности и конкурентоспособности предприятий через внедрение собственной системы электронной почты, что позволит оптимизировать бизнес-процессы предприятия.

**Целью** данной курсовой работы является разработка системы электронной почты для улучшения операционной эффективности и повышения уровня безопасности обмена информацией в организации.

Для выполнения курсовой работы были поставлены следующие **задачи**:

1. Реализация передачи сообщений средствами электронной почты
2. Управление сообщениями и данными пользователей
3. Реализация файловой системы

**Объектом** данной работы является электронная почта, то есть организация, осуществляющая отправку электронных сообщений.

**Предметом** данной работы является разработка информационной системы для управления электронной почтой. Это включает в себя проектирование и создание программного обеспечения, которое будет оптимизировать основные процессы работы с электронными сообщениями, такие как учет почтовых ящиков, управление контактами. Разработанная система будет предоставлять набор функций и возможностей для обмена информацией с использованием электронной почты.

**Метод** исследования, используемый в данной работе для электронной почты, является аналитическим и систематическим анализом.

**В первой главе** курсовой работы рассматриваются теоретические аспекты, связанные с электронной почтой, включая ее основные понятия и функции.

**Во второй главе** работы формулируется постановка задачи, где рассматриваются методы решения и предлагается описание выбранного языка программирования для разработки электронной почты. В этой главе также предоставляется описание разработанной программы.

В приложении работы представлено электронное сообщение, сформированное программой, а также приведен листинг разработанной программы.

# **ГЛАВА 1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ**

# **Описание предметной области**

**Организация** **электронной почты** специализируется на обмене электронными сообщениями и является неотъемлемой частью современного информационного обмена. В современном мире электронная почта стала ключевым средством коммуникации как в личной, так и в деловой сферах, обеспечивая быструю и удобную связь между людьми и организациями.

**Основная задача** организации электронной почты заключается в обеспечении качественного обмена электронными сообщениями, а также предоставлении надежной и безопасной платформы для передачи информации. Это включает в себя функции отправки, приема, хранения и организации сообщений.

Организация электронной почты обслуживает различные типы пользователей, включая частных лиц, предприятия, государственные учреждения и другие организации. Она также предоставляет разнообразные функциональные возможности, такие как управление контактами, календарем, задачами и файлами, для обеспечения эффективного обмена информацией.

**Предметная область электронной почты** включает в себя различные процессы и операции, связанные с обменом электронными сообщениями и управлением почтовыми ящиками. Основные функции электронной почты включают**:**

1. Управление почтовыми ящиками: Регистрация и систематический учет почтовых ящиков пользователей, включая информацию о владельцах, адресах электронной почты, статусе ящиков и привилегиях доступа.
2. Управление заказами и планирование рассылок: Создание и отправка электронных сообщений, планирование рассылок, определение времени отправки и целевой аудитории, а также анализ эффективности кампаний.
3. Фильтрация и обработка входящей почты: Автоматическая фильтрация спама и вредоносных сообщений, организация входящей почты в папки и категории, автоматическое оповещение о важных сообщениях.
4. Учет контактов и адресной книги: Хранение информации о контактах пользователей, их электронных адресах, группировка контактов по категориям, возможность быстрой рассылки сообщений группам пользователей.
5. Безопасность и конфиденциальность: Защита электронных сообщений от несанкционированного доступа, шифрование данных, проверка на подлинность отправителя и цифровые подписи.

**Целью** разработки информационной системы для управления электронной почтой является повышение операционной эффективности и качества обслуживания. Информационная система позволяет автоматизировать и оптимизировать основные процессы работы с электронными сообщениями, что приводит к ускорению обработки почтовых запросов, снижению риска пропуска важных сообщений и улучшению взаимодействия с отправителями.

В следующих разделах работы будет проведен обзор существующих информационных систем для управления электронной почтой, их функциональных возможностей и особенностей реализации. Будет проведен анализ их преимуществ и недостатков, а также выявлены основные требования и потребности пользователей в данной области. На основе этого анализа будет предложено описание разработанной программы, учитывающей специфику управления электронной почтой и включающей в себя необходимый набор функций для эффективной работы.

# **Анализ существующих разработок**

Анализ различных программ электронной почты в современном секторе обмена информации имеет важное значение для понимания тенденций и достижений в этой области. Рассмотрим различные электронные почты, оценим их преимущества, функциональность и эффективность.

**Цель анализа** - выявление ключевых трендов и лучших практик в области систем электронной почты, а также оценка потенциала обеспечения удовлетворения потребностей клиентов.

Рассмотрим почтовую систему Gmail:

**Gmail** (Рис. 1) - это бесплатный почтовый сервис, разработанный компанией Google. Он предоставляет возможность отправки, приема и организации электронных сообщений, а также обладает рядом дополнительных функций, таких как фильтрация спама, интеграция с другими сервисами Google, облачное хранение вложений и доступ к почте через веб-интерфейс и мобильные приложения. Gmail является одним из самых популярных почтовых сервисов в мире и используется миллионами пользователей.

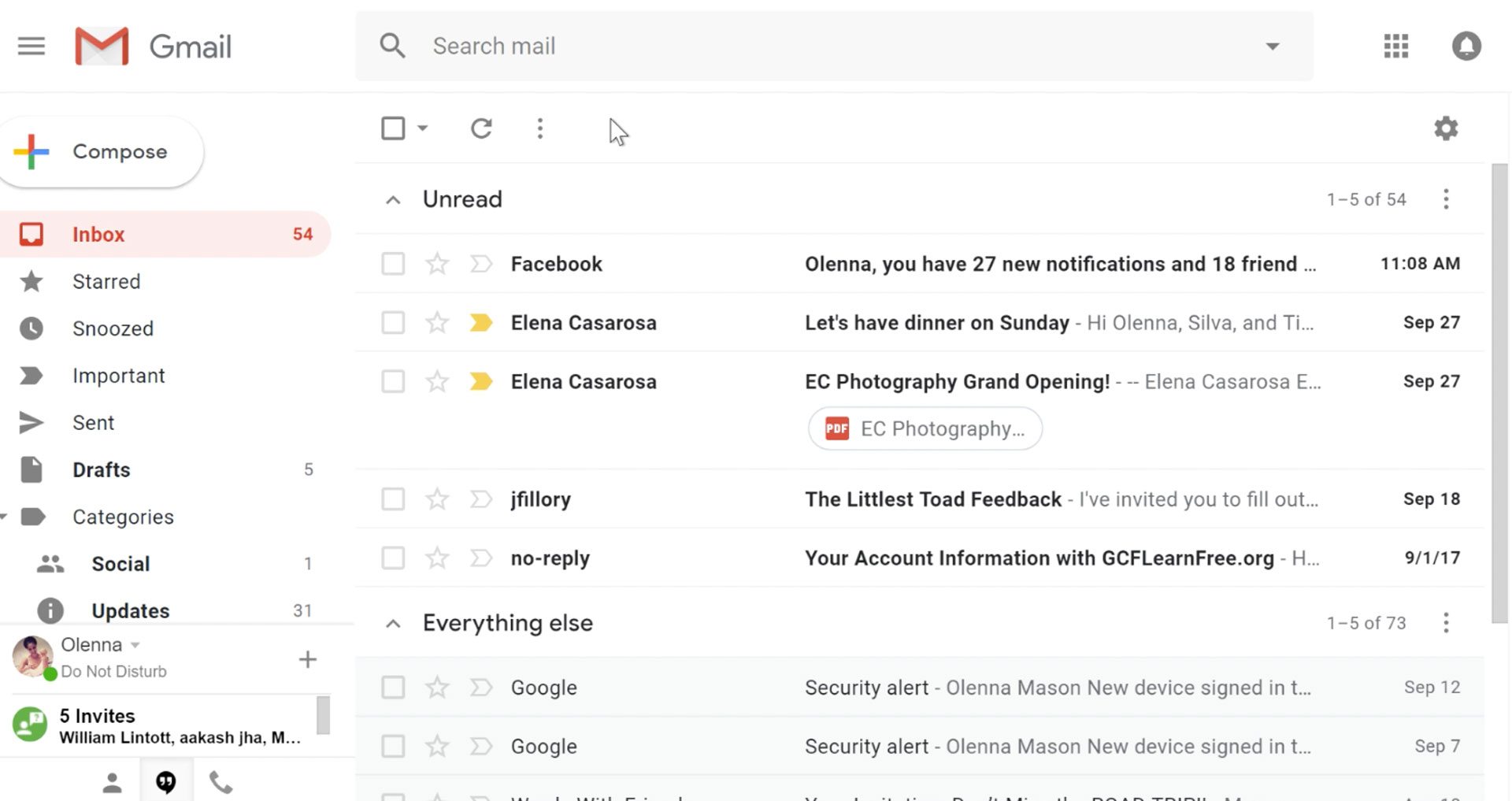


Рис. 1 Приложение «Gmail»

Программа имеет **интуитивно понятный пользовательский интерфейс** (Рис. 2), который делает использование Gmail удобным даже для непрофессионалов. Она предоставляет информацию в понятной форме и облегчает взаимодействие с системой обмена информации.

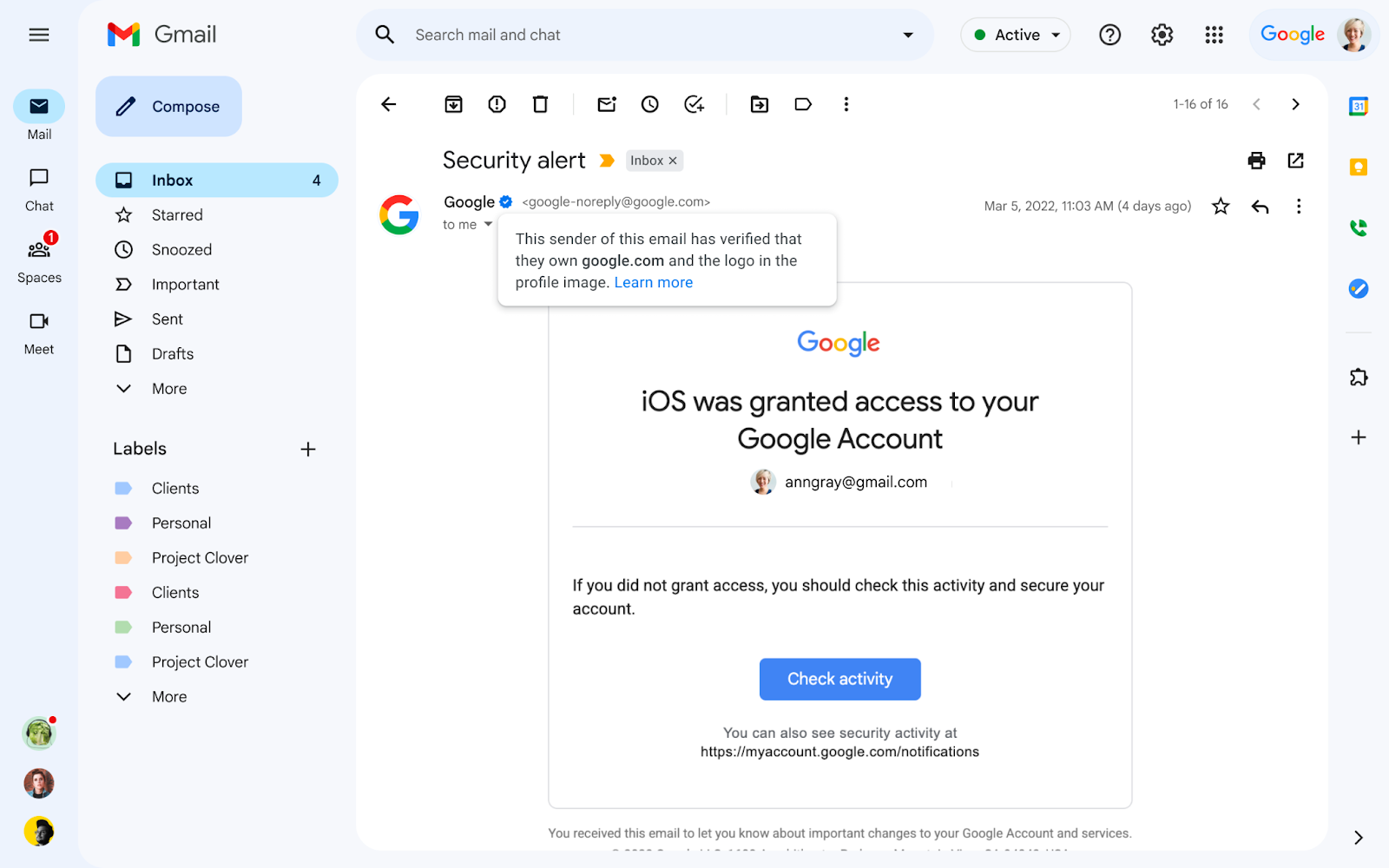


Рис. 2 Пользовательский интерфейс

**Gmail предоставляет подробные отчеты и данные** (Рис. 3) о состоянии почтового ящика, обнаруженных проблемах и выполненных действиях. Это помогает пользователям получать объективную информацию о состоянии и управлении своей электронной почтой, обеспечивая прозрачность в процессе обработки сообщений.

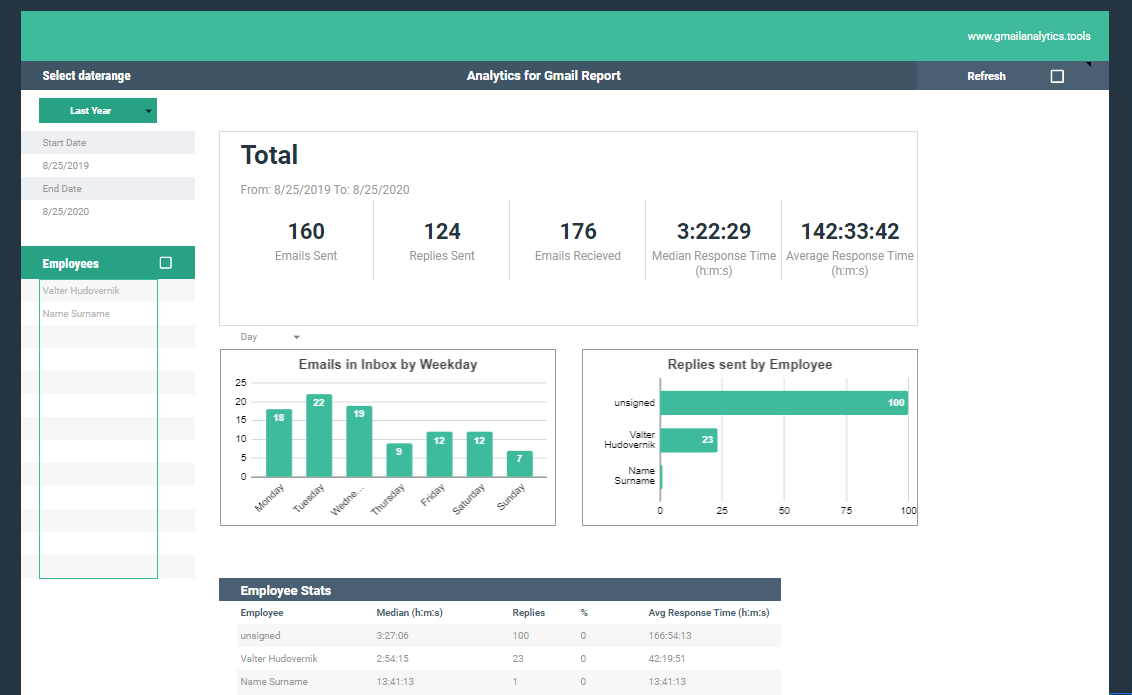


Рис. 3 Пример отчета и данных

Некоторые пользователи могут столкнуться с **проблемами в получении поддержки** или обновлений программы Gmail.

В зависимости от региона или поставщика услуги электронной почты, доступность поддержки и актуальных обновлений может быть неоднородной.

**Большим недостатком** Gmail является необходимость подключения к интернету для доступа к почтовому ящику. Пользователи должны иметь доступ к интернету и убедиться в наличии соответствующего устройства и настроенного интернет-соединения, что может потребовать дополнительных усилий и ресурсов.

Также время от времени, с выпуском новых версий и обновлений Gmail, могут возникать проблемы совместимости с различными операционными системами и устройствами. Для оптимального функционирования сервиса с новыми версиями операционных систем или на различных устройствах может потребоваться обновление программного обеспечения или использование дополнительных функций.

# **Описание выявленных достоинств и недостатков**

**Описание достоинств** программы Gmail:

1. Бесплатность и масштаб: Gmail предоставляет свои услуги абсолютно бесплатно для пользователей и обладает огромным масштабом, позволяя создавать и использовать почтовые ящики с уникальным адресом электронной почты.
2. Большой объем хранилища: Пользователям предоставляется значительное количество свободного места для хранения электронных сообщений и вложений, что позволяет сохранять большие объемы информации без необходимости регулярно удалять сообщения.
3. Интеграция с другими сервисами Google: Gmail легко интегрируется с другими сервисами Google, такими как Google Drive, Google Календарь и Google Документы, что обеспечивает удобство в работе и позволяет эффективно управлять всеми аспектами своей работы и коммуникаций через одну платформу.

**Описание недостатков** программы Gmail:

1. Ограничения конфиденциальности данных: Gmail сканирует содержимое электронных сообщений для персонализации рекламы и улучшения своих услуг, что может вызывать опасения по поводу конфиденциальности данных пользователей.
2. Ограниченные возможности работы офлайн: Несмотря на то, что Gmail предоставляет некоторые функции работы в автономном режиме через расширения и дополнительные инструменты, эти возможности ограничены и могут быть недостаточными для пользователей, которым требуется полноценная работа без подключения к интернету.
3. Ограничения на размер вложений: Gmail имеет ограничения на размер вложений в электронных сообщениях, что может быть проблематично при отправке больших файлов или документов, требующих высокой пропускной способности и времени загрузки.

# **Используемый язык программирования C#**

В процессе выполнения курсовой работы использован язык программирования C# , а также интерфейс программирования приложений Windows Forms

Язык C# - это высокоуровневый язык программирования, разработанный Microsoft. Он был представлен в 2000 году и является одним из основных языков программирования в экосистеме Microsoft.NET. C# сочетает в себе элементы объектно-ориентированного программирования (ООП) и функционального программирования, и предоставляет мощные инструменты для разработки широкого спектра приложений.

C# используется во множестве областей, включая разработку настольных приложений, веб-разработку, создание мобильных приложений, игр и многое другое. Он широко применяется для разработки приложений под операционные системы Windows, macOS и Linux.

Программа на C# может содержать различные компоненты, такие как классы, структуры данных, методы, свойства и события. Существует возможность использовать библиотеки и фреймворки, предоставленные Microsoft или сторонними разработчиками, чтобы упростить разработку и добавить дополнительные возможности к вашей программе.

C# также имеет средства для обработки исключений, взаимодействия с базами данных, работы с файлами и многими другими задачами. Язык поддерживает сборку мусора для автоматического освобождения памяти и обладает мощным инструментарием для отладки кода и создания тестов.

Одним из основных преимуществ использования C# является его интеграция с .NET Framework или .NET Core, что позволяет использовать множество готовых компонентов и библиотек для создания эффективных и масштабируемых приложений.

В целом, C# является мощным и гибким языком программирования, который позволяет создавать разнообразные программы с использованием различных платформ и технологий.

ASP.NET — это фреймворк для разработки веб-приложений, разработанный компанией Microsoft, который предоставляет инструменты для создания динамических и интерактивных веб-сайтов и веб-приложений. В рамках ASP.NET, C# используется для создания веб-приложений с помощью технологии Web Forms.

С помощью C# в ASP.NET разработчики могут создавать разнообразные веб-элементы управления, такие как кнопки, текстовые поля, списки, таблицы и другие. Существует возможность размещать их на веб-страницах и настраивать их свойства, чтобы создать интерактивные пользовательские интерфейсы.

ASP.NET позволяет разработчикам создавать веб-приложения, которые могут взаимодействовать с базами данных, обрабатывать формы, обеспечивать аутентификацию и авторизацию пользователей, а также создавать динамический контент, который обновляется без перезагрузки страницы.

ASP.NET предоставляет возможности для создания веб-приложений, включая разработку веб API (Application Programming Interface), которые позволяют взаимодействовать с приложением через интернет посредством HTTP запросов.

Веб API в ASP.NET позволяют создавать точки доступа (эндпоинты), к которым можно обращаться из любого клиентского приложения, будь то веб-приложение, мобильное приложение или даже другой сервер.

Основные концепции и возможности веб API в ASP.NET:

1. ASP.NET Web API Framework: Это фреймворк, входящий в состав ASP.NET, который облегчает создание и развертывание веб API. Он предоставляет множество инструментов для создания API с поддержкой различных форматов обмена данными, таких как JSON и XML.
2. Маршрутизация: Существует возможность определить маршруты URL, которые указывают на методы вашего API. Это делается с помощью атрибутов маршрутизации или настройками конфигурации.
3. Контроллеры: Контроллеры в ASP.NET Web API представляют собой классы, которые обрабатывают входящие HTTP запросы и возвращают соответствующие HTTP ответы. В контроллерах определяются методы, которые будут вызываться при обращении к определенным URL.
4. Модели: Существует возможность использовать модели данных для передачи информации между клиентом и сервером. Обычно это классы C# (или других поддерживаемых языков), которые отображают структуру данных, которые передаются через API.
5. Форматы обмена данными: ASP.NET Web API поддерживает различные форматы обмена данными, такие как JSON, XML и другие. Существует возможность настроить форматы сериализации и десериализации данных по выбору.
6. Атрибуты: ASP.NET Web API использует атрибуты для настройки маршрутизации, аутентификации, авторизации и других аспектов API.
7. Аутентификация и авторизация: Существует возможность реализовать механизмы аутентификации и авторизации в вашем веб API, чтобы обеспечить безопасность и контроль доступа к ресурсам.
8. Фильтры: ASP.NET Web API позволяет использовать фильтры для добавления дополнительной логики в обработку запросов и ответов, например, логирование, кэширование или обработка ошибок.
9. Документация: Хорошая документация важна для веб API. Существует возможность использовать инструменты, такие как Swagger (Рис. 4) или OpenAPI, для автоматической генерации документации на основе кода.

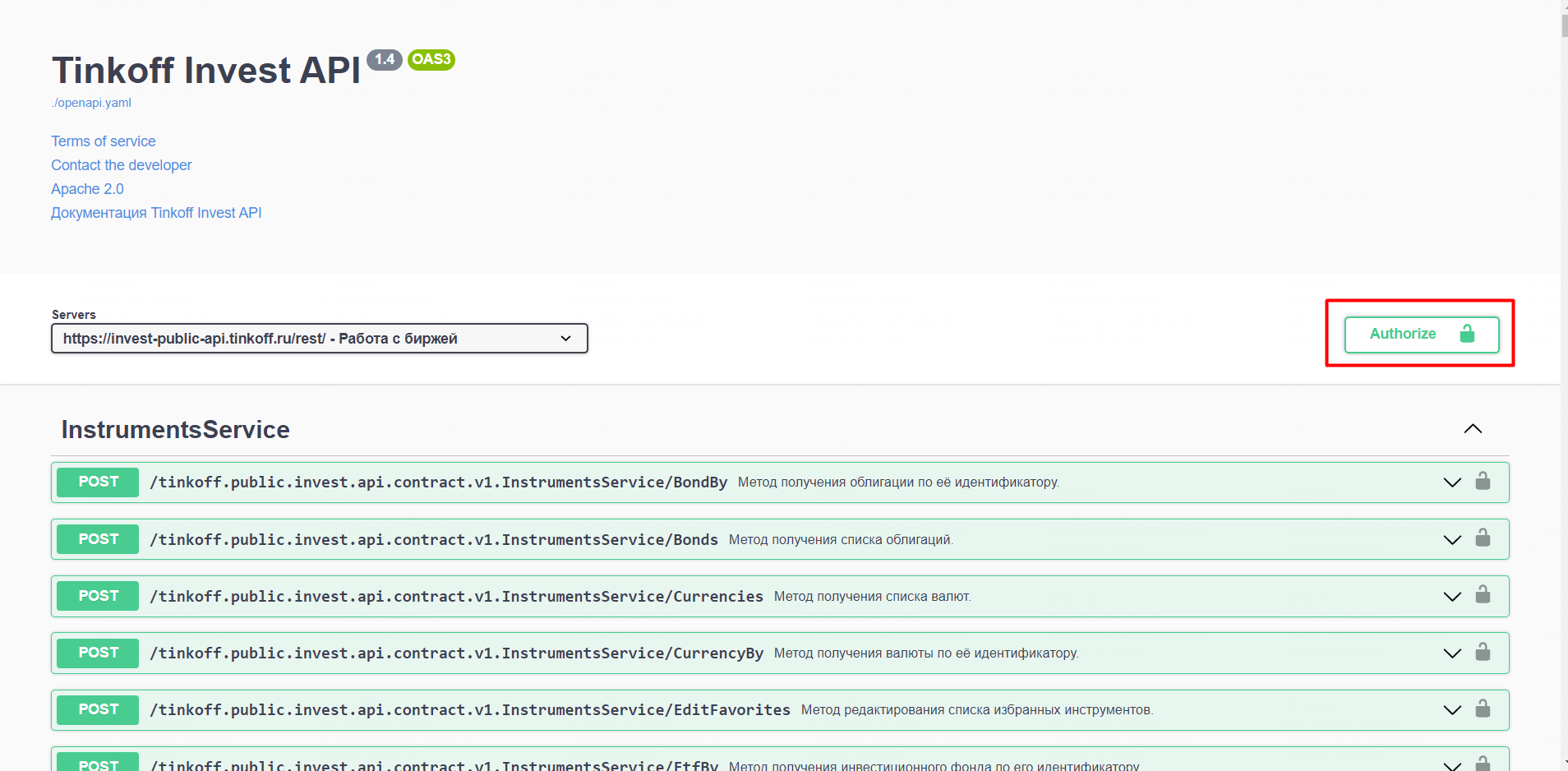


Рис. 4 Интерфейс Swagger

SQL Server - это мощная и популярная система управления реляционными базами данных (RDBMS), разработанная и распространяемая корпорацией Microsoft. Он предоставляет средства для хранения, управления и обработки данных, используемых в различных приложениях и системах. Основные возможности SQL Server:

1. Хранение и управление данными: SQL Server обеспечивает мощные средства для хранения и управления данными. Он поддерживает различные типы данных, включая числа, строки, даты, времена, географические данные и многие другие.
2. Язык SQL: SQL Server поддерживает SQL (Structured Query Language), стандартный язык для управления и манипулирования данными в реляционных базах данных. SQL используется для выполнения запросов данных, создания и изменения таблиц, а также для управления доступом к данным.
3. Транзакционная поддержка: SQL Server обеспечивает поддержку транзакций, что позволяет группировать операции базы данных в логические единицы работы. Это обеспечивает целостность данных и защиту от потери данных в случае сбоев.
4. Масштабируемость и производительность: SQL Server обеспечивает многоуровневую архитектуру, которая позволяет масштабировать базы данных и обрабатывать большие объемы данных. Он также предоставляет инструменты для оптимизации производительности запросов и базы данных.
5. Безопасность: SQL Server обеспечивает многоуровневую систему безопасности, которая позволяет управлять доступом к данным на уровне сервера, базы данных и объектов базы данных. Он поддерживает аутентификацию и авторизацию пользователей, а также шифрование данных для обеспечения конфиденциальности.
6. Резервное копирование и восстановление данных: SQL Server предоставляет инструменты для создания резервных копий баз данных и их восстановления в случае сбоев или потери данных. Это обеспечивает защиту данных и обеспечивает возможность восстановления работы в случае чрезвычайных ситуаций.
7. Автоматизация и планирование задач: SQL Server предоставляет возможности для автоматизации и планирования регулярных задач, таких как резервное копирование, оптимизация баз данных и выполнение регулярных обслуживающих операций.
8. Аналитика и отчетность: SQL Server включает в себя инструменты для анализа данных и создания отчетов, такие как SQL Server Analysis Services (SSAS) для аналитической обработки данных и SQL Server Reporting Services (SSRS) для создания и распространения отчетов.
9. Интеграция с другими приложениями: SQL Server обеспечивает возможности интеграции с другими приложениями и технологиями через различные протоколы и интерфейсы, такие как ODBC, JDBC, OLE DB, ADO.NET и другие.

SQL Server Management Studio (SSMS) - это интегрированная среда разработки и администрирования для работы с Microsoft SQL Server (Рис. 5) и базами данных SQL Azure. Она предоставляет различные инструменты для управления базами данных, разработки запросов, настройки сервера, а также мониторинга и оптимизации производительности. Основные функции и возможности SQL Server Management Studio:

1. Подключение к серверам баз данных: SSMS позволяет подключаться к различным серверам баз данных SQL Server, включая локальные и удаленные серверы, а также базы данных SQL Azure в облаке Microsoft Azure.
2. Создание и управление базами данных: С помощью SSMS существует возможность создавать новые базы данных, изменять и удалять существующие базы данных, настраивать параметры баз данных, такие как размер файлов данных и журналов транзакций, а также управлять файлами баз данных.
3. Разработка запросов и скриптов: SSMS предоставляет мощные инструменты для разработки SQL-запросов, хранимых процедур, функций и других объектов баз данных. Включены редактор кода с подсветкой синтаксиса, автодополнением, функцией выделения и т.д.
4. Визуальное проектирование баз данных: SSMS позволяет создавать и изменять схемы баз данных с помощью визуальных диаграмм. Это позволяет легче визуализировать структуру баз данных и вносить изменения с помощью перетаскивания и редактирования объектов.
5. Администрирование сервера: SSMS предоставляет инструменты для администрирования SQL Server, включая управление службами и экземплярами сервера, настройку параметров сервера, резервное копирование и восстановление баз данных, мониторинг производительности и многое другое.
6. Мониторинг и оптимизация производительности: SSMS включает в себя инструменты для мониторинга производительности SQL Server, такие как просмотр активных сеансов, выполнение запросов и использование ресурсов сервера. Он также предоставляет средства для оптимизации производительности запросов и баз данных.
7. Интеграция с другими инструментами Microsoft: SSMS интегрируется с другими инструментами и сервисами Microsoft, такими как Visual Studio и Azure, что облегчает разработку и управление базами данных в экосистеме Microsoft.

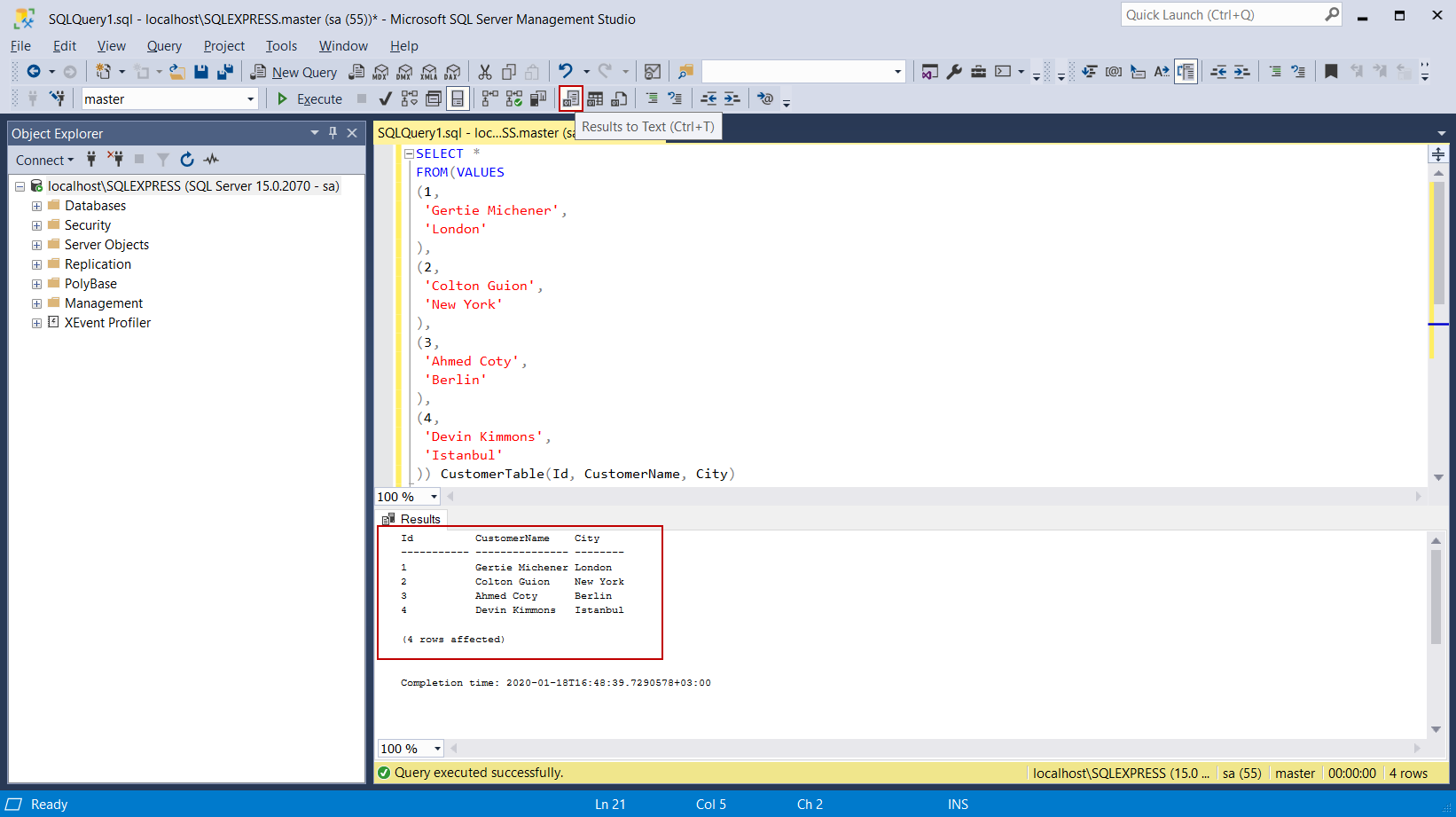


Рис. 5 Интерфейс SQL server management studio

# **Общая постановка задачи**

Целью данной курсовой работы является разработка системы электронной почты, которая будет эффективно обрабатывать запросы на обмен информацией, облегчать процессы управления и повышать уровень обслуживания сообщений. В рамках работы необходимо решить следующие задачи:

1. Определение функциональности системы: На основе анализа потребностей определить основные функции, которые должны быть реализованы в системе электронной почты. В программе должны быть реализованы следующие функции:
   1. **Управление пользовательской базой данных:** Возможность хранения информации о пользователях, их контактных данных, истории обслуживания и предыдущих ремонтах.
   2. **Учет и управление сообщениями:** Система должна позволять вести учет сообщений и контролировать их.
   3. **Добавление контактов**: Возможность добавления пользователем личного контакта для быстрой отправки сообщений.
   4. **Обновление базы данных**: Возможность обновлять базу данных.
   5. **Работа с базами данных SQL Server**: Программа должна иметь возможность подключения из базы данных SQL Server.
   6. **Регистрация и авторизация:** Система регистрировать новых и авторизировать уже существующих пользователей.
   7. **Отсутствие привязанности к конкретной СУБД:** Возможность изменять СУБД в ходе поддержки и обновления программы.
   8. **Изменение базы данных**: Возможность изменять значения

в базах данных

* 1. **Обмен файлами**: Возможность хранить и делиться файлами
  2. **Учет и управления электронными сообщениями**: Возможность хранить и изменять информацию о запчастях
  3. **Управление пользователями**: Возможность хранить и изменять информацию о пользователях
  4. **Создание новых полей в базах данных**: Программа должна иметь возможность создавать новые поля для работы с данными

1. **Разработка и реализация системы**: На основе функциональных требований разработать систему электронной почты и реализовать необходимые модули и функции. Обеспечить интеграцию с базами данных SQL Server.
2. Диаграмма базы данных: для более наглядного представления структуры системы электронной почты, разработана диаграмма базы данных (Рис. 6), которая показывает основные сущности и их взаимосвязи.

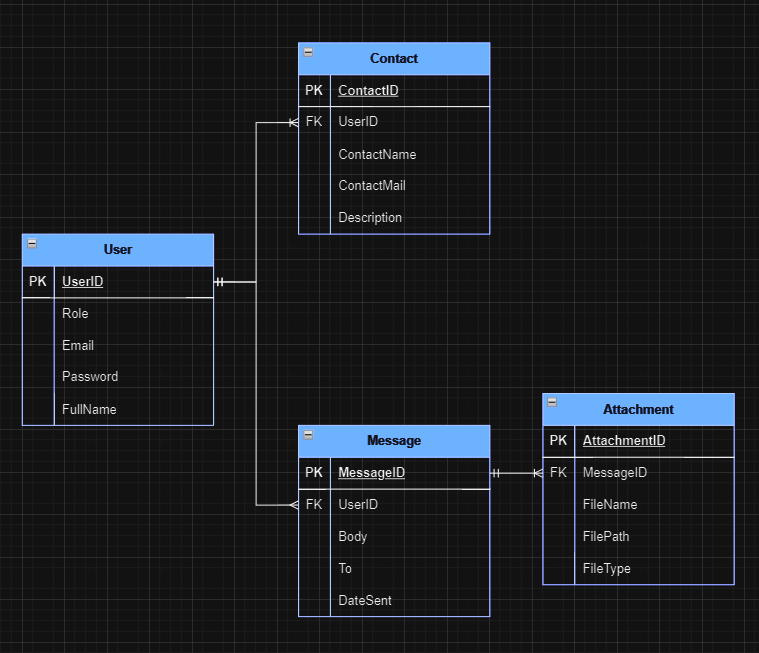


Рис. 6 Диаграмма базы данных

В представленной диаграмме базы данных присутствуют четыре таблицы: «User», «Contact», «Message» и «Attachment». Эти таблицы содержат информацию о пользователях, контактах, сообщениях и вложениях, а также устанавливают связи между ними с помощью первичных и внешних ключей.

Таблица «User»: содержит информацию о пользователях, включая уникальный идентификатор пользователя (UserId), его роль (Role), адрес электронной почты (Email), полное имя (FullName) и пароль (Password).

Таблица «Contact»: содержит информацию о контактах пользователя. У каждого контакта есть уникальный идентификатор (ContactId), а также связь с конкретным пользователем через внешний ключ (UserId). Дополнительные поля включают имя контакта (ContactName), адрес электронной почты контакта (UserEmailId) и описание (Description).

Таблица, «Message»: отражает информацию о сообщениях, отправленных пользователями. У каждого сообщения есть свой уникальный идентификатор (MessageId), а также связь с отправителем через внешний ключ (UserId). Поля сообщения включают адресата (To), текст сообщения (Body) и дату отправки (DateSent).

Таблица «Attachment»: хранится информация о вложениях к сообщениям. Каждое вложение имеет свой уникальный идентификатор (AttachmentId), а также связь с соответствующим сообщением через внешний ключ (MessageId). Дополнительные поля включают имя файла (FileName), тип файла (FileType) и путь к файлу (FilePath).

Связи между таблицами указывают на отношения между их записями. Например, связь от UserId в таблице User к UserId в таблицах Contact и Message показывает, что каждый контакт и каждое сообщение связаны с определенным пользователем. Связь от MessageId в таблице Message к MessageId в таблице Attachment показывает, что каждое вложение связано с определенным сообщением.

Таким образом, эта база данных структурирована чтобы поддерживать функциональность приложения, в котором пользователи могут отправлять сообщения с вложениями и сохранять контактную информацию.

1. Диаграмма прецедентов: представляет собой визуальное изображение взаимодействий между различными участниками системы и её функциональными возможностями (Рис. 7). В данном случае присутствуют два основных актера: «Пользователь» и «Администратор».

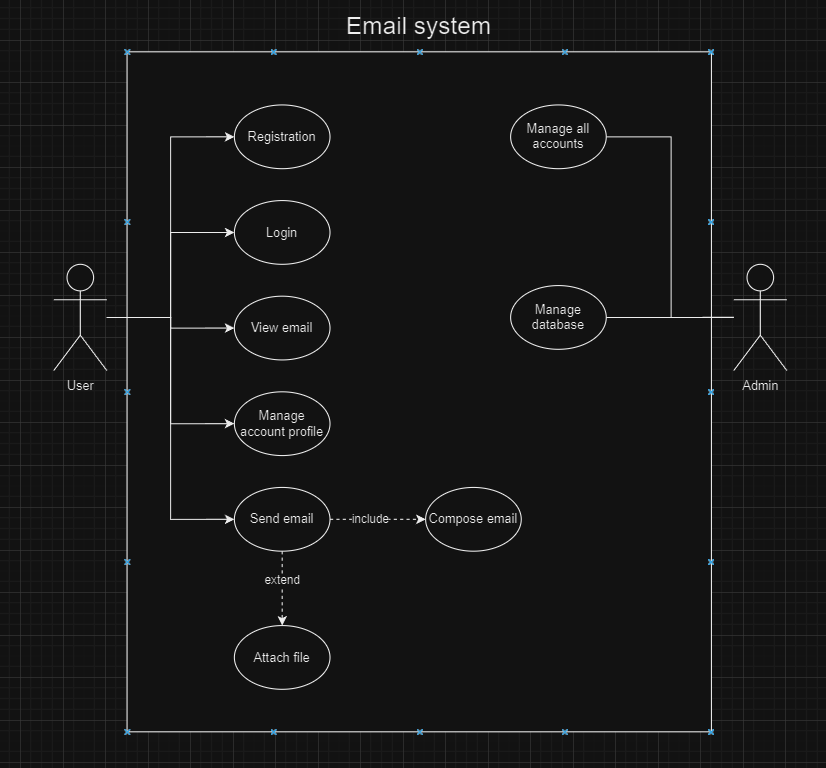


Рис. 7 Диаграмма прецендентов

Пользователь, как основной пользователь системы, имеет ряд основных действий, которые он может выполнять в системе:

Регистрация: Этот прецедент представляет собой процесс регистрации новой учётной записи в системе. Пользователь заполняет необходимые поля, такие как имя, адрес электронной почты и пароль, и отправляет данные для создания учётной записи.

Вход в систему: после регистрации или при последующих посещениях системы пользователь может войти в свою учётную запись. Этот прецедент включает в себя процесс аутентификации пользователя, где он предоставляет свои учётные данные (например, адрес электронной почты и пароль) для входа в систему.

Просмотр почты: Пользователь может просматривать свою электронную почту, просматривать полученные сообщения и отвечать на них.

Управление профилем: Этот прецедент описывает возможность пользователя изменять свои настройки профиля, такие как смена пароля, обновление контактной информации и другие персонализированные настройки.

Отправка почты: Один из основных функциональных элементов системы - отправка электронных сообщений. Пользователь может создавать новые сообщения, указывать адресата, тему и содержание сообщения, а также прикреплять к ним файлы.

1. Диаграмма классов для доменного слоя (Рис. 8) в проектировании программного обеспечения является мощным инструментом для визуализации структуры и отношений между классами, которые составляют основу программной системы. Она помогает разработчикам лучше понять, как компоненты системы взаимодействуют друг с другом на уровне домена, что является ключевым аспектом при разработке сложных приложений.

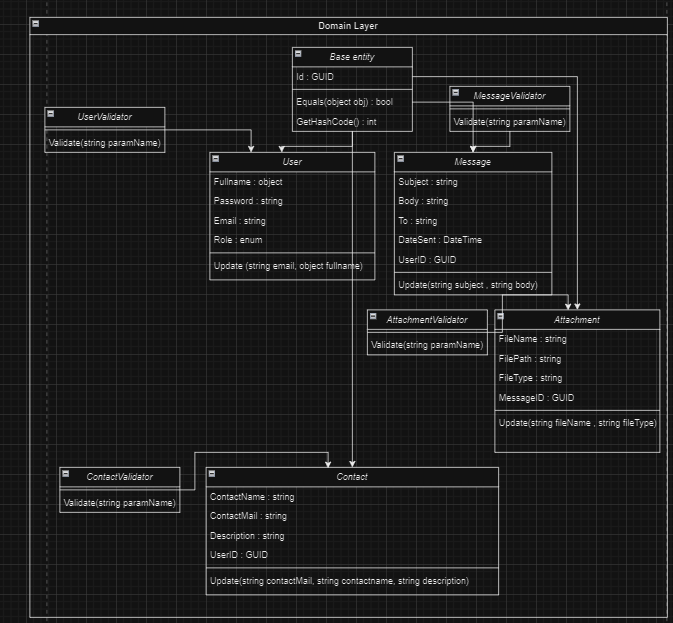


Рис. 8 Диаграмма классов для доменного слоя

В данной диаграмме основные классы, такие как BaseEntity, UserValidator, User, MessageValidator, Message и AttachmentValidator, представляют различные абстракции или концепции в рамках доменной модели. Каждый из этих классов обычно имеет свои уникальные атрибуты и методы, которые отражают их поведение и характеристики. Например, класс User может содержать атрибуты, такие как полное имя пользователя (FullName), пароль (Password), адрес электронной почты (Email), роль (Role), а также методы для обновления этих атрибутов.

Кроме того, на диаграмме отображаются различные типы отношений между классами, такие как наследование и ассоциации. Наследование позволяет классам наследовать атрибуты и методы от других классов, что способствует повторному использованию кода и упрощает его поддержку. Ассоциации, с другой стороны, указывают на связи или взаимодействия между классами, например, класс User может быть ассоциирован с классом Message через отношение отправителя-получателя.

Использование диаграммы классов доменного слоя позволяет не только лучше понять структуру программной системы, но и обеспечивает эффективное средство коммуникации между членами команды разработки, архитекторами и другими заинтересованными сторонами. Она помогает улучшить общее понимание архитектуры системы и способствует ее дальнейшему развитию и сопровождению.

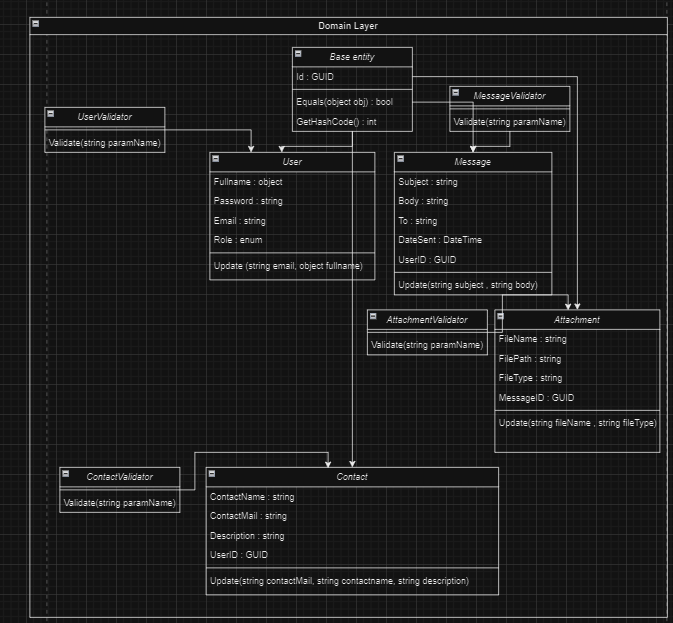
1. Диаграмма классов для слоя инфраструктуры (Рис. 9): на рисунке представлена диаграмма классов с несколькими классами и их отношениями в контексте разработки программного обеспечения. Диаграмма структурирована на три колонки, каждая из которых представляет разные компоненты, помеченные как «Infrastructure», «Context» и «Repositories».
2. 

Рис. 9 Диаграмма классов для слоя инфраструктуры

В колонке «Infrastructure» присутствует четыре класса: UserConfigTypeConfiguration, ContactConfigTypeConfiguration, AttachmentConfigTypeConfiguration и DatabaseConfiguration. Каждый класс имеет свойства, перечисленные в нем, такие как UserConfigTypeId в классе UserConfigTypeConfiguration.

Колонка «Context» содержит один класс под названием DatabaseContext со свойствами, такими как Users: DbSet, Messages: DbSet, и методами, такими как OnModelCreating(ModelBuilder modelBuilder).

Колонка «Repositories» содержит несколько классов репозиториев.

Каждый класс репозитория имеет определенные в нем методы. Например, IMessageRepository включает методы:

GetAsync(Guid id, CancellationToken token): object response

UpdateAsync(Message message, CancellationToken token): object response

Присутствуют линии ассоциаций, соединяющие эти классы, указывающие на их отношения.

Эта диаграмма актуальна для понимания, как различные компоненты программной системы взаимодействуют друг с другом через определенные интерфейсы и типы данных. Она предоставляет визуальное представление структуры системы, которое может быть особенно полезно для разработчиков при проектировании или понимании архитектуры приложения.

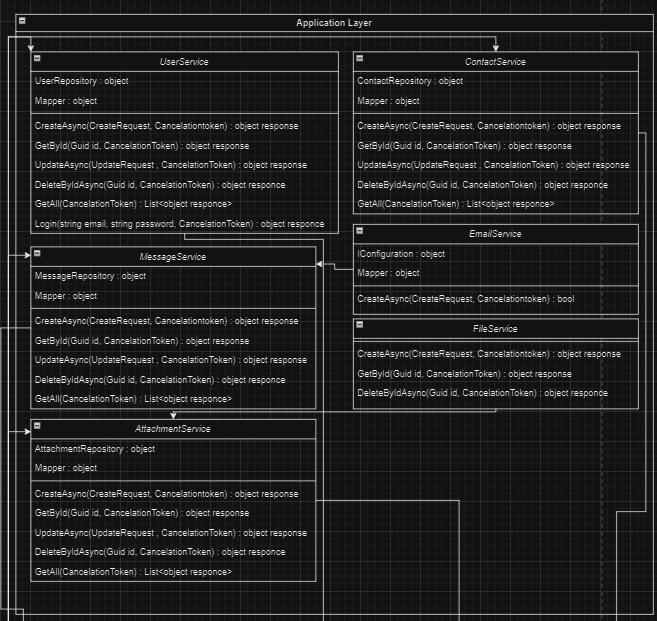
1. Диаграмма классов для приложения (Рис. 10): представлена диаграмма которая отображает архитектуру слоя приложения, в частности, детализируя различные сервисы и их взаимодействия. Она включает в себя множество блоков, помеченных разными названиями сервисов, такими как «UserService», «ContactService», «MessageService» и другие. Каждый блок содержит список методов или функций с параметрами и возвращаемыми типами, указывающими на операции, которые может выполнять этот сервис. Диаграмма актуальна для понимания структуры и функциональности бэкенда приложения, показывая, как различные компоненты разработаны для совместной работы.
2. 

Рис. 10 Диаграмма классов для доменного слоя

1. Реализация чистой архитектуры (Рис. 11): реализовать чистую архитектуру в приложении с учетом ключевых требований:
   1. Разделение ответственностей: Каждый компонент должен иметь четко определенную ответственность и заниматься только одним аспектом системы. Например, бизнес-логика должна быть отделена от взаимодействия с базой данных или пользовательским интерфейсом.
   2. Зависимости должны быть обратимыми: Компоненты верхнего уровня не должны зависеть от компонентов нижнего уровня. Зависимости должны быть направлены от абстракций к деталям, что позволяет легко заменять или модифицировать конкретные реализации.
   3. Однозначность интерфейсов: Интерфейсы между компонентами должны быть явными и четко определенными. Это помогает понять, какие методы и данные доступны для использования, и избежать неявных зависимостей.
   4. Тестируемость : Код должен быть легко тестируемым без необходимости запуска всей системы. Каждый компонент должен иметь набор тестов, которые могут быть запущены автоматически.
   5. Легкость замены компонентов : Компоненты должны быть спроектированы таким образом, чтобы их можно было легко заменить новой реализацией без изменения остальной части системы. Например, замена базы данных должна быть минимально инвазивной операцией.
   6. Легкость понимания и поддержки: Структура приложения должна быть понятной для разработчиков, которые присоединяются к проекту. Код должен быть хорошо документирован, и его структура должна соответствовать бизнес-логике системы.
   7. Соблюдение принципов SOLID: Каждый компонент должен следовать принципам SOLID, таким как Single Responsibility Principle и Open/Closed Principle.

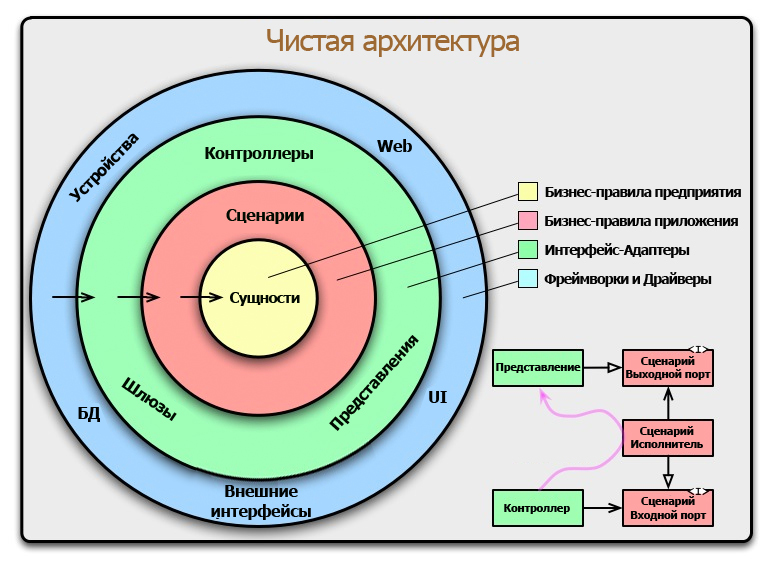


Рис. 11 Структура чистой архитектуры

**Итогом работы** должна быть разработанная, реализованная и протестированная система электронной почты, способная эффективно отправлять сообщения и поддерживать конфиденциальность пользователей.

# **ЗАКЛЮЧЕНИЕ**

В результате выполнения курсовой работы разработана информационная система для автомастерской, предназначенная для оптимизации в работе основных процессов.

Программа разработана на языке программирования C#, согласно требованиям, к системам электронной почты. В данной программе обусловлены все необходимые разделы и области необходимые для эффективного функционирования почтового сервиса.

Разработанная система проста в использовании, обеспечивает интуитивно понятный интерфейс. Она включает в себя все необходимые модули и функции, такие как учет отправка сообщений, добавление контактов, обмен файлами и управление данными пользователей.

Внедрение разработанной программы позволит значительно упростить работу с электронными сообщениям. Это приведет к повышению эффективности работы почтового сервиса, сокращению количества неполадок и увеличению удовлетворенности пользователей.

СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

Научная и учебно-методическая литература

1) Борисенко, В.В. (2016). Основы программирования. Интернет-Университет Информационных Технологий.

2) Туральчук, К.А. (2019). Параллельное программирование с помощью языка C#. Интернет-Университет Информационных Технологий.

3) Золотарёв, О.В. (2013). Технология внедрения корпоративных информационных систем: методические указания к лабораторным работам. Российский новый университет.

Интернет-ресурсы

4) Metanit С#: (<https://metanit.com/sharp>)

5)Соммервиль, И. (2018). Объектно-ориентированное мышление. Pearson Education.

6)Шилдт, Г. (2018). C# 8.0. Полное руководство. Вильямс.

7)Макконнелл, С. (2015). Совершенный код. Практическое руководство по разработке программного обеспечения. Вильямс.

8)Мартин, Р. (2019). Чистый код: создание, анализ и рефакторинг. Питер.

9)Freeman, A., Robson, D. (2014). Head First Design Patterns. O'Reilly Media.

ПРИЛОЖЕНИЕ

|  |  |
| --- | --- |
| using LeMail.Domain.Validations;  using LeMail.Domain.Validations.Primitives;  using LeMail.Domain.Validations.Validators.Entities;  using LeMail.Domain.ValueObjects;  using System.Security.Cryptography;  namespace LeMail.Domain.Entities;  /// <summary>  /// User Entity  /// </summary>  public class User : BaseEntity  {  /// <summary>  /// Role enum  /// </summary>  public Role Role { get; set; }  /// <summary>  /// Email adress  /// </summary>  public string Email { get; set; }  /// <summary>  /// User password  /// </summary>  public string Password { get; set; }  /// <summary>  /// User fullname  /// </summary>  public FullName FullName { get; set; }    /// <summary>  /// Collection of messages  /// </summary>  public ICollection<Message> Messages { get; set; }  /// <summary>  /// Collection of Contacts  /// </summary>  public ICollection<Contact> Contacts { get; set; }  public User()  {  var validator = new UserValidator(nameof(User));  validator.ValidateWithExceptions(this);    }    /// <summary>  /// Update User entity method  /// </summary>  /// <param name="email"></param>  /// <param name="fullName"></param>  public void Update(string email, FullName fullName)  {  FullName.Update(fullName);  Email = email;  var validator = new UserValidator(nameof(User));  validator.ValidateWithExceptions(this);  }  }  using LeMail.Domain.Validations;  using LeMail.Domain.Validations.Validators.Entities;  namespace LeMail.Domain.Entities;  /// <summary>  /// Message Entity  /// </summary>  public class Message : BaseEntity  {  /// <summary>  /// Navigation property  /// </summary>  public User User { get; init; }  /// <summary>  /// Foreign key  /// </summary>  public Guid UserId { get; init; }  /// <summary>  /// Email subject  /// </summary>  public string Subject { get; set; }  /// <summary>  /// Email body text  /// </summary>  public string Body { get; set; }  /// <summary>  /// Address email  /// </summary>  public string To { get; set; }  /// <summary>  /// Send date  /// </summary>  public DateTime DateSent { get; set; } = DateTime.Now;    public ICollection<Attachment> Attachments { get; set; }  // Добавьте конструктор без параметров  public Message()  {  var validator = new MessageValidator(nameof(Message));  validator.ValidateWithExceptions(this);  }  /// <summary>  /// Update Message entity method  /// </summary>  /// <param name="subject"></param>  /// <param name="body"></param>  public void Update(string subject, string body)  {  Subject = subject;  Body = body;  DateSent = DateTime.Now;  var validator = new MessageValidator(nameof(Message));  validator.ValidateWithExceptions(this);  }  }  using LeMail.Domain.Validations;  using LeMail.Domain.Validations.Validators.Entities;  namespace LeMail.Domain.Entities;  /// <summary>  /// Contact entity  /// </summary>  public class Contact : BaseEntity  {  /// <summary>  /// name of contact  /// </summary>  public string ContactName { get; set; }  /// <summary>  /// contact email address  /// </summary>  public string ContactMail { get; set; }  /// <summary>  /// description of contact  /// </summary>  public string Description { get; set; }  /// <summary>  /// Navigation property  /// </summary>  public User User { get; set; }  /// <summary>  /// Foreign key  /// </summary>  public Guid UserId { get; set; }  public Contact()  {  var validator = new ContactValidator(nameof(Contact));  validator.ValidateWithExceptions(this);  }  /// <summary>  /// Update Contact entity method  /// </summary>  /// <param name="contactName"></param>  /// <param name="contactMail"></param>  /// <param name="description"></param>  public void Update(string contactName, string contactMail, string description)  {  ContactName = contactName;  ContactMail = contactMail;  Description = description;  var validator = new ContactValidator(nameof(Contact));  validator.ValidateWithExceptions(this);  }  }  namespace LeMail.Domain.Entities;  /// <summary>  /// Base entity class  /// </summary>  public abstract class BaseEntity  {  /// <summary>  /// Id of entity  /// </summary>  public Guid Id { get; set; }  /// <summary>  /// Equals override  /// </summary>  /// <param name="obj"></param>  /// <returns></returns>  public override bool Equals(object? obj)  {  if (obj == null)  return false;  else if (obj is not BaseEntity baseEntity)  return false;  else if (baseEntity.Id != Id)  return false;  else if (this.GetHashCode() != baseEntity.GetHashCode())  return false;  return true;  }  /// <summary>  /// GetHasCodeOverride  /// </summary>  /// <returns></returns>  public override int GetHashCode()  {  return Id.GetHashCode();  }  }  using LeMail.Domain.Validations;  using LeMail.Domain.Validations.Validators.Entities;  namespace LeMail.Domain.Entities;  public class Attachment : BaseEntity  {  /// <summary>  /// File name  /// </summary>  public string FileName { get; set; }  /// <summary>  /// File path  /// </summary>  public string FilePath { get; set; }    /// <summary>  /// Extensions of File  /// </summary>  public string FileType { get; set; }    public Guid MessageId { get; set; }    public Message Message { get; set; }  public Attachment()  {  var validator = new AttachmentValidator(nameof(Attachment));  validator.ValidateWithExceptions(this);  }  public void Update(string fileName, string filePath, string fileType)  {  FileName = fileName;  FilePath = filePath;  FileType = fileType;  var validator = new AttachmentValidator(nameof(Attachment));  validator.ValidateWithExceptions(this);  }  } | using AutoMapper;  using LeMail.Application.Interfaces.Repository;  using LeMail.Domain.Entities;  using LeMail.Application.Dto\_s.User.Requests;  using LeMail.Application.Dto\_s.User.Responses;  using LeMail.Application.Interfaces.Services;  using LeMail.Domain.Validations;  using LeMail.Domain.Validations.Validators.Entities;  namespace LeMail.Application.Services  {  public class UserService : IUserService  {  private readonly IUserRepository \_userRepository;  private readonly IMapper \_mapper;  public UserService(IUserRepository userRepository, IMapper mapper)  {  \_userRepository = userRepository;  \_mapper = mapper;  }  public async Task<CreateUserResponse> CreateUserAsync(CreateUserRequest request, CancellationToken cancellationToken)  {  var userEntity = \_mapper.Map<User>(request);    var validator = new UserValidator(nameof(User));  validator.ValidateWithExceptions(userEntity);    var createdUser = await \_userRepository.CreateAsync(userEntity, cancellationToken);  return \_mapper.Map<CreateUserResponse>(createdUser);  }  public async Task<GetUserResponse> GetUserByIdAsync(Guid id, CancellationToken cancellationToken)  {  var userEntity = await \_userRepository.GetByIdAsync(id, cancellationToken);  return \_mapper.Map<GetUserResponse>(userEntity);  }  public async Task<UpdateUserResponse> UpdateUserAsync(UpdateUserRequest request, CancellationToken cancellationToken)  {  var user = await \_userRepository.GetByIdAsync(request.Id, cancellationToken);  \_mapper.Map(request, user);    var updatedUser = await \_userRepository.UpdateAsync(user, cancellationToken);    return \_mapper.Map<UpdateUserResponse>(updatedUser);  }  public async Task<bool> DeleteUserByIdAsync(Guid id, CancellationToken cancellationToken)  {  return await \_userRepository.DeleteByIdAsync(id, cancellationToken);  }  public async Task<List<GetUserResponse>> GetAllUsersAsync(CancellationToken cancellationToken)  {  var userEntities = await \_userRepository.GetAllListAsync(cancellationToken);  return \_mapper.Map<List<GetUserResponse>>(userEntities);  }  public async Task<GetUserResponse> LoginAsync(string email, string password, CancellationToken cancellationToken)  {  var user = await \_userRepository.LoginAsync(email, password, cancellationToken);  return \_mapper.Map<GetUserResponse>(user);  }  }  }  using AutoMapper;  using LeMail.Application.Dto\_s.Message.Requests;  using LeMail.Application.Dto\_s.Message.Responses;  using LeMail.Application.Interfaces.Repository;  using LeMail.Application.Interfaces.Services;  using LeMail.Domain.Entities;  using LeMail.Domain.Validations;  using LeMail.Domain.Validations.Validators.Entities;  namespace LeMail.Application.Services;  public class MessageService : IMessageService  {  private readonly IMessageRepository \_messageRepository;  private readonly IEmailService \_emailService;  private readonly IMapper \_mapper;  public MessageService(IMessageRepository messageRepository, IMapper mapper, IEmailService emailService)  {  \_messageRepository = messageRepository;  \_mapper = mapper;  \_emailService = emailService;  }    public async Task<CreateMessageResponse> CreateMessageAsync(CreateMessageRequest request, CancellationToken cancellationToken)  {  var message = \_mapper.Map<Message>(request);    var validator = new MessageValidator(nameof(Message));  validator.ValidateWithExceptions(message);    var createdMessage = await \_messageRepository.CreateAsync(message,cancellationToken);  var response = \_mapper.Map<CreateMessageResponse>(createdMessage);    await \_emailService.SendEmailAsync(request, cancellationToken);    return response;  }  public async Task<GetMessageResponse> GetMessageByIdAsync(Guid id, CancellationToken cancellationToken)  {  var message = await \_messageRepository.GetByIdAsync(id,cancellationToken);  var response = \_mapper.Map<GetMessageResponse>(message);    return response;  }  public async Task<UpdateMessageResponse> UpdateMessageAsync(UpdateMessageRequest request, CancellationToken cancellationToken)  {  var message = await \_messageRepository.GetByIdAsync(request.Id, cancellationToken);    \_mapper.Map(request, message);    var updatedMessage = await \_messageRepository.UpdateAsync(message, cancellationToken);  var response = \_mapper.Map<UpdateMessageResponse>(updatedMessage);    return response;  }  public async Task<bool> DeleteMessageByIdAsync(Guid id, CancellationToken cancellationToken)  {  return await \_messageRepository.DeleteByIdAsync(id, cancellationToken);  }  public async Task<List<GetMessageResponse>> GetAllMessagesAsync(CancellationToken cancellationToken)  {  var messageEntities = await \_messageRepository.GetAllListAsync(cancellationToken);  var response = \_mapper.Map<List<GetMessageResponse>>(messageEntities);    return response;  }  public async Task<List<GetMessageResponse>> GetAllListByUserIdAsync(Guid id, CancellationToken cancellationToken)  {  var messageEntities = await \_messageRepository.GetAllListByUserIdAsync(id,cancellationToken);  var response = \_mapper.Map<List<GetMessageResponse>>(messageEntities);    return response;  }  }  using LeMail.Application.Interfaces.Services;  using Microsoft.AspNetCore.Http;  using Microsoft.AspNetCore.Http.Internal;  namespace LeMail.Application.Services;  public class FileService : IFileService  {  public async Task<bool> CreateFileAsync(IFormFile file, string path)  {  await using var stream = new FileStream(path, FileMode.Create);    var directory = Path.GetDirectoryName(path);  if (!Directory.Exists(directory))  {  Directory.CreateDirectory(directory);  }  await file.CopyToAsync(stream);  return true;  }  public async Task<bool> DeleteFileAsync(string path)  {  if (!File.Exists(path))  {  return false;  }  File.Delete(path);  return true;  }  public async Task<IFormFile> GetFileAsync(string path)  {  if (!File.Exists(path))  {  return null;  }  var fileStream = File.OpenRead(path);    var file = new FormFile(fileStream, 0, fileStream.Length, null, Path.GetFileName(path))  {  Headers = new HeaderDictionary(),  ContentType = "application/octet-stream"  };    return file;  }  } |