

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(ФГБОУ ВО «КубГУ»)

Факультет компьютерных технологий и прикладной математики
Кафедра анализа данных и искусственного интеллекта

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

**РАЗРАБОТКА КОРПОРАТИВНОГО МЕССЕНДЖЕРА С
ШИФРОВАНИЕ ДАННЫХ**

Работу выполнил _____ Б.А. Михаенко
(подпись)

Направление подготовки 09.03.03 Прикладная информатика

Направленность (профиль) Прикладная информатика в экономике

Научный руководитель
канд. техн. наук, доц. _____ А.В. Харченко
(подпись)

Нормоконтролер
канд. пед. наук, доц. _____ А.В. Харченко
(подпись)

Краснодар
2024

РЕФЕРАТ

Выпускная квалификационная работа 70 с., 31 рис., 1 таблица, 9 источников.

РАЗРАБОТКА КОРПОРАТИВНОГО МЕССЕНДЖЕРА С ШИФРОВАНИЕМ ДАННЫХ

Объектом разработки данной работы является корпоративный мессенджер с шифрованием данных.

Целью работы является изучение технологий разработки корпоративных сетей общения, а также создание своего собственного продукта.

В данной работе были рассмотрены необходимые технологии для создания корпоративного мессенджера с шифрованием данных, а также был проведен сравнительный анализ существующих решений в сфере сетевого общения.

В результате была разработан собственный корпоративный мессенджер. Основные функции данного продукта включают в себя регистрации и авторизации пользователей, личный профиль пользователя, поиск зарегистрированных пользователей, создание диалогового чата, создание команд, настройка команд и личного профиля пользователей.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ.....	4
1 Сравнительный анализ корпоративных мессенджеров	5
2 Технологии создания корпоративных мессенджеров.....	9
2.1 Основные архитектурные решения.....	9
2.2 Основные языки программирования и их фреймворки	18
2.3 Базы данных	24
2.4 Системы кэширования.....	26
2.5 Протоколы передачи данных	28
2.6 Системы контейнеризации (Docker).....	31
2.7 Методы шифрования данных.....	32
3 Реализация собственного корпоративного мессенджера	35
3.1 Архитектура.....	35
3.2 Шифрование	40
3.3 База данных MongoDB.....	44
3.4 Роутинг и реализация API.....	47
3.5 Контейнеризация и Docker.....	49
3.6 Взаимодействие клиента с приложением.....	52
ЗАКЛЮЧЕНИЕ.....	69
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ.....	70

ВВЕДЕНИЕ

В современном мире информационная безопасность является одним из важнейших аспектов деятельности любой компании. Конфиденциальность данных, особенно в сфере бизнеса, имеет огромное значение для обеспечения конкурентоспособности и устойчивого развития. В эпоху цифровых технологий обмен конфиденциальной информацией между сотрудниками компании происходит преимущественно через различные мессенджеры. Однако многие из них не обеспечивают должного уровня безопасности, что делает их использование в корпоративной среде крайне рискованным.

Многие мессенджеры, являющиеся бытовыми продуктами в повседневной жизни человека, заполнены необходимой для него информацией: чаты с другими людьми, не относящимися к работе, каналы, сообщества, ленты – все это приводит к отсутствию концентрации внимания и, как следствие, к не продуктивной работе сотрудника.

Корпоративный мессенджер (или Корпоративная социальная сеть) – это программа или платформа для мгновенного обмена сообщениями и материалами между сотрудниками компании. Данный тип мессенджеров позволяет избежать проблемы отвлекающих факторов, влияющих на продуктивность сотрудников, а шифрование данных позволяет сохранить передающиеся по сети данные в безопасности.

Предложенная работа посвящена сравнительному анализу корпоративных мессенджеров, изучению технологий и разработке собственного мессенджера.

Первая глава данной работы содержит описание и сравнение, а также выявление недостатков существующих корпоративных мессенджеров.

Вторая глава посвящена описанию технологий, используемых для реализации возможностей общения, а также шифрования данных в корпоративных мессенджерах.

Третья глава содержит описание реализации собственного корпоративного мессенджера, а также объяснения выбора конкретных технологий.

Четвертая глава показывает взаимодействие пользователя с разработанным продуктом.

1 Сравнительный анализ корпоративных мессенджеров

В настоящий момент существует множество различных корпоративных мессенджеров, каждый из реализованных продуктов пытается быть уникальным и внедрить новые функции. В данной главе мы рассмотрим основные, конкурирующие между собой, корпоративные мессенджеры и выявим их преимущества и недостатки. Сравнить данные продукты будем по следующим категориям:

- Безопасность,
- Стоимость,
- Масштабируемость,
- Независимость.

Это основные категории, которые влияют на выбор подходящего мессенджера для компании. Безопасность отвечает за то, каким образом шифруются и передаются данные; Стоимость отвечает за количество затрачиваемых денежных ресурсов на использование такого продукта; Масштабируемость отвечает за возможность настройки продукта под нужды компании; Независимость, в свою очередь, отвечает за независимую от политической ситуации в мире работу данного продукта.

Сравнение корпоративных мессенджеров, используемых в компаниях на данный момент представлено в таблице 1.

Таблица 1 – Сравнение основных корпоративных мессенджеров

Мессенджер	Безопасность	Стоимость	Масштабируемость	Независимость
Rocket.Chat	Открытый исходный код, доступно сквозное шифрование	Бесплатный для саморазвертывания, платные планы для облачного хостинга	Является OpenSource проектом	Является независимым продуктом
Slack	Закрытый исходный код, шифрование на серверах компании	Платные планы, бесплатная ограниченная версия	Закрытый исходный код, нет возможности программной настройки	Есть вероятность блокировки со стороны разработчиков
Telegram	Открытый исходный код, доступно сквозное шифрование	Бесплатный	Является OpenSource проектом, есть возможность реализации внутренних элементов (чат-боты)	Является независимым продуктом
VK Teams	Закрытый исходный код, шифрование на серверах компании	Платные планы, нужна регистрация компании	Закрытый исходный код, нет возможности программной настройки	Является отечественным продуктом
Discord	Закрытый исходный код, шифрование на серверах компании	Бесплатный, есть платная подписка с расширенными возможностями	Закрытый исходный код, нет возможности программной настройки	Является независимым продуктом
Microsoft Teams	Закрытый исходный код, шифрование на серверах компании	Необходима покупка лицензии	Закрытый исходный код, нет возможности программной настройки	Есть вероятность блокировки со стороны разработчиков

Исходя из представленной таблицы можно сделать вывод о популярности тех или иных мессенджеров в компаниях.

Slack имеет крупное сообщество пользователей, что позволяет ему занимать лидирующую позицию в выборе корпоративной связи. Он имеет широкий спектр функций. Однако имеет также и ряд недостатков, основным из которых является платный план, что может повлиять на выбор данного продукта. Платные планы могут быть дорогостоящими для больших команд. Большой спектр функции, также может быть не всегда полезен, так как из-за их количества растет и время обучения персонала на быстрое и качественное использование данного продукта.

Большое предпочтение среди компаний отдается Rocket.Chat. Так как данный продукт является OpenSource-проектом (Проект с открытым исходным кодом), что дает возможность программной настройки данного мессенджера под нужды компании, бесплатность, а также возможность использования данного продукта на собственных мощностях компании. Однако имеет менее широкий спектр функций в отличии от Slack, что может повлиять на выбор данного продукта.

Также большую популярность обретает Telegram в жизни многих компаний. Имея возможность расширения и автоматизации задач, с помощью встроенных чат-ботов, а также большого количества библиотек помогающих непосредственно в разработке вышеупомянутых чат-ботов, данный мессенджер не уступает в функциональности своим конкурентам. Однако в связи с тем, что данный продукт по совместительству является социальной сетью, в нем появляется проблема уменьшения концентрации внимания сотрудника из-за существования различных отвлекающих факторов, таких как: беседы с другими людьми, не относящимся к рабочему процессу, каналы, публикации, трансляции. В связи с этим Telegram все же может быть отвергнут компаниями при выборе продукта осуществления внутренней связи сотрудников.

VK Teams имеет меньшую аудиторию, чем предшествующие ему мессенджеры, однако является отечественным продуктом, что делает его независимым от политической ситуации в мире. Также он имеет широкий ряд встроенных функций и непосредственную связь с самой социальной сетью VK. Однако работа с данным продуктом осуществляется через платный план, в связи с чем он может не рассматриваться компаниями в угоду бесплатным проектам с открытым исходным кодом.

Microsoft Teams является облачной платформой для корпоративной коммуникации. Входит в состав пакета Microsoft 365, в связи с чем имеет тесную интеграцию с другими продуктами компании, что позволяет работать не только с коллегами, но редактировать и оформлять документы в одной экосистеме. Однако Microsoft Teams имеет ряд существенных недостатков. Во-первых, это большое время обучения сотрудников из-за большого количества функций, ограничение бесплатной версии и дорогостоящий тарифный план. Также данный продукт не является независимым и может быть заблокирован по решению разработчиков, что является достаточно большим недостатком в ряду уже перечисленных. В связи с чем, данный продукт теряет популярность в многих компаниях.

Discord – платформа, изначально созданная для коммуникации между игроками. Однако ее функционал, включающий в себя возможность создания серверов для различных тематик, внутренней системы ролей, голосового, видео и текстового чата позволяет использовать эту систему, как нечто приближенное к корпоративному мессенджеру. Однако здесь также возникает проблема концентрации сотрудника, а также более глубокая настройка сервера включает в себя платную подписку, что также может повлиять на выбор данного продукта, как средство связи между сотрудниками.

Исходя из описанного выше мы выявили основные недостатки, а также преимущества существующих корпоративных мессенджеров, рассмотрели и сравнили их между собой.

2 Технологии создания корпоративных мессенджеров

В данной главе будут рассмотрены основные технологии, архитектурные решения, протоколы позволяющие создать расширяемый и удобный сервер корпоративного мессенджера. А также рассмотрим основные методы, протоколы и решения в сфере шифрования данных и их безопасной передачи.

2.1 Основные архитектурные решения

Правильный выбор архитектурного решения, очевидно, зависит от поставленной задачи. В зависимости от сложности вашего продукта зависит выбор нужной архитектуры. Однако каждое из далее представленных решений имеет свои преимущества и недостатки и позволяет реализовывать продукты разной сложности и масштабируемости.

На данный момент самыми популярными архитектурными решениями являются:

1. Монолитная архитектура;
2. Микросервисная архитектура.

Рассмотрим каждое из этих решений поподробнее.

1. Монолитная архитектура

Монолитная архитектура представляет собой решение или подход к разработке программного продукта, при котором кодовая база этого продукта находится в одном исполняемом модуле или приложении, чаще всего написанном на одном языке программирования. Большое количество существующих программных продуктов использует именно этот подход.

Данный подход имеет следующие преимущества:

1. Простота развертывания;
2. Меньшее количество точек отказа;
3. Фокусирование на бизнес-логике приложения.

- Простота развертывания

Благодаря тому что все приложение имеет одну кодовую базу и представляет собой единую сущность, появляется то самое преимущество в простоте его развертывания. Это означает что данный проект может запускаться без надобности настройки дополнительных зависимых сервисов для его работы. Чаще всего все зависимости обычно интегрируются внутрь данного приложения, это также упрощает работу с ними. Также для монолитного приложения используется единый набор конфигурационных файлов для его развертывания, что повышает удобность настройки такого приложения.

Запуск такого монолитного приложения может требовать меньшее количество инфраструктурных ресурсов, чем, например, развертывание нескольких отдельных сервисов.

- Меньшее количество точек отказа

Поскольку все компоненты находятся в одном приложении взаимодействия между этими компонентами происходит напрямую с помощью вызова тех или иных функций, импорта модулей, которые также являются частью реализации и находятся в одной кодовой базе, что и остальные компоненты. Соответственно это упрощает обмен данными между теми или иными частями приложения и уменьшает риск отказа из-за проблем с внешними или отдельно развернутыми сервисами, являющимися частью данного приложения.

Также за счет существования единой точки входа в приложение, существенно упрощается обработка исключений, так как проблема может быть локализована в одном месте, что позволит быстрее ее решить и обнаружить.

- Фокусирование на бизнес-логике приложения

В силу того, что разработчикам, при использовании монолитной архитектуры, не нужно разделять логику между сервисами, они могут сконцентрировать внимание на реализации бизнес-логики приложения. На первых этапах разработки монолитная архитектура позволяет гораздо быстрее запустить и протестировать нужный продукт, так как не нужно тратить время на разделение логики между сервисами, а также реализации общения между ними, что позволяет подготовить продукт к релизу гораздо быстрее.

Однако у монолитной архитектуры есть и свои недостатки:

1. Сложность масштабирования при росте приложения;
2. Медленные итерации;
3. Одно место отказа.

Рассмотрим каждый из них по порядку.

- Сложность масштабирования при росте приложения

В силу того, что разработчикам, при использовании монолитной архитектуры, не нужно разделять логику между сервисами, они могут сконцентрировать внимание на реализации бизнес-логики приложения. На первых этапах разработки монолитная архитектура позволяет гораздо быстрее запустить и протестировать нужный продукт, так как не нужно тратить время на разделение логики между сервисами, а также реализации общения между ними, что позволяет подготовить продукт к релизу гораздо быстрее.

- Медленные итерации

Поскольку все части приложения находятся внутри одного запускаемого модуля при внесении малейших изменений придется пересобирать и перезапускать все приложение целиком вне зависимости от того в какой части

приложения было изменение. Это может замедлить процесс разработки данного продукта.

- Одно место отказа

Одно из преимуществ данного подхода, к сожалению, также является его недостатком. Из-за того, что приложение, построенное на монолитной архитектуре, имеет одну точку входа и все его части не являются не зависимыми сервисами, возникновение ошибки в одной из частей приложения приведет к его частичной или полной остановке на момент отладки и исправления.

2. Микросервисная архитектура

Микросервисная архитектура представляет собой решение или подход к разработке программного продукта, при котором кодовая база этого продукта разбивается на автономные модули функционирующие отдельно от других сервисов, каждый из которых может иметь свою собственную базу данных, а также использовать совсем другой язык программирования для реализации этого сервиса.

Однако и у этого подхода, очевидно, есть свои преимущества и недостатки.

Рассмотрим преимущества данного подхода по порядку:

1. Гибкость и масштабируемость;
2. Улучшенная отказоустойчивость;
3. Независимые обновления или изменения.

- Гибкость и масштабируемость

В силу того, что данный подход представляет собой разделение логики между различными сервисами, разработчики получают возможность более удобной масштабируемости продукта. За счет того, что логика разделена между сервисами, они являются не зависимыми от реализации друг друга в

связи с чем появляется возможность более гибкой масштабируемости того или иного сервиса, при этом основная работа приложения или отдельных сервисов не будет нарушена.

- Улучшенная отказоустойчивость

Исходя из того, что приложение в микросервисной архитектуре собирается из разделенных сервисов, мы получаем улучшенную отказоустойчивость. Это означает, что независимо от того в каком из сервисов произошла ошибка или сбой, все приложение продолжит работу, кроме данного сервиса. Таким образом мы получаем изоляцию сбоев, ведь проблема распространяется только на один из сервисов, не затрагивая при этом работу остального приложения. Также сервисы могут быть запущены на отдельных серверах, что позволяет распределить нагрузку между ними и не нагружать один сервер.

- Независимые обновления и изменения

Благодаря разделению логики приложения в микросервисной архитектуре, мы получаем возможность более гибкого добавления изменений в структуру проекта. Это означает, что добавление нового функционала или изменение существующего не будет затрагивать полностью работу приложения, а также других сервисов, что позволит более удобно масштабировать данный продукт.

Однако у микросервисного подхода также существуют и недостатки:

1. Сложность управления и общения;
2. Усложненное тестирование и отладка;
3. Увеличение затрат на инфраструктуру и нагрузки на сеть.

Рассмотрим каждый из них подробнее.

- Сложность управления и общения

Управление большим количеством микросервисов, является сложной задачей. В отличие от монолитной архитектуры, для правильной работы микросервисов необходимо обеспечить согласованность данных, а также настроить их правильную передачу, что может усложнить разработку, а также привести к большей нагрузке на сеть.

- Усложненное тестирование и отладка

Так как микросервисы являются распределенной системой, запуск, тестирование, а также отладка становятся более сложными. За счет того, что микросервисы взаимодействуют друг с другом через сеть, необходимо проводить интеграционное тестирование, позволяющее выявить корректность работы компонентов между собой. Из-за распределенной системы возникающие ошибки, а точнее, их источник может быть не явным, что приводит к большему времени на отладку и повторное тестирование работы сервисов.

- Увеличение затрат на инфраструктуру и нагрузки на сеть

В микросервисной архитектуре каждый из сервисов является отдельно запущенным процессом или контейнером, в свою очередь данный процесс или контейнер использует некоторые ресурсы системы, что означает, что при росте данных сервисов будет расти и потребление ресурсов системы. Также поскольку микросервисы чаще всего общаются между собой по сети, это может привести к увеличению сетевого трафика внутри приложения, а также задержкам работы при большом количестве сервисов и интенсивной передаче данных между ними.

3. Слоистая архитектура

Слоистая архитектура или как еще ее называют многоуровневая представляет собой решение или подход к разработке программного продукта, при котором кодовая база этого продукта разбивается на своеобразные слои или отдельные уровни, каждый из которых выполняет определенные функции и в свою очередь может иметь различную реализацию.

Однако и у этого подхода, очевидно, есть свои преимущества и недостатки.

Рассмотрим преимущества данного подхода по порядку:

1. Модульность;
2. Четкая структура;
3. Удобство масштабирования.

- Модульность

Благодаря разделению логики между на слои или уровни, появляется возможность, во-первых, распределения нагрузки между модулями, во-вторых, удобство расширения такого продукта. Однако для правильной работы модулей в таком подходе необходимо организовать независимость каждого из уровней, имеется в виду, что изменения в одном модуле не должны влиять на работу другого, а также организовать четкие интерфейсы работы с каждым из модулей и уровни абстракций. Благодаря правильно построенной системе абстракций мы получаем возможность не только разделить большой системы на более мелкие подсистемы, но и возможность разбить сложные компоненты составляющие эти системы на более легкие, понятные и удобные в использовании, что позволит упростить процесс разработки.

- Четкая структура

Благодаря построению уровня абстракций между модулями слоистой архитектуры, а также реализации интерфейсов взаимодействия с этими модулями достигается более удобная и четкая структура, которая в свою

очередь позволяет расширять существующий функционал и вывести некоторый свод правил разработки, где каждый из компонентов будет реализован по некоторому плану и иметь определенный интерфейс взаимодействия что позволит гораздо быстрее разобраться в написанном коде, а также добавить новый или изменить уже существующий функционал.

- Удобство масштабирования

Слоистая архитектура позволяет масштабировать построенную систему как вертикально (увеличение ресурсов на существующих слоях), так и горизонтально (добавление новых экземпляров слоев). Это обеспечивает гибкость построенной системы.

К сожалению, и данный подход имеет свои недостатки:

1. Сложность;
2. Зависимость от абстракций,
3. Производительность.

- Сложность

В слоистой архитектуре при увеличении числа слоев, составляющих приложение, система может стать сложной для понимания и управления. Иногда возникает риск большой связанности между некоторыми слоями или их перегрузки, что усложнит разработку, отладку и сопровождение такого продукта. Также за счет роста числа слоев, может вырасти и количество используемых абстракций, что приводит к следующей проблеме.

- Зависимость от абстракций

При росте слоев может возникнуть проблема создания и использования излишнего количества абстракций, которые также могут усложнить систему и затруднить понимание и расширение такой структуры. Также это может привести к проблемам с отладкой такой системы, ведь из-за высокого уровня

используемых абстракций источник ошибки может быть не очевиден сразу, что приведет к потере большего времени при исправлении ошибки или сбоя.

- Ограничения производительности

Некоторые слои могут являться узким местом в системе из-за медленного доступа к данным, большого объема запросов или дополнительных причин, что может привести к замедлению работы зависимых слоев и ухудшению производительности системы. Также производительность системы зависит и от уровня абстракций, иногда гораздо проще, быстрее и удобнее вызвать некоторый метод или функцию на прямую чем делать это через большой уровень абстракций, создание дополнительных экземпляров объектов и так далее.

4. Чистая архитектура, как разновидность Слоистой архитектуры

Слоистая архитектура является более обширным термином, который описывает архитектурный подход, основанный на разделении системы на слои или уровни. Однако у этого подхода есть несколько конкретных реализаций. Одной из таких реализаций является «Чистая архитектура», предложенная специалистом Робертом Мартином и описанная в одноименной книге.

Чистая архитектура представляет собой конкретную реализацию Слоистой архитектуры и подразумевает использование более строгих правил и принципов для создания систем, например принципы объектно-ориентированного программирования или принципы SOLID описывающие схему построения конкретных компонентов, составляющих чистую архитектуру.

В данной главе мы рассмотрели основные принципы построения архитектуры приложений, выявили их преимущества и недостатки. Выбор конкретной реализации, очевидно, зависит от сложности реализуемого проекта.

2.2 Основные языки программирования и их фреймворки

Для реализации корпоративного мессенджера, как и любого другого продукта кроме выбора архитектуры, необходимо выбрать язык программирования. Однако в настоящее время большинство продуктов не реализуются, используя чистые возможности языка, ведь существует огромное количество библиотек и фреймворков позволяющих упростить реализацию тех или иных функций внутри приложения и даже полностью собрать все необходимое для старта проекта.

В данной главе будут рассмотрены основные языки программирования и их библиотеки позволяющие реализовать функционал как серверной, так и клиентской части корпоративного мессенджера.

Языки и их библиотеки и фреймворки используемые для реализации серверной части:

1. Java
 - Spring Framework
2. JavaScript (Node.js)
 - Express.js;
3. Python
 - Django;
 - FastAPI.

Рассмотрим каждый из языков и его библиотеки по порядку.

- Язык программирования Java

Данный язык программирования был создан в 1995 году компанией Sun Microsystems и стал одним из самых популярных языков программирования в мире благодаря списку преимуществ:

1. Платформенная независимость – означает что код, написанный на этом языке, может быть выполнен на любой виртуальной машине Java, вне зависимости от операционной системы или платформы, на которой запускается эта виртуальная машина;
2. ООП – означает что данный язык основан на объектно-ориентированной парадигме программирования и позволяет создавать расширяемы и гибкие приложения;
3. Большая библиотека – данный язык программирования поставляется с обширной стандартной библиотекой, которая содержит большое количество полезных расширений языка;
4. Многопоточность – означает что данный язык встроено поддерживает многопоточность, что позволяет эффективные приложения способные выполнять несколько задач одновременно.

Рассмотрим основной фреймворк для разработки приложений на языке Java – Spring Framework.

- Spring Framework

Данный фреймворк является одним из самых популярных и широко используемых фреймворков для языка Java. Он предоставляет большой набор инструментов и функциональности, позволяющий создавать веб-приложения, микросервисы, приложения для обработки данных и многое другое. Он предоставляет следующие возможности:

1. Управление зависимостями с помощью Инверсии управления (IoC) и Внедрения зависимостей (DI);

2. Модульность – данный фреймворк состоит из нескольких модулей позволяющих управлять транзакциями, работать с базами данных, обрабатывать веб-запросы и т.д;
 3. Spring Boost – данный модуль предоставляет удобные инструменты для настройки и управления приложениями, а также встроенную поддержку многих технологий, например, веб-серверы.
- Язык программирования JavaScript (Node.js)

Данный язык программирования был создан в 1995 программистом Бренданом Эйхом, который в тот период работал в компании Netscape Communications Corporation. Он разработал данный язык за очень короткий срок, всего за 10 дней, как скриптовый язык для веб-страниц в браузере Netscape Navigator.

JavaScript быстро занял позицию одного из основных языков программирования для веб-разработки. С течением времени и вовсе стал стандартом для создания скриптов на клиентской стороне. А затем благодаря развитию среды выполнения Node.js, стал использоваться для написания кода серверной стороны, что привело к широкому распространению языка во всех областях веб-разработки.

Преимущества языка JavaScript:

1. ООП – язык JavaScript является объектно-ориентированным языком программирования и поддерживает основные принципы данного подхода;
2. Многопоточность и асинхронность – благодаря встроенному механизму событийного цикла (Event Loop) и асинхронным функциям, он позволяет обрабатывать большое количество задач «параллельно» без блокировки основного потока выполнения;
3. Функциональное программирование – означает что функции в этом языке выступают как объекты первого класса и могут быть переданы как аргументы в другие функции или возвращены ними.

- Среда Node.js

Данная среда выполнения, построенная на движке V8 Chrome, который обеспечивает быструю и эффективную работу с JavaScript, позволяет выполнять язык JavaScript на стороне сервера, что делает его еще одним инструментом разработки серверной части приложений. Рассмотрим какие дополнения к языку JavaScript привнесла среда Node.js:

1. Модульность – данная среда разрешает использование модульности и переиспользования кода благодаря системе модулей CommonJS, которая в свою очередь позволяет разбивать код на отдельные;
2. Производительность – благодаря использованию движка V8 Chrome, Node.js обладает высокой производительностью и скоростью выполнения JavaScript-кода, учитывая что данный язык является интерпретируемым, то есть выполняется строка за строкой непосредственно в среде;
3. Асинхронно и событийно-ориентированное программирование – данная среда использует асинхронную модель ввода/вывода, что позволяет обрабатывать большое количество запросов одновременно без блокировки потоков выполнения.

- Язык программирования Python

Данный язык программирования был создан в конце 1980-х и начале 1990-х годов разработчиком Гвидо ван Россумом. В начале 1991 года он выпустил первую версию языка Python. На данный момент язык Python остается одним из наиболее популярных и востребованных языков программирования и занимает лидирующие позиции в различных сферах разработки продуктов, благодаря своим преимуществам:

1. Простота и читаемость кода – данный язык имеет очень ясный и простой синтаксис, что практически позволяет читать код даже не разбирающимся людям, также это делает его легким для изучения и

использования в целом, что также влияет, например, на работу между сотрудниками в командном проекте;

2. Кроссплатформенность – язык Python поддерживает все основные операционные системы, что позволяет разрабатывать продукты, которые работают на разных платформах без изменений в исходном коде;
3. Стандартная библиотека – данный язык поставляется с обширной стандартной библиотекой, которая содержит модули и функции позволяющие решать задачи различной сложности, что в свою очередь упрощает разработку и уменьшает зависимость от сторонних библиотек;
4. Расширяемость возможностей языка – язык Python имеет большое и активное сообщество разработчиков, которые создают библиотеки, фреймворки и инструменты для различных задач.

- Фреймворк Django

Django является высокоуровневым веб-фреймворком реализованным на языке Python, который позволяет быстро и эффективно создавать мощные веб-приложения. Данный фреймворк предоставляет разработчикам готовые инструменты и шаблоны для решения различных задач, также имеет ряд преимуществ:

1. Высокая производительность и масштабируемость – данный фреймворк изначально был разработан с учетом производительности и масштабируемости, следует принципам “Don’t repeat yourself” (DRY) и “Convention over Configuration”. Он позволяет создавать веб-приложения способные обрабатывать большие нагрузки и масштабироваться по мере необходимости;
2. ORM – Django предоставляет свою ORM (Object-Relational Mapping), которая позволяет работать с базами данных, используя объектно-

ориентированный подход и позволяет разработчикам работать с данными, как с обычными объектами Python;

3. Шаблоны – фреймворк предоставляет мощную систему шаблонов, которая позволяет разделять логику приложения и представление данных, в различные составляющие части приложения. Данные шаблоны позволяют создавать динамические HTML-страницы с использованием основных конструкций языка программирования внутри этих шаблонов.

- Фреймворк FastAPI

FastAPI является современным веб-фреймворком для языка Python, который позволяет быстро и эффективно создавать веб-приложения с помощью языка Python. Он был создан Себастьяном Рамиресом и впервые выпущен в 2018 году. Данный фреймворк разрабатывался в угоду производительности и простоты в использовании и имеет следующие преимущества:

1. Высокая производительность – FastAPI в основе использует другой асинхронный фреймворк Starlette, а также использует синтаксис языка Python 3.7 и выше позволяющие добавлять аннотации типов, что обеспечивает высокую производительность и позволяет обрабатывать большие объемы запросов;
2. Автоматическая документация и тестирование – данный фреймворк предоставляет встроенную поддержку автоматической документации и тестирования созданного API, благодаря стандарту OpenAPI. Это означает, что благодаря аннотации типов, используемой при написании, а также комментариев к коду и встроенных дополнительных надстроек, например декораторов, позволяет создавать документацию к API автоматически, а также получить возможность протестировать созданное API;

3. Поддержка асинхронности – FastAPI полностью поддерживает асинхронное программирование на языке Python, что в свою очередь позволяет создавать масштабируемые и эффективные приложения;

В данной главе мы познакомились с основными языками программирования используемых для реализации серверной части веб-приложений, а также с их фреймворками и библиотеками позволяющим упростить и ускорить этот процесс.

2.3 Базы данных

В этой главе будут рассмотрены основные типы баз данных использующихся при создании корпоративных мессенджеров и в принципе любых программных продуктов.

- Основные типы баз данных

Существует несколько различных типов баз данных, каждый из которых имеет свои особенности и применения. Рассмотрим два основных типа баз данных которые чаще всего используются при создании программных продуктов:

1. Реляционные базы данных – это базы, которые организуют хранение данных в виде таблиц, состоящих из строк и столбцов, где каждая строка является отдельной записью (отдельным объектом), а столбец представляет конкретное поле или атрибут этого объекта;
2. NoSQL (Not only SQL) – представляют собой отличный от обычных реляционных баз данных тип, который имеет множество реализаций и позволяет хранить данные в различных форматах, например документно-ориентированные базы данных MongoDB.

Рассмотрим самых ярких представителей каждого из типов по порядку.

- PostgreSQL

PostgreSQL является мощной объектно-реляционной системой управления базами данных или СУБД, которая в свою очередь предоставляет обширные возможности для обработки и хранения данных, а также является одной из самых мощных и надежных открытых реляционных баз данных.

Данная СУБД является одним из самых популярных инструментов работы с данными благодаря своим особенностям:

1. Расширенные возможности существующей реляционной модели – данная система полностью удовлетворяет стандартам SQL, а также приносит несколько дополнительных функций, упрощающих обработку и хранение данных, например: триггеры, хранимые процедуры и так далее;
2. Асинхронность и транзакции – PostgreSQL поддерживает асинхронное программирование, а также обеспечивает основные свойства транзакций (атомарность, согласованность, изолированность и устойчивость);
3. Расширяемость – эта СУБД также предоставляет широкий набор инструментов для масштабирования базы данных и управления ее производительностью

- MongoDB

Как уже было упомянуто выше MongoDB является представителем NoSQL-типа баз данных. Она имеет документ-ориентированный формат и является очень мощным инструментом, позволяющим хранить данные разной структуры. Данная система является одним из самых популярных NoSQL-решений, благодаря следующим особенностям:

1. Документ-ориентированная модель – MongoDB представляет данные в формате JSON-подобных документов, называемых BSON (Binary JSON). Каждый из таких документов содержит в себе пары ключ-

значение и может иметь абсолютно любую структуру, что позволяет очень удобно хранить разнообразные данные (например геоданные)

2. Масштабируемость и высокая производительность – Данная система обеспечивает горизонтальную масштабируемость, что в свою очередь позволяет распределять данные между серверами и добиваться высоких значений по производительности;
3. Встроенный гибкий язык запросов – В MongoDB используется язык запросов, основанный на языке JavaScript, что позволяет добиться лучшей читаемости таких запросов. Также данный язык предоставляет большой набор инструментов, таких как: операторы для поиска, фильтрации, обновления и удаления данных;

В данной главе были рассмотрены основные и одни из популярных типов систем баз данных, а также наиболее ярких представителей каждого из типов, каждый из которых предоставляет широкий набор инструментов для решения задач различной сложности

2.4 Системы кэширования

В предыдущих главах были представлены основы для создания любого программного продукта, в том числе и корпоративного мессенджера. Однако используя только эти технологии можно столкнуться с проблемой задержки данных. Например, при плохом соединении пользователь может долго ожидать отправки сообщения или загрузки чата, при этом даже если соединение сети хорошее, а устройство пользователя позволяет обрабатывать данные быстро, можно столкнуться с проблемой повторения запросов и затраты ресурсов и времени на получение данных, которые пользователь получал, возможно, несколько секунд назад. Нецелесообразно каждый раз на действие пользователя отправлять запрос, который будет нагружать систему в попытка получить из базы данных полученную раньше информацию. Для этого полученную информацию, не всю, но та что скорее всего понадобится

пользователю несколько раз, необходимо где-то хранить. Хранилищами таких данных являются системы кэширования, позволяющие сохранить необходимые данные в быстродействующей памяти (кэше) с целью ускорения доступа к этим данным.

Существуют различные типы кэширования позволяющие хранить данные в разными способами:

1. Локальное кэширование – такой тип сохраняет данные в памяти на уровне приложения или операционной системы на одном конкретном узле, что обеспечивает быстрый доступ, однако не подходит для масштабирования приложений;
2. Распределенное кэширование – данный тип, наоборот, распределяет данные по нескольким узлам или серверам, что обеспечивает масштабируемость, но может привести к чуть большему времени получения данных. Такой тип часто применяется в микросервисной архитектуре.

Рассмотрим системы кэширования на одной из самых популярных систем Redis

- Redis

Redis (Remote Dictionary Server) является высокопроизводительной системой управления данными использующаяся как система кэширования и позволяет хранить данные в виде пар ключ-значение. Кроме того данная система имеет еще несколько особенностей, что делает ее не только системой кэширования, но также и системой для реализации очереди сообщений (брокером сообщений):

1. Поддержка различных структур данных – Redis позволяет хранить данные в виде различных типов данных, например: списки, которые могут использоваться для реализации хранения сообщений или

журналов событий, множества, позволяющие хранить уникальные значения и так далее

2. Поддержка транзакций и атомарных операций – данная система кэширования позволяет выполнять атомарные операции над данными и даже объединять несколько таких операций в транзакции, что в свою очередь обеспечивает целостность данных и предотвращает ситуации неправильных мутаций данных;
3. Pub/Sub (публикация/подписка) – в Redis встроен механизм, который позволяет создавать асинхронные сообщения между различными компонентами приложения. Эта система используется для реализации каналов связи, где одни клиенты могут публиковать сообщения, а другие клиенты подписываться на необходимые каналы и перехватывать опубликованные сообщения

Системы кэширования позволяют увеличить производительность вашего приложения, некоторые из них, такие как Redis также предоставляют дополнительные возможности, например асинхронного общения между компонентами или сервисами вашего продукта. В совокупности эти удобства позволяют улучшить и пользовательский опыт, ведь приложение будет менее подвергнуто зависаниям и простоям в ожидание получения необходимой информации.

2.5 Протоколы передачи данных

Кроме проблемы доступности данных и скорости их загрузки при разработке большого проекта (корпоративный мессенджер) может возникнуть проблема, связанная с передачей данных между клиентом и сервером. Для передачи данных, собственно, существуют протоколы их передачи. С некоторыми из них мы знакомы из курса сетевого программирования и сталкивались с ними непосредственно в жизни.

Одним из таких протоколов является протокол HTTP и его безопасная версия HTTPS. Данный протокол является основным протоколом передачи

данных, он определяет правила обмена информацией между клиентами и серверами. Основным принципом работы данного протокола является принцип запрос-ответ, при котором клиент отправляет определенный запрос (GET, POST, PUT, DELETE) содержащий необходимую для сервера информацию (заголовки, данные) и получает ответ от сервера, который также хранит некоторые данные необходимые клиенту. Возможно, это информация, полученная из базы данных или от другого API. В любом случае можно выделить одну особенность таких запросов, это метаданные, которые передаются вместе с необходимыми данными, те же самые заголовки и так далее.

Однако такая передача информации не всегда выгодна, основные проблемы могут возникнуть при большом количестве запросов от клиентов, например при отправке сообщения клиентом. Получается, что для каждого написанного сообщения, сервер должен принять и обработать один из запросов, сформировать ответ, содержащий необходимую информацию и отправить обратно клиенту. Здесь возникает другая проблема, очевидно, что в реализации мессенджера, нам нужно поддерживать одну и ту же версию конкретного чата у двух и более пользователей. Но если пользователи могут отправлять и принимать данные только с помощью HTTP-запросов, то моментально оповещение других пользователей о новом сообщении невозможно.

Одним из решений является отправка запросов на обновление текущего состояния, клиентами к серверу. Однако если использовать обычные HTTP-запросы это приведет к перегрузке системы из-за огромного количества принимаемых и обрабатываемых запросов. Но данная проблема имеет решение, реализованное в различных подходах или протоколах, рассмотрим их по порядку.

- Long pooling

Long pooling – это специальная техника коммуникации между клиентом и сервером позволяющая избежать большого количества запросов, благодаря своему принципу работы. В отличие от традиционных запросов, long pooling сервер поддерживает открытое соединение с клиентом и задерживает ответ до тех пор, пока не будет доступна некоторая обновленная информация, которую можно отправить обратно клиенту. Это означает, что когда клиент отправляет запрос на сервер, сервер в свою очередь, не сразу возвращает ответ пользователю, а оставляет это соединение открытым до момента появления новой информации необходимой пользователю. Ожидание ответа также имеет ограничение или таймаут по истечению которого в случае отсутствия ответа от сервера клиент отправит новый запрос на обновление. Таким образом мы сможем сократить количество ненужных запросов, нагружающих сеть.

- WebSocket

WebSocket – это протокол передачи данных, которые устанавливает двустороннюю связь между клиентом и сервером через одно открытое TCP/IP соединение между клиентским браузером и сервером. Такой подход позволяет обеим сторонам обмениваться данными в реальном времени без отправки дополнительных запросов на обработку.

Встроенный протокол рукопожатия (handshake protocol) с помощью отправки запроса по защищенному протоколу HTTPS позволит обмениваться необходимой информацией между клиентом и сервером для установки постоянного TCP/IP соединения. Это означает, что в случае успешного рукопожатия, соединение переходит в режим WebSocket, который в свою очередь не использует дополнительные заголовки и метаданные, что во многом уменьшает размер передаваемых пакетов и позволяет обмениваться только необходимой информацией.

2.6 Системы контейнеризации (Docker)

Во время обсуждения архитектур также был рассмотрен такой этап реализации приложения, как его непосредственный запуск и сборка или тестирование. Если разработка продукта ведется небольшой командой, то данная проблема может и не возникнуть, однако в больших командах, где процесс реализации разделен на несколько этапов таких как: разработка, написание тестов разработчиками, ревью написанного кода, тестирование командой тестировщиков – может возникнуть проблема запуска нужной версии продуктов на каждом из этапов в каждой из команд. Одним из решений, конечно, является использование системы контроля версий и тэгирования определенных готовых версий для более удобной сборки проектов. Однако возникшую проблему одна система контроля версий решить не сможет. Запуск такого продукта на разных этапах заставит разные команды, возможно не разбирающихся в конкретной реализации, общаться с коллегами для клонирования репозитория такого проекта, установки зависимостей и многое другое, что является лишними взаимодействиями и приведет к трате драгоценного времени, которое можно было сконцентрировать на доработку продукта или разработку новых возможностей.

Для решения проблемы взаимодействия существуют системы контейнеризации, например Docker, позволяющие упаковывать готовые приложения и их зависимости в легковесные, изолированные контейнеры, которые в свою очередь позволяют развернуть нужную версию приложения без надобности в установке и клонировании необходимых зависимостей.

Контейнеры обеспечивают консистентную среду выполнения для приложений и позволяют упаковать все необходимое в нечто, однако гораздо более легковесное и эффективное, похожее на виртуальную машину. Благодаря этому разработчики смогут делиться готовыми собранными версиями приложения с другими командами не опасаясь, что нужные зависимости не будут установлены или клонированы.

Такие контейнеры можно собирать в большие взаимодействующие между собой системы. Ведь система докер позволяет настроить внутреннюю сеть для общения нескольких контейнеров между собой.

Сборка такого проекта уместается в один Dockerfile содержащий версии необходимых изображений (например языка программирования), перечисления необходимых для копирования директорий и файлов, а также команд запуска проекта.

Также существует система Docker Compose позволяющая запустить несколько контейнеров внутри одной сети, описав все необходимое внутри файла docker-compose.yml, содержащего параметры необходимые для запуска каждого из контейнеров.

2.7 Методы шифрования данных

Кроме выбора архитектуры, языка и других различных технологий, для создания корпоративного мессенджера и не только его, но и многих других различных программных продуктов, необходимо использовать шифрование данных. В основном это необходимо, чтобы никто другой, кроме клиентов системы не смог получить доступ к важным данным. Для небольших проектов не использующих и не хранящих конфиденциальную информацию пользователей, например, как переписка, можно обойтись защищённой версией протокола HTTP (HTTPS). Однако в крупных проектах необходимо использовать дополнительное шифрование позволяющее закодировать передаваемую пользователем информацию и хранить ее в базе в зашифрованном виде, что позволит даже в случае утечки данных из базы сохранить их защищенность. Давайте рассмотрим несколько существующих и широко используемых алгоритмов шифрования данных.

- AES

Advanced Encryption Standard (AES) – является симметричным алгоритмом шифрования. Данный алгоритм считается одним из самых

безопасных и эффективных алгоритмов шифрования. Основные характеристики данного алгоритма, следующие:

1. Симметричность – означает что данный шифр использует один и тот же ключ для шифрования и расшифровки
2. Блочный шифр – AES использует блоки фиксированного размера для шифрования, также он разбивает входные на данные блоки и шифрует каждый из них отдельно;
3. Длина ключа – данный алгоритм поддерживает ключи различной длины (128, 192 и 256 бит). Чем длиннее ключ тем большую безопасность он обеспечивает, хоть и может замедлить процесс шифрования и расшифровки.

- Протокол Диффи-Хеллмана

Протокол Диффи-Хеллмана – это метод обмена ключами, позволяющий клиентам безопасно согласовать общий секретный ключ между собой через открытый канал связи. Данный протокол является основой для многих систем шифрования и обеспечения безопасности, таких как SSL/TLS, SSH и другие. Основная идея работы заключается в следующем:

1. Обе стороны договариваются о некоторых общих параметрах, таких как простое число P и генератор G ;
2. Каждая сторона генерирует секретный ключ
3. Из сгенерированного секретного ключа каждая сторона генерирует открытый ключ, который является результатом возведения генератора G в степень своего секретного ключа по модулю P ;
4. Клиенты обмениваются ключами;
5. Каждый из клиентов вычисляет общий секретный ключ, который является результатом возведения полученного от другой стороны открытого ключа в степень своего секретного ключа по модулю P . Полученный общий секретный ключ используется для шифрования и расшифровывания сообщений.

- RSA

RSA (Rivest-Shamir-Adleman) – это ассимитричный метод шифрования, который использует публичный и приватный ключ для шифрования и расшифровки. Данный метод получил широкое распространение благодаря математической прочности и безопасности, которую он предоставляет. Основные этапы работы заключается в следующем:

1. Генерируется два ключа: приватный и публичный, приватный ключ является секретным и используется для расшифровки данных, а публичный ключ распространяется и используется для шифрования данных;
2. Полученные данные разбиваются на блоки и каждый из блоков шифруется публичным ключем;
3. Для расшифровки полученное расшифрованное сообщение разбивается также на блоки и каждый из них расшифровывается приватным ключем.

3 Реализация собственного корпоративного мессенджера

В данной главе будет представлена реализация функционала серверной и клиентской частей собственного корпоративного мессенджера, рассмотрены основные технологии, используемые при создании.

Серверная часть

Языком программирования, использовавшимся для создания серверной части, был выбран Python. Также для реализации собственного API был выбран фреймворк FastAPI, в качестве системы кэширования используется Redis, а за контейнеризацию отвечает Docker и модуль Docker Compose.

3.1 Архитектура

Для данного проекта была выбрана монолитная архитектура. В угоду скорости разработки, простоты развертывания и, учитывая реализацию только основных функций таких как общение, небольшого масштаба приложения.

Для создания компонентов и разделения логики была выбрана чистая архитектура, описанная Робертом Мартином. То есть сам проект имеет монолитную структуру, однако взаимодействия между компонентами этой структуры построены на основе чистой архитектуры. Проект разделен на модули составляющие все приложение. На рисунке 1 можно увидеть, как выглядит структура проекта. Основным модулем является модуль app содержащий также внутри себя файл main.py, который в свою очередь содержит реализацию создания экземпляра приложения и подключения основных роутов API. Все остальные модули разделены в рамках своих функций где например модуль config содержащий файл __init__.py, а также сам конфигурационный файл config.toml реализует класс для обработки и чтения файлов конфигурации.

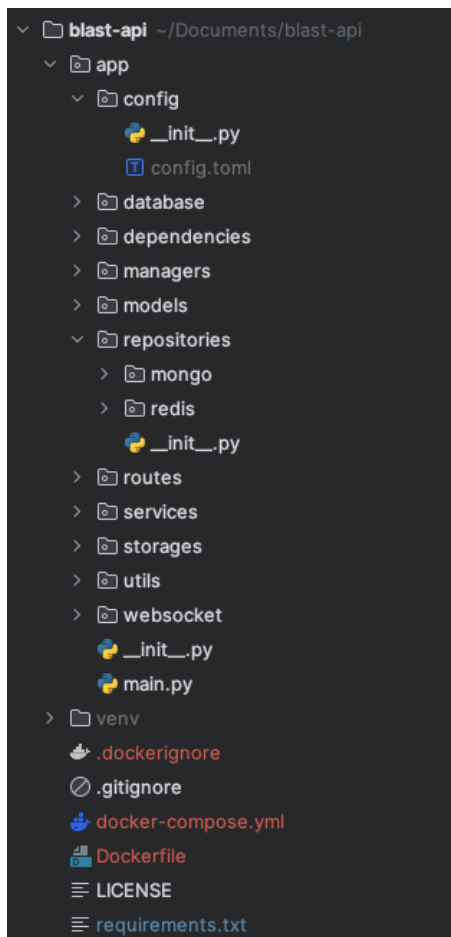


Рисунок 1 – структура проекта

Также для разделения бизнес-логики был использован паттерн проектирования «Репозиторий». Суть паттерна заключается в том, что разработчик строит несколько объектов, так называемых репозиториях для взаимодействия с базой данных и разными другими сущностями. Тем самым вынося бизнес-логику для работы с ними и предоставляя единый интерфейс взаимодействия для всех остальных компонентов проекта.

Можно заметить, что на рисунке 1, в структуре проекта репозитории являются отдельным реализованным модулем repositories.

Для репозитория работы с базой данных была создан абстрактный класс, описывающий сигнатуру обязательных методов этого класса. Такой подход был выбран для возможности реализации одного из принципов SOLID, а именно инверсию зависимостей (Dependency Inversion) и позволяет

использовать такую технику, как Dependency Injection, позволяющую передавать в компонент необходимые зависимости рисунок 2.

```
from abc import ABC, abstractmethod

6 usages  ↗ KOVALSKI +1
class DataBaseRepository(ABC):
    6 usages  ↗ KOVALSKI +1
    @abstractmethod
    async def get_one(self, table_name: str, filter_by: list, projection: dict = None):
        raise NotImplementedError

    ↗ KOVALSKI +1
    @abstractmethod
    async def get_all(self, table_name: str, filter_by: list, projection: dict = None):
        raise NotImplementedError

    ↗ KOVALSKI
    @abstractmethod
    async def insert_one(self, table_name: str, data: dict):
        raise NotImplementedError

5 usages (5 dynamic)  ↗ KOVALSKI
    @abstractmethod
    async def insert_many(self, table_name: str, data: list):
        raise NotImplementedError

    ↗ KOVALSKI
    @abstractmethod
    async def edit(self, table_name: str, update_data: dict, filter_by: list):
        raise NotImplementedError

    ↗ KOVALSKI
    @abstractmethod
    async def delete_one(self, table_name: str, filter_by: list):
        raise NotImplementedError
```

Рисунок 2 — реализация абстрактного класса репозитория

Внутри модуля repositories кроме абстрактного класса репозитория, также хранятся конкретные реализации этой абстракции. Например, подмодуль mongo реализует нужную абстракцию и позволяет работать с базой MongoDB, сокращенный код можно увидеть на рисунке 3.

```

from motor.motor_asyncio import AsyncIOMotorCollection

from app import MONGO_CLIENT
from app.repositories import DataBaseRepository

from typing import Type
from pydantic import BaseModel

17 usages  ⚡ KOVALSKI +1
class MongoRepository(DataBaseRepository):
    client = MONGO_CLIENT

    ⚡ KOVALSKI
    def __init__(self, model: Type[BaseModel]):
        self.model = model

    ⚡ KOVALSKI +1
    async def get_one(self, table_name: str, filter_by, projection: dict = None):...

    ⚡ KOVALSKI +1
    async def get_all(self, table_name: str, filter_by, projection: dict = None):...

    ⚡ KOVALSKI
    async def insert_one(self, table_name: str, data: dict):...

    5 usages (5 dynamic)  ⚡ KOVALSKI
    async def insert_many(self, table_name: str, data: list):...

    ⚡ KOVALSKI
    async def edit(self, table_name: str, update_query: dict, filter_by):...

    ⚡ KOVALSKI
    async def delete_one(self, table_name: str, filter_by):...

```

Рисунок 3 — реализация абстрактного класса для доступа к MongoDB

Также для полноты реализации чистой архитектуры был создан дополнительный модуль сервисов, обеспечивающий методы для работы с чатами, пользователями, а также сервис работы с Redis Pub/Sub. Данные сервисы позволяют вынести необходимую бизнес логику в методы и освободить роуты от не нужного копирования существующего кода, а также от большого количества бизнес-логики содержащейся в нем. Что также удовлетворяет принципу DRY (don't repeat yourself) реализация нескольких сервисов показана на рисунках 4 и 5.

```

class ChatsService:
    """ KOVALSKI """
    def __init__(
        self,
        chats_repo: DataBaseRepository,
        cache_repo: RedisRepository = None
    ):
        self.chats_repo = chats_repo
        self.users_service = UsersService(MongoRepository(MongoUser))
        self.cache_repo = cache_repo
        self.collection_name = "chats"
        self.connections = ConnectionsStorage()

4 usages """ KOVALSKI +1
    async def find_chat_by_id(self, chat_id: ObjectId) -> Union[Chat, None]:...

2 usages """ KOVALSKI
    async def find_chat_by_members(self, members: List[str]) -> Union[Chat, None]:...

2 usages """ KOVALSKI +1 *
    async def find_user_chats(self, username: str) -> List[Chat]:...

1 usage """ KOVALSKI +1 *
    async def get_user_chat(self, username: str, chat_id: ObjectId) -> Union[None, Chat]:...

2 usages """ KOVALSKI *
    async def add_name_to_chat(self, chat: Chat, username):...

1 usage """ KOVALSKI +1
    async def save_chat_to_cache(self, chat: Chat):...

1 usage """ KOVALSKI +1
    async def get_chat_from_cache_by_id(self, chat_id: ObjectId):...

```

Рисунок 4 — реализация сервиса чатов

Как видно сервис представляет из себя класс содержащий инициализирующий метод `__init__()` внутри которого создаются экземпляры необходимых репозиториев, в свою очередь можно заметить использование абстрактных классов репозиториев в параметрах данного инициализирующего метода. Таким образом, в данный сервис могут быть переданы любые дочерние классы реализующие абстрактные классы репозиториев, что позволяет избавиться от зависимости от конкретной реализации.

```

from fastapi.exceptions import HTTPException

from app.models import *

from app.repositories import DataBaseRepository
from app.repositories.redis import RedisRepository

14 usages  ± KOVALSKI
class UsersService:
    ± KOVALSKI
    def __init__(self, users_repo: DataBaseRepository, cache_repo: RedisRepository = None):
        self.users_repo = users_repo
        self.cache_repo = cache_repo
        self.collection_name = "users"

3 usages  ± KOVALSKI
async def find_registered_user(self, user: Union[RegistrationUser, LoginUser, TokenUser]) -> Union[MongoUser, None]:...

± KOVALSKI
async def find_user_by_id(self, user_id: ObjectId) -> Union[MongoUser, None]:...

4 usages  ± KOVALSKI
async def find_user_by_username(self, username: str) -> Union[MongoUser, None]:...

± KOVALSKI
async def find_user_by_email(self, email: str) -> Union[MongoUser, None]:...

1 usage  ± KOVALSKI
async def get_users(self) -> List[MongoUser]:...

1 usage  ± KOVALSKI
async def create_user(self, user: RegistrationUser) -> Union[MongoUser, None]:...

```

Рисунок 5 — реализация сервиса работы с пользователями

Сервис пользователей имеет похожую реализацию за исключением того, что работает с другой коллекцией данных и другой моделью, и соответственно содержит реализацию других методов.

3.2 Шифрование

Для данного проекта алгоритмом шифрования был выбран RSA. Имеющий большое количество возможных готовых библиотек, он предоставляет прочную безопасность и надежное шифрование. Алгоритм работы методов шифрования, следующий:

1. Пользователь регистрируется или авторизуется
2. После успешной регистрации или авторизации, сервер генерирует два ключа приватный и открытый
3. Приватный ключ сохраняется вместе с данными пользователя

4. Публичный ключ отправляется на сторону клиента и сохраняется в `SessionStorage`, для безопасности от утечки ключа через общедоступные ресурсы, такие как `Cookies` или `LocalStorage`
5. При отправке сообщения от клиента, происходит шифрование контента сообщения публичным ключем
6. Сервер получает сообщение и расшифровывает его
7. После чего сервер хэширует сообщение и сохраняет его в коллекции чата
8. После сервер отправляет сообщение всем активным пользователям используя для шифрования их закрытый ключ
9. Клиенты при получении сообщения расшифровывают его с помощью своего публичного ключа

Шифрование на стороне сервера реализовано с помощью библиотеки `cryptography` предоставляющей обширный набор инструментов для работы с разными алгоритмами шифрования. Методы для работы с генерациями ключей расшифровки и зашифровки данных вынесены в отдельный модуль `utils` рисунок 6.

```

def generate_private_key():
    """Генерация пары ключей"""
    private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048,
        backend=default_backend()
    )
    return private_key

4 usages new *
def load_primary_key(user: MongoUser):
    """Подгрузка приватного ключа из модели"""
    private_key = serialization.load_pem_private_key(
        user.private_key.encode(),
        password=None,
        backend=default_backend()
    )

    return private_key

new *
def encrypt_data(message: Message, public_key):
    """Шифрование данных с помощью RSA и дополнительным хэшированием"""
    encrypted_data = public_key.encrypt(
        message.content.encode('utf-8'),
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )
    return encrypted_data

new *
def decrypt_data(message: Message, private_key):
    """Расшифровка данных с помощью RSA и дополнительного хэшированием"""
    decrypted_data = private_key.decrypt(
        message.content,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )
    return decrypted_data.decode('utf-8')

```

Рисунок 6 – Реализация методов шифрования на стороне сервера

В свою очередь на стороне клиента реализован класс Cipher реализующий методы шифрования и расшифровки данных, а также установки в SessionStorage и получения из SessionStorage публичного ключа пользователя рисунок 7.

```

export class Cipher {
  no usages
  constructor() {
    this.publicKey = null;
    this.privateKey = null;
    this.encryptor = new JSEncrypt();
    this.decryptor = new JSEncrypt();

    this.encryptor.setRsaOptions({
      encryptionScheme: 'oaep',
      hash: 'sha256',
    });
    this.decryptor.setRsaOptions({
      encryptionScheme: 'oaep',
      hash: 'sha256',
    });
  }

  1 usage
  setPublicKey(publicKeyPEM) {
    this.publicKey = publicKeyPEM;
    this.encryptor.setPublicKey(publicKeyPEM);
  }

  1 usage
  setPrivateKey(privateKeyPEM) {
    this.privateKey = privateKeyPEM;
    this.decryptor.setPrivateKey(privateKeyPEM);
  }

  1 usage
  encryptData(data) {
    if (!this.publicKey) {
      throw new Error( message: 'Public key is not set');
    }
    return this.encryptor.encrypt(data);
  }

  1 usage
  decryptData(encryptedData) {
    if (!this.privateKey) {
      throw new Error( message: 'Private key is not set');
    }
    return this.decryptor.decrypt(encryptedData);
  }
}

```

Рисунок 7 – Класс реализации шифрования на стороне клиента

3.3 База данных MongoDB

Как уже было упомянуто выше, для данного проекта была выбрана база данных MongoDB. За счет большой производительности, а также возможности хранить данные различных форматов, ее использование в проекте становится очень хорошим решением.

Для подключения к базе данных был также реализован отдельный абстрактный класс в модуле database, описывающий сигнатуры необходимых методов. Реализация данного класса показана на рисунке 8.

```
import logging

from abc import ABC, abstractmethod

4 usages  ↗ KOVALSKI
class DataBaseClient(ABC):

    __base_client_name__ = __name__

    ↗ KOVALSKI
    def __init__(self):
        self.log = logging.getLogger(self.__base_client_name__)

    ↗ KOVALSKI
    @abstractmethod
    def connect(self):
        raise NotImplementedError("This method should be implemented")
```

Рисунок 8 – Абстрактный класс подключения к базе данных

Внутри описанного модуля кроме абстрактного класса также содержатся подмодули с классами, наследующими и реализующими необходимые методы данной абстракции, например модель mongo представленный на рисунке 9. Данный класс представляет собой клиент подключения к базе данных MongoDB. Также данный класс добавляет к методу подключения connect, дополнительные свойства позволяющие

получить созданный экземпляр подключения к базе данных, а также экземпляр самой базы.

```
from app.database import DataBaseClient
from fastapi.exceptions import HTTPException

from motor.motor_asyncio import AsyncIOMotorClient
from app.config import config

2 usages  ▾ KOVALSKI *
class MongoDBClient(DataBaseClient):
    ▾ KOVALSKI *
    def __init__(self):
        super().__init__()

    self.connection_string = config['mongodb']['uri_test']
    self.__database_name = config['mongodb']['database']['name']
    self.__mongo_client = None
    self.__mongo_db = None

    ▾ KOVALSKI
    @property
    def client(self):
        return self.__mongo_client

    ▾ KOVALSKI
    @property
    def database(self):
        return self.__mongo_db

    ▾ KOVALSKI
    def connect(self):
        """Метод подключения к базе данных MongoDB"""
        try:
            self.__mongo_client = AsyncIOMotorClient(self.connection_string)
            self.__mongo_db = self.__mongo_client[self.__database_name]
        except Exception as err:
            self.log.error(f"Произошла ошибка при подключении к базе: {err}")
            return HTTPException(500, "Ошибка подключения")
```

Рисунок 9 – Реализация клиента подключения к базе данных MongoDB

Для работы с необходимыми моделями был создан отдельный модуль `models`, содержащий реализацию нужных моделей данных. В свою очередь для описания моделей использовалась библиотека `Pydantic` являющаяся своего рода ORM, и позволяющая очень удобно собирать модели в нужный формат для выгрузки, например JSON или BSON. Реализацию нескольких моделей мы можем увидеть на рисунке 10.

```

class PyObjectId(ObjectId):
    # KOVALSKI
    @classmethod
    def __get_pydantic_core_schema__(cls, source: Type[Any], handler: GetCoreSchemaHandler):
        return core_schema.no_info_plain_validator_function(cls.validate)

    # KOVALSKI
    @classmethod
    def validate(cls, v):
        if not ObjectId.is_valid(v):
            raise ValueError("Invalid objectid")
        return ObjectId(v)

    # KOVALSKI
    def __str__(self):
        return super().__str__()

6 usages # KOVALSKI
class BaseMongoModel(BaseModel):
    id: PyObjectId = Field(default_factory=PyObjectId, alias="_id")
    create_timestamp: str = datetime.utcnow().isoformat()

    # KOVALSKI
    @field_serializer("id")
    def serialize_id(self, id, _info):
        return str(id)

5 usages # KOVALSKI
class RegistrationUser(BaseModel):
    username: str
    password: str
    email: EmailStr
    name: str
    surname: str

3 usages # KOVALSKI
class LoginUser(BaseModel):
    username: Optional[str] = None
    password: str
    email: Optional[EmailStr] = None

```

Рисунок 10 — Реализация нескольких моделей данных

3.4 Роутинг и реализация API

Снова обращаясь к структуре, можно увидеть модуль routes, реализовывающий основное API мессенджера, рассмотрим несколько роутов содержащихся в данном модуле реализация данных роутов показана на рисунках 11 и 12.

```
@router.post('/register')
async def registration(user: RegistrationUser):
    """Регистрация пользователя"""

    user_service = UsersService(MongoRepository(MongoUser))
    existing_user = await user_service.find_registered_user(user)

    if existing_user:
        raise HTTPException(403, f"Пользователь с такими данными уже существует")

    token_user = await user_service.create_user(user)

    if not token_user:
        raise HTTPException(500, f"Не удалось создать пользователя {user.username}")

    token_user = TokenUser(**token_user.model_dump())

    jwt_token = encode_jwt_token(token_user.model_dump())

    return {
        "token": jwt_token,
        **token_user.model_dump()
    }

# KOVALSKI *
@router.post("/login")
async def login(user: LoginUser):
    """Аутентификация пользователя"""

    user_service = UsersService(MongoRepository(MongoUser))
    token_user = await user_service.find_registered_user(user)

    if not token_user:
        raise HTTPException(401, "Пользователь не найден")

    token_user = TokenUser(**token_user.model_dump())

    jwt_token = encode_jwt_token(
        token_user.model_dump()
    )

    return {
        "token": jwt_token,
        **token_user.model_dump()
    }
```

Рисунок 11 — Реализация роутов регистрации и аутентификации

На рисунке 7 можно увидеть реализацию роутов отвечающих за регистрацию и авторизацию пользователя. Также для более удобной авторизации и во избежание постоянного запроса на аутентификацию от пользователя, используются JWT-токены позволяющие определить авторизовался пользователь или нет. А установленный срок действия токена позволит хранить его только определенное время, по истечению которого токен будет не валиден.

```

KOVALSKI
@router.get("/all", description="Получение всех чатов пользователя")
async def get_all_user_chats(token: dict = Depends(token_security)):
    token_user = TokenUser(**token)
    chats_service = ChatsService(MongoRepository(Chat))

    user = await UsersService(
        MongoRepository(MongoUser)
    ).find_user_by_username(token_user.username)

    if not user:
        raise HTTPException(401, "Пользователь не зарегистрирован")

    chats = await chats_service.find_user_chats(user.username)

    return chats

KOVALSKI +1
@router.get("/{chat_id}", description="Получение чата пользователя по id чата")
async def get_user_chat_by_id(chat_id: str, token: dict = Depends(token_security)):
    token_user = TokenUser(**token)
    chats_service = ChatsService(MongoRepository(Chat), RedisRepository(Chat))

    user = await UsersService(
        MongoRepository(MongoUser)
    ).find_user_by_username(token_user.username)

    if not user:
        raise HTTPException(401, "Пользователь не зарегистрирован")

    chat_id = ObjectId(chat_id)
    chat = await chats_service.get_user_chat(user.username, chat_id)

    return chat
```

Рисунок 12 – Часть реализации роутов чатов пользователя

Также отдельным роутом выделена часть работы с установкой WebSocket подключения, в данном роуте мы можем увидеть еще один объект, используемый для централизации работы с подключениями ConnectionManager он находится в модуле managers. Реализация данного роута представлена на рисунке 13.

```
router = APIRouter(
    prefix='/communication',
    tags=['communication']
)

JWT_KEY = config['keys']['jwt']

# KOVALSKI
@router.websocket("/ws/{token}")
async def websocket_connection(websocket: WebSocket, token):
    """
    Установка websocket соединения с клиентом,
    формирование асинхронных обработчиков redis и websocket сообщений
    """
    user_service = UsersService(MongoRepository(MongoUser), RedisRepository(MongoUser))
    connection_manager = ConnectionManager()

    user_dict_from_token = utils.decode_jwt_token(token)
    user_token_model = TokenUser(**user_dict_from_token)

    user_from_db = await user_service.find_registered_user(user_token_model)

    if not user_from_db:
        raise HTTPException(404, "Пользователь не найден")

    await connection_manager.connect(user_from_db, websocket)

    websocket_messages_handler_task = asyncio.create_task(connection_manager.handle_websocket_messages(user_from_db))
    redis_messages_handler_task = asyncio.create_task(connection_manager.handle_redis_messages(user_from_db))

    await asyncio.gather(websocket_messages_handler_task, redis_messages_handler_task)
```

Рисунок 13 – Реализация роута установки WebSocket-соединения с клиентом

3.5 Контейнеризация и Docker

Для удобства сборки и настройки проекта использовалась система контейнеризации Docker, а также модуль Docker Compose позволяющие объединить сборку нескольких контейнеров в описание одного

конфигурационного файла. Реализация каждого из файлов представлены соответственно на рисунках 14 и 15.

```
FROM python:3.8-slim

WORKDIR /api

COPY ./app ./app
COPY ./requirements.txt .

RUN apt-get update
RUN apt-get install -y gdal-bin libgdal-dev g++
RUN apt-get install -y build-essential cmake libboost-dev libexpat1-dev zlib1g-dev libbz2-dev
RUN pip3 install -r requirements.txt

CMD uvicorn app.main:app --host 0.0.0.0 --port 8080
```

Рисунок 14 – Конфигурационный файл Docker

Структура Dockerfile, описывающая схему сбора контейнера не является чем-то сложным. Здесь используются простые конструкции позволяющие указать на основе какого изображения будет запускаться тот или иной контейнер (1 строка), путь основной директории, хранящей проект (3 строка), обязательные к копированию файлы (5,6 строка), а также необходимые для запуска установки зависимостей и проекта команды.

```
version: '3.8'
services:
  api:
    build: .
    ports:
      - "3030:8080"
    networks:
      - blast_network
    restart: always
  mongodb:
    image: "mongodb/mongodb-community-server:latest"
    ports:
      - "27017:27017"
    networks:
      - blast_network
    restart: always
  redis:
    image: "redis/redis-stack:latest"
    ports:
      - "6379:6379"
      - "8001:8001"
    networks:
      - blast_network
    restart: always
networks:
  blast_network:
```

Рисунок 15 – Конфигурационный файл Docker Compose

Файл модуля Docker Compose содержит в себе описание сборки сразу нескольких контейнеров, так, например для указания конкретных инструкций сборки проекта указывается путь к текущей директории, содержащей в себе Dockerfile (4 строка). Также в особенностях можно выделить описание внутренней сети, в каждом из сервисов описывается характеристика networks содержащая название сети, к которой будет принадлежать контейнер.

Клиентская часть

Языком программирования реализации клиентской части был выбран JavaScript, в свою очередь фреймворком позволяющими создавать одностраничные веб-приложения (SPA) был выбран Vue, также была использована библиотека элементов Vuetify для более быстрой сборки фронтенд-части с наименьшим количеством необходимого стилизования тех или иных компонентов. Для управления состояния веб-приложения использовался менеджер состояний Vuex.

3.6 Взаимодействие клиента с приложением

Далее будет рассмотрено взаимодействие клиента непосредственно с реализованным приложением.

Для начала взаимодействия с другими пользователями клиенту необходимо пройти регистрацию или авторизацию, если у него уже существовал аккаунт, как показано на рисунке 16.

Регистрация

Уже есть аккаунт? [Войдите](#)

Рисунок 16 – Форма регистрации клиента

Если у пользователя уже существовал аккаунт, у него есть возможность войти в него перейдя по выделенной ссылке ниже, после чего его перебросит на страницу авторизации рисунок 17.

Blast!

Вход

ВОЙТИ

Еще нет аккаунта? [Зарегистрируйтесь](#)

Рисунок 17 – Страница авторизации пользователей

После ввода данных в формы авторизации или регистрации как показано на рисунке 18 и 19 пользователь нажимает на кнопку регистрации или кнопку войти, после чего, в случае успешной регистрации или авторизации в куки браузера пользователя сохраняется JWT-токен авторизации пользователя, а самого клиента перенаправляют на страницу чатов как показано на рисунке 20.

Blast!

Регистрация

Имя

Фамилия

Email

Логин

Пароль

ЗАРЕГИСТРИРОВАТЬСЯ

Уже есть аккаунт? [Войдите](#)

Рисунок 18 – Заполненные поля формы регистрации

Blast!

Вход

Логин или email

Пароль

ВОЙТИ

Еще нет аккаунта? [Зарегистрируйтесь](#)

Рисунок 19 – Заполненные поля формы авторизации

Если у пользователя не существовало чатов, то на странице будет отображена подсказка со ссылкой на поиск пользователей и создания чата рисунок 20.

Blast!

Чаты

Команды



Находите своих
друзей или коллег и
начинайте чаты!

НАЙТИ



Рисунок 20 – Пустая страница чатов с подсказкой

Как мы видим кроме подсказки на странице появилось основное меню, содержащее кнопки навигации по страницам чатов, пользователей и настроек аккаунта, а также над подсказкой отображаются две дополнительные кнопки переключения страниц Чаты и Команды.

При нажатии кнопки найти, клиента перебросит на страницу поиска других пользователей, на которой можно будет начать диалог как показано на рисунке 21.

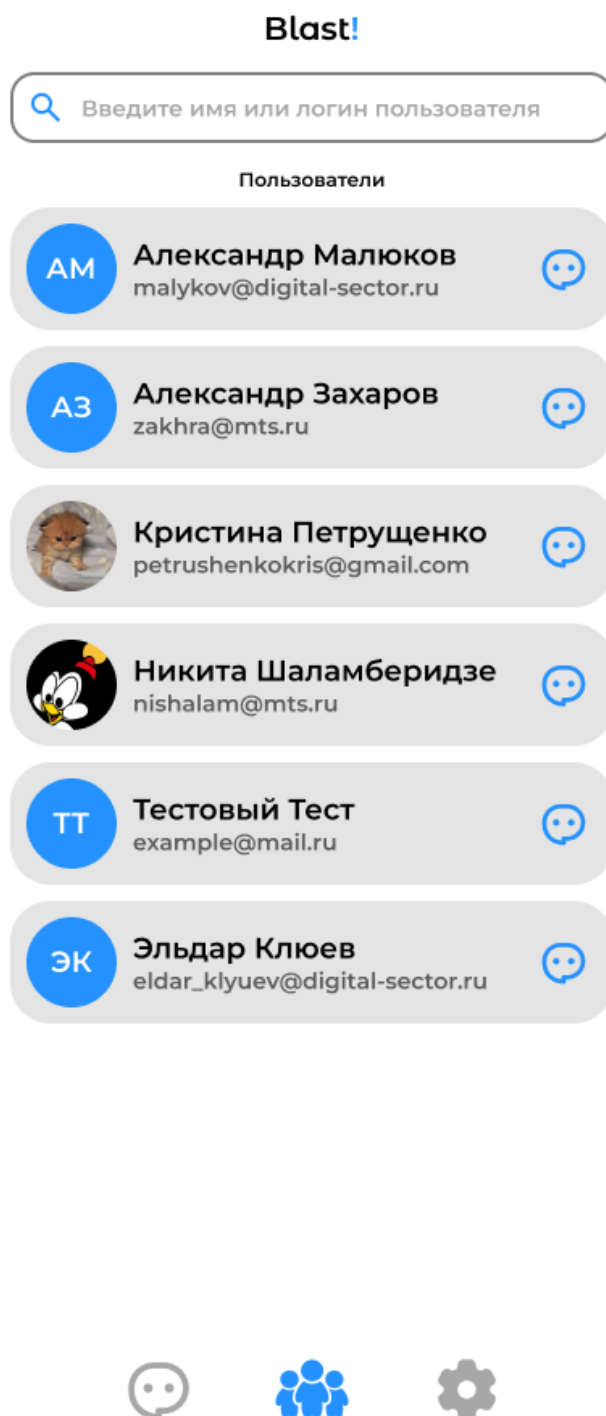


Рисунок 21 – Страница поиска пользователей

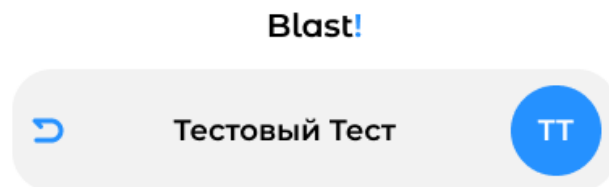
При нажатии на кнопку чата рядом с пользователем, клиента перебросит на страницу пустого чата, чтобы чат был создан необходимо, отправить

сообщение, после чего сервер получит сообщение и дополнительную информацию для создания нового чата рисунок 22.



Рисунок 22 – Пустой чат с пользователем

Ввод текста в поле ввода и нажатие на кнопку отправки создает сообщение с введенным текстом и отправляет его на сервер с помощью WebSocket-соединения рисунок 23.



Тестовое сообщение 14:31



Рисунок 23 – Первое сообщение в чате

Если пользователь перейдет на страницу команд (рисунок 20) при этом он не создавал или не был добавлен ни в одну команду его перебросит на пустую страницу с подсказкой рисунок 24.

Blast!

Чаты

Команды



Создавайте команды
чтобы оставаться на
связи с друзьями и
коллегами

СОЗДАТЬ









Рисунок 24 – Пустая страница команд

При нажатии на кнопку создать пользователя перебросит на страницу создания новой команды рисунок 25.

Blast!

Введите название команды...

Выберите пользователей которых хотите
добавить в команду

-  **Александр Малюков**
malykov@digital-sector.ru
-  **Александр Захаров**
zakhra@mts.ru
-  **Кристина Петрущенко**
petrushenkokris@gmail.com
-  **Никита Шаламберидзе**
nishalam@mts.ru
-  **Тестовый Тест**
example@mail.ru
-  **Эльдар Ключев**
eldar_klyuev@digital-sector.ru

СОЗДАТЬ КОМАНДУ





Рисунок 25 – страница создания команды

На странице создания команды есть возможность ввести название будущей команды, а также с помощью чекбоксов определить участников данной команды. После того как поле имени команды перестанет быть пустым, а количество выбранных участников команды будет больше 1, кнопка создания команды будет снова активна рисунок 26.

Blast!

Birthday Bot

Выберите пользователей которых хотите добавить в команду

AM	Александр Малюков malykov@digital-sector.ru	<input checked="" type="checkbox"/>
A3	Александр Захаров zakhra@mts.ru	<input type="checkbox"/>
	Кристина Петрущенко petrushenkokris@gmail.com	<input type="checkbox"/>
	Никита Шаламберидзе nishalam@mts.ru	<input type="checkbox"/>
ТТ	Тестовый Тест example@mail.ru	<input type="checkbox"/>
ЭК	Эльдар Ключев eldar_klyuev@digital-sector.ru	<input checked="" type="checkbox"/>

СОЗДАТЬ КОМАНДУ




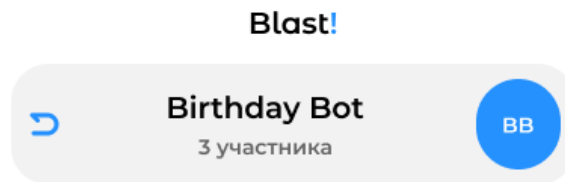
  

Рисунок 26 – Заполненная форма создания страницы

После нажатия на кнопку создать команду, пользователя перебросит в пустой чат команды с подсказкой как на рисунке 27.



Поприветствуйте участников команды, отправив первое сообщение в чат!



Рисунок 27 – Пустой чат команды с подсказкой

В командах каждое сообщения пользователя, кроме собственного, дополнительно отмечается именем отправителя, как показано на рисунке 28.

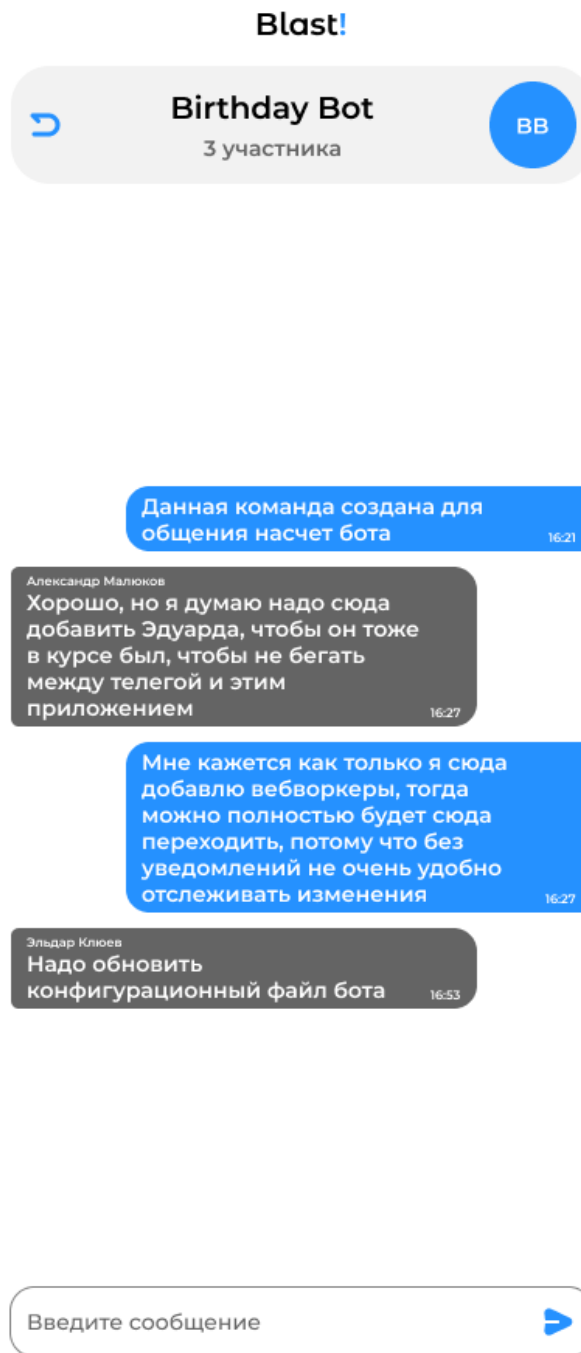


Рисунок 28 – Чат команды с сообщениями пользователей

При нажатии на кнопку стрелочки (кнопка возврата) пользователю будет выведен список текущих активных команд как на рисунке 29.

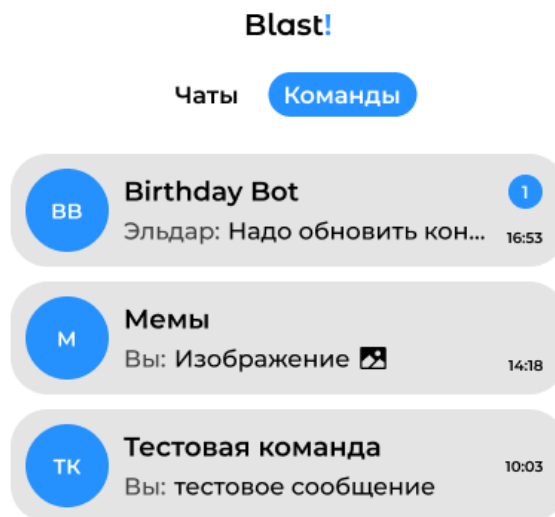


Рисунок 29 – Список активных командных чатов

При переходе обратно на страницу чатов, пользователю также будут выведен список активных чатов рисунок 30.

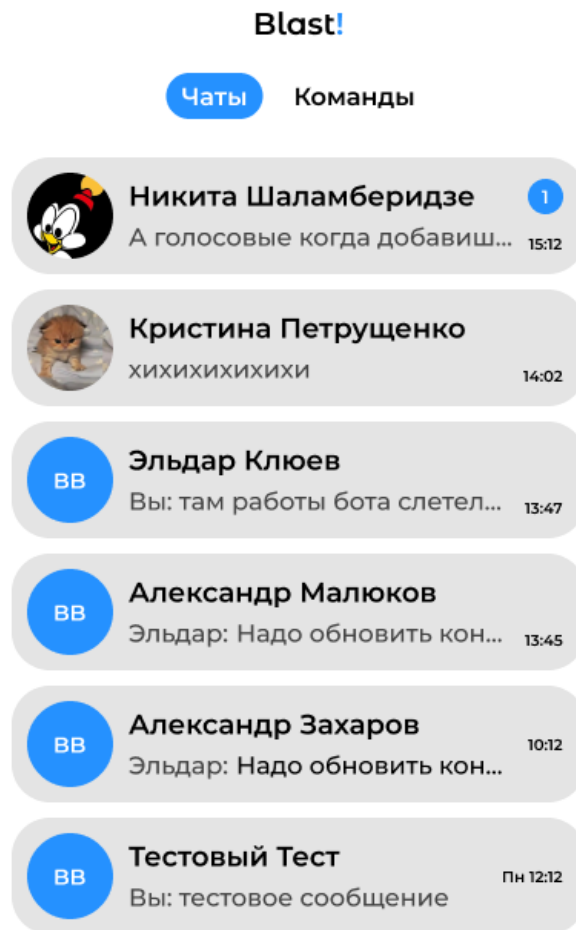


Рисунок 30 – Список активных чатов пользователя

Также у команды есть возможность настроить список участников, добавить или удалить некоторых, как показано на рисунке 31.

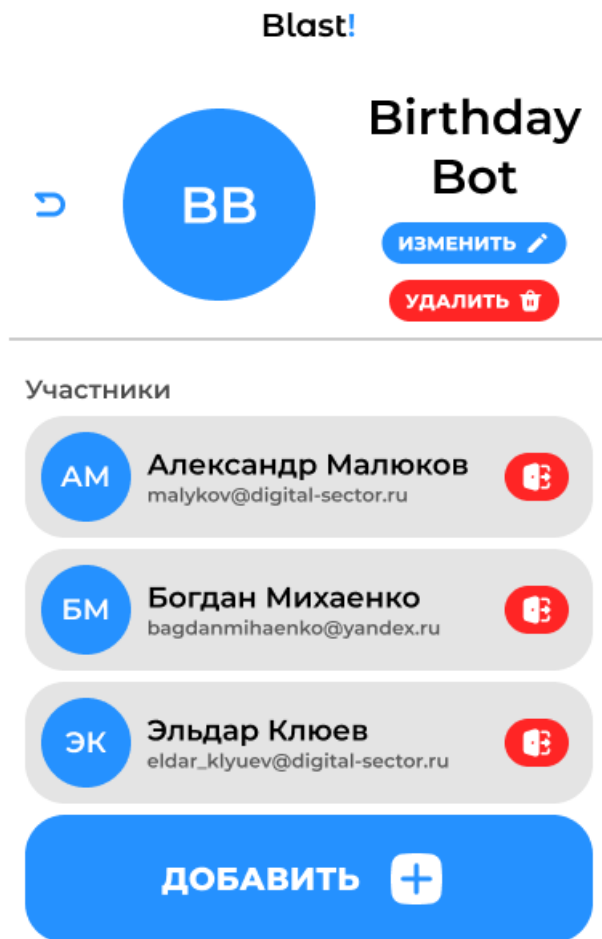


Рисунок 31 – Настройка команды

ЗАКЛЮЧЕНИЕ

В рамках выполнения данной работы была исследована история цифровизации образовательного процесса, а также проанализировано влияние цифровизации образования на учащихся с ограниченными возможностями.

Был проведен анализ существующих платформ для изучения жестового языка, а также спроектированы компоненты собственной образовательной платформы по изучению русскому языку жестов.

Была разработана платформа для изучения русского жестового языка онлайн, основными функциями которой являются система регистрации и авторизации, личный профиль, разработанная сурдопедагогами и сурдопереводчиками авторская программа обучения, возможность изучения алфавита и цифр, лекции по темам, словарь к каждой теме лекции, интерактивная игра, для закрепления пройденного материала.

В дальнейшем планируется расширение функционала, увеличение количества языков для обучения, добавление новых игр, лекций, словарей, добавление системы друзей и чата, а также возможности отслеживания своих результатов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Мартин Р.С.. Чистая архитектура / URL: https://codelibrary.info/download/1057_Chistaja_arhitektura.pdf (дата обращения: 03.04.2024).
2. Современная архитектура веб приложений / URL: <https://na-journal.ru/1-2024-informacionnye-tehnologii/7942-sovremennaya-arhitektura-veb-prilozhenii> (дата обращения: 05.04.2024)
3. Анализ современных алгоритмов шифрования данных / URL: <https://cyberleninka.ru/article/n/analiz-sovremennyh-algoritmov-shifrovaniya-dannyh> (дата обращения: 07.04.2024)
4. Docker документация. – Официальный сайт Docker. – URL: <https://www.docker.com/> (дата обращения: 16.04.2024).
5. Docker Compose документация. – Официальный сайт Docker. – URL: <https://docs.docker.com/compose/> (дата обращения: 15.04.2024).
6. Python документация. – Официальный сайт Python. – URL: <https://www.python.org/doc/> (дата обращения: 02.04.2024).
7. FastAPI документация. – Официальный сайт FastAPI. – URL: <https://fastapi.tiangolo.com/> (дата обращения: 04.04.2024).
8. Современный учебник JavaScript : сайт. – URL: <https://learn.javascript.ru/intro> (дата обращения: 21.04.2024).
9. Vue.js – Прогрессивный JavaScript фреймворк : сайт. – URL: <https://ru.vuejs.org> (дата обращения: 22.04.2024).