МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ

ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных технологий

Кафедра Информационных систем и технологий

Специальность 1-40 01 01 «Программное обеспечение информационных

технологий»

Специализация Программирование интернет-приложений

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**к курсовому проекту на тему:**

Веб-приложение для просчёта стоимости маршрута каршеринга

Выполнил студент Городилов Михаил Петрович

(Ф.И.О.)

Руководитель проекта асс. Кантарович В.С.

(учен. степень, звание, должность, подпись, Ф.И.О.)

Заведующий кафедрой к.т.н., доц. Смелов В.В.

(учен. степень, звание, должность, подпись, Ф.И.О.)

Консультанты асс. Кантарович В.С.

(учен. степень, звание, должность, подпись, Ф.И.О.)

Нормоконтролер асс. Кантарович В.С.

(учен. степень, звание, должность, подпись, Ф.И.О.)

Курсовой проект защищен с оценкой

Минск 2021

**Содержание**

[**Введение** 2](#_Toc90530521)

[**1 Постановка задачи** 2](#_Toc90530522)

[**1.1 Обзор аналогов** 2](#_Toc90530523)

[**2 Проектирование программного средства** 2](#_Toc90530524)

[**2.1 Архитектура** 2](#_Toc90530525)

[**2.2 Описание используемых технологий** 2](#_Toc90530526)

[**2.3 Серверная часть приложения** 2](#_Toc90530527)

[**2.3.1 Безопасность. Аутентификация и авторизация** 2](#_Toc90530528)

[**2.4 Взаимосвязь компонентов** 2](#_Toc90530529)

[**2.4.1 Логическое взаимодействие компонентов** 2](#_Toc90530530)

[**2.4.2 Физическое взаимодействие компонентов** 2](#_Toc90530531)

[**2.5 Диаграмма UML** 2](#_Toc90530532)

[**2.6 Сущности в базе данных и связи между ними** 2](#_Toc90530533)

[**3 Реализация приложения** 2](#_Toc90530534)

[**3.1 Серверная часть** 2](#_Toc90530535)

[**3.1.1 Конфигурация** 2](#_Toc90530536)

[**3.1.2 Реализация REST API. Контроллеры** 2](#_Toc90530537)

[**3.1.3 Сервисы.** 2](#_Toc90530538)

[**3.1.4 Репозитории. Взаимодействие с БД** 2](#_Toc90530539)

[**3.1.5 DTO. Конвертация разных типов объектов** 2](#_Toc90530540)

[**3.1.6 JWT. Аутентификация и авторизация** 2](#_Toc90530541)

[**3.1.7 Обмен сообщениями с клиентом в реальном времени** 2](#_Toc90530542)

[**3.2 Клиентская часть** 2](#_Toc90530543)

[**3.2.1 Хранение состояния** 2](#_Toc90530544)

[**3.2.2 Маршрутизация** 2](#_Toc90530545)

[**3.2.3 Обмен сообщениями с сервером в реальном времени.** 2](#_Toc90530546)

[**3.2.4 Видеосвязь в реальном времени.** 2](#_Toc90530547)

[**3.3 Развёртывание приложения в docker** 2](#_Toc90530548)

[**3.4 Непрерывная интеграция и непрерывная поставка** 2](#_Toc90530549)

[**4 Тестирование приложения** 2](#_Toc90530550)

[**5 Руководство пользователя** 2](#_Toc90530551)

[**5.1 Регистрация пользователя** 2](#_Toc90530552)

[**5.2 Аутентификация и авторизация пользователя** 2](#_Toc90530553)

[**5.3 Запуск конференции** 2](#_Toc90530554)

[**5.4 Страниц встреч** 2](#_Toc90530555)

[**Заключение** 2](#_Toc90530556)

[**Список используемых источников** 2](#_Toc90530557)

[**ПРИЛОЖЕНИЕ А** 2](#_Toc90530558)

[**ПРИЛОЖЕНИЕ Б** 2](#_Toc90530559)

[**ПРИЛОЖЕНИЕ В** 2](#_Toc90530560)

[**ПРИЛОЖЕНИЕ Г** 2](#_Toc90530561)

[**ПРИЛОЖЕНИЕ Д** 2](#_Toc90530562)

[**ПРИЛОЖЕНИЕ Е** 2](#_Toc90530563)

[**ПРИЛОЖЕНИЕ Ж** 2](#_Toc90530564)

# **Введение**

Важнейшей тенденцией в устойчивом развитии городской инфраструктуры является коллективное пользование автомобильным транспортом в форме общественного транспорта, такси, проката автомобилей, карпулинга и каршеринга.

Развитие каршеринга, наряду с другими видами аренды и механизмами совместного использования дорогостоящих активов, является ключевым трендом развития современных экономических отношений. Современные люди не желают нести издержки, связанные с владением, а аренда позволяет их избежать, но при этом сохраняет удобства и преимущества личного пользования требуемым имуществом. Ряд исследований показывает, что у жителей крупных мегаполисов с развитой системой каршеринга уменьшаются потребности во владении личным автомобилем.

В связи с бурным ростом индустрии возникает проблема в выборе оптимального автомобиля для поездки и проблема расчёта её стоимости. Для помощи в подборе ближайшего автомобиля, в Беларуси уже существуют определённые приложения, но они не решают вторую проблему и доступны только для мобильных устройств.

Таким образом, целью курсового проекта является создание адаптивного веб-приложения, в котором пользователи смогут подбирать автомобиль для поездки и вычислять её стоимость.

1. **Постановка задачи**

Главная задача курсового проекта является разработка адаптивного веб-приложения, позволяющего пользователю просматривать автомобили на карте, строить маршруты и получать их стоимость. В данном курсовом проекте приложение должно удовлетворять следующим требованиям:

* обеспечивать реализацию 3 ролей: гость, зарегистрированный пользователь и администратор;
* обеспечивать регистрацию гостя;
* предоставлять доступ к базе автомобилей (добавлять и менять цены тарифов, добавлять цены тарифов для новых автомобилей) администратору;
* обеспечивать авторизацию для администратора и зарегистрированного пользователя;
* позволять просматривать карту расположения автомобилей;
* позволять получать расчёт тарифов по маршруту;
* предоставлять возможность сохранения своего маршрута для авторизованного пользователя;
* предоставлять возможность сохранения фильтров автомобилей для авторизованного пользователя.

## **Обзор аналогов**

На рынке Беларуси присутствует один аналог приложния, позволяющий отображать автомобили всех компаний каршеринга на одной карте.

Carshare – Android и IOS приложение, которое позволяет отобразить машины всех белорусских каршерингов на одной карте. Его релиз состоялся в 2018 году. Количество скачиваний приложения в Google Play: 5 тысяч.

Интерфейс «Carshare» представлен на рисунке 1.1.

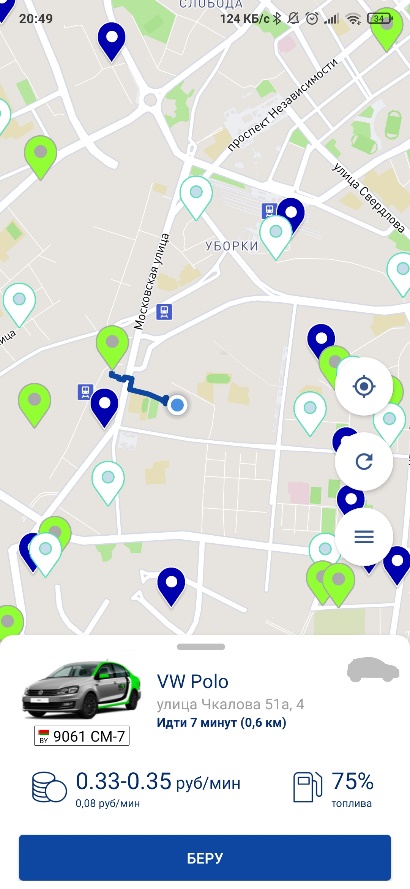


Рисунок 1.1 – Интерфейс «Carshare»

Проанализировав приложение, можно выделить основные плюсы и минусы данного программного средства:

Основные плюсы:

* стабильная работа карты;
* доступны фильтры по топливу и моделям;
* прокладывание маршрута до автомобиля.

Основные минусы:

* нет отображения загрузки автомобилей;
* машины могут появляться на карте не с первого раза;
* отсутствуют обновления для Android устройств, что в данный момент приводит к некорректному отображению фильтров;
* нет точных тарифов;

# **Проектирование программного средства**

Разработка архитектуры проекта – одна из важнейших задач в процессе работы над приложением, так как в зависимости от неё определяется уровень зависимости компонентами приложения, и насколько легко расширяемы её составные части.

Архитектура проекта – это его строение как оно видно (или должно быть видно) из вне его, т.е. представление программного средства как системы, состоящей из некоторой совокупности взаимодействующих подсистем. В качестве таких подсистем выступают обычно отдельные программы. Разработка архитектуры является первым этапом борьбы со сложностью программного средства, на котором реализуется принцип выделения относительно независимых компонент.

Основные задачи разработки архитектуры проекта:

Выделение программных подсистем и отображение на них внешних функций (заданных по внешнем описании) программного средства;

Определение способов взаимодействия между выделенными программными подсистемами.

С учетом принимаемых на этом этапе решений производится дальнейшая конкретизация и функциональных спецификаций.

## **Архитектура**

В основе будет использоваться микросервисная архитектура. Микросервисная архитектура – распространенный подход к разработке программного обеспечения, когда приложение разбивается на небольшие автономные компоненты (микросервисы) с четко определенными интерфейсами. Именно эта архитектура характерна для cloud-native приложений, которые сейчас популярны благодаря преимуществам, что открывают для бизнеса облачные среды.

Клиентская часть приложения является Multi-Page Application с поддержкой серверного рендеринга, то есть приложение, использует несколько HTML страниц и организует взаимодействие с пользователем через динамически подгружаемый HTML, CSS, JavaScript код, генерируемый на основе ответов, поступивших на AJAX запросы.

Серверная часть будет выполнять мероприятия по поддержке аутентификации и авторизации, а также будет ответственна за всё взаимодействие с базами данных.

## **Описание используемых технологий**

Проект реализован на основе существующего приложения CarShare, которое является агрегатором белорусских машин каршеринга. В частности, будет использован существующий веб-сервис, для получения списка доступных автомобилей.

Необходимый дополнительный функционал: сервис для просчёта тарифов, сервис для работы с данными пользователей, сервис-адаптер для получения реальных автомобилей, сервис аутентификации и авторизации, сервис конфигурации тарифов и клиентское веб-приложение. Для реализации данного функционала используются следующие технологии: ASP.NET Core, Mongo DB – для серверной части, и React c Next.js – для клиентской. В качестве провайдера карт выбран сервис maptiler.

ASP.NET Core – это новый общедоступный и кроссплатформенный фреймворк для создания современного облака приложений, связанных с подключением к интернету, таких как веб-приложения, приложения для интернета вещей и мобильных серверов. Приложения ASP.NET Core могут работать на .NET Core или на полной платформе .NET Framework. Этот фреймворк был спроектирован таким образом, чтобы обеспечить оптимизированную платформу разработки для приложений, которые перемещаются в облако или выполняются локально. Он состоит из модульных компонентов с минимальной перегрузкой, поэтому вы сохраняете гибкость при построении своих решений. Существует возможность разрабатывать и запускать кроссплатформенные ASP.NET Core приложения на Windows, Mac и Linux. Фреймворк ASP.NET Core общедоступен на GitHub.

MongoDB – система управления базами данных, которая работает с документоориентированной моделью данных. В отличие от реляционных СУБД, MongoDB не требуются таблицы, схемы или отдельный язык запросов. Информация хранится в виде документов либо коллекций.

React – это JavaScript библиотека GUI (англ) с открытым исходным кодом, сосредоточенная на одной конкретной цели – эффективном выполнение задач в рамках разработки пользовательского интерфейса. Его можно отнести к категории “V” в архитектурном шаблоне MVC (модель-вид-контроллер).

Next.js – это фреймворк, основанный на React, который позволяет создавать веб-приложения с улучшенной производительностью и улучшенным пользовательским опытом с помощью дополнительных функций предварительного рендеринга, таких как полноценный рендеринг на стороне сервера (SSR) и статическая генерация страниц (SSG).

## **Серверная часть приложения**

При проектировании классов и интерфейсов для серверной части приложения необходимо стараться держать низкий уровень избыточности и связности между составляющими частями, а также усиленно следить за корректностью кода, поскольку от этого будет зависеть работа всех подключенных клиентов. Добиться всего этого можно с помощью грамотного распределения логики сервера по разным уровням, определяя формат объектов, которые будут использоваться на каждом уровне, и преобразования объектов при передаче с уровня на уровень, а также используя подходящие паттерны проектирования.

Для того, чтобы обмениваться данными между клиентами и сервисами был выбран архитектурный стиль REST. По своему определению он широко использует возможности протокола HTTP и поэтому не имеет состояния (stateless). Это современный подход, использующийся в большинстве веб-сервисов и поэтому имеющий множество примеров реализаций различных функций на его основе, таких как аутентификация, выполнение CRUD-операций и т.д. В REST-приложениях в подавляющем большинстве случаев используется JSON для передачи данных, его также было решено взять как формат данных при передаче между клиентами и сервером.

### **Безопасность. Аутентификация и авторизация**

Важнейшим аспектом проектирования серверной части приложения является то, как пользователи будут входить в систему и как система будет понимать, что очередной присланный запрос отправлен уже находящимся в системе пользователем.

Стандартными подходами при реализации этих функций является либо аутентификация с помощью логина и пароля, при этом используя собственную БД в качестве их места хранения, либо использование сторонних сервисов, предоставляющих своё API для того, чтобы можно было осуществить вход. Это может быть API на основе протокола OAuth, либо какие-то собственные разработки. При рассмотрении данного вопроса лучше всего обратить внимание на то, какими будут приложение-клиенты, то, в каком окружении, на каких системах они будут запускаться. В нашем случае это веб-клиент и сами сервисы. В связи с этим было принято решение об применении технологии Identity Server.

Применение Identity Server позволит отделить операцию аутентификации от используемых сервисов и вынести её в общее место. Сервисы будут делать запрос на IS, чтобы удостоверится в подлинности токена.

## **Взаимосвязь компонентов**

### **Логическое взаимодействие компонентов**

В решении курсового проекта была использована REST архитектура. Рассмотрим компоненты, используемые в данной архитектуре.

Клиент – это программа, использующая API. Клиент делает запросы к API, чтобы получить некоторую информацию или что-то изменить в приложении. Веб-браузер является клиентом – он взаимодействует с API-интерфейсом, чтобы получить от него содержимое страницы. Запрошенная информация отправляется обратно в браузер и отображается на экране.

Ресурс – это любая информация, которую API может предоставить клиенту. Например, ресурсом в данном API может быть пользователь, фотография или товар. Каждый ресурс имеет уникальное имя, называемое идентификатором ресурса.

Сервер используется приложением, которое принимает запросы клиентов и содержит ресурсы, которые требуются клиенту. Сервер имеет API для взаимодействия с клиентами, не предоставляя им прямого доступа к контенту, хранящемуся в базе данных.

Так же была использована микросервисная архитектура. В ходе развития информационных технологий требования по производительности и доступности к информационным системам, как вновь разрабатываемым, так и уже существующим, постоянно росли. Для того чтобы системы соответствовали таким постоянно растущим требованиям, происходило наращивание вычислительных мощностей, то есть применялось так называемое вертикальное масштабирование. Однако эффективность вертикального масштабирования оказалась весьма ограниченной, так как прирост вычислительных мощностей конкретного сервера не давал необходимого прироста производительности информационной системы, что повлекло за собой появление горизонтального масштабирования.

Горизонтальное масштабирование осуществляется не за счет увеличения вычислительных мощностей конкретного сервера, а за счет добавления новых узлов в систему.

Микросервисная архитектура – это архитектура, которая учитывает возможность горизонтального масштабирования еще на этапе проектирования системы.

Архитектурный стиль микросервисов – это подход, при котором единое приложение строится как набор небольших сервисов, каждый из которых работает в собственном процессе и коммуницирует с остальными, используя легковесные механизмы, как правило, HTTP.

### **Физическое взаимодействие компонентов**

Работа всего приложения будет осуществляться на серверах Microsoft Azure. Компоненты клиентской и серверной части будут располагаться в докер контейнерах.

#### **Docker**

В основе работы Docker лежит стандартизированный способ исполнения кода. Docker – это операционная система для контейнеров. Подобно тому как виртуальная машина создает виртуальное представление аппаратного обеспечения сервера (то есть устраняет необходимость непосредственно управлять таковым), контейнеры создают виртуальное представление серверной операционной системы. После установки на каждый сервер Docker предоставляет доступ к простым командам, необходимым для сборки, запуска или остановки контейнеров. Схема основных компонентов Docker показана на рисунке.

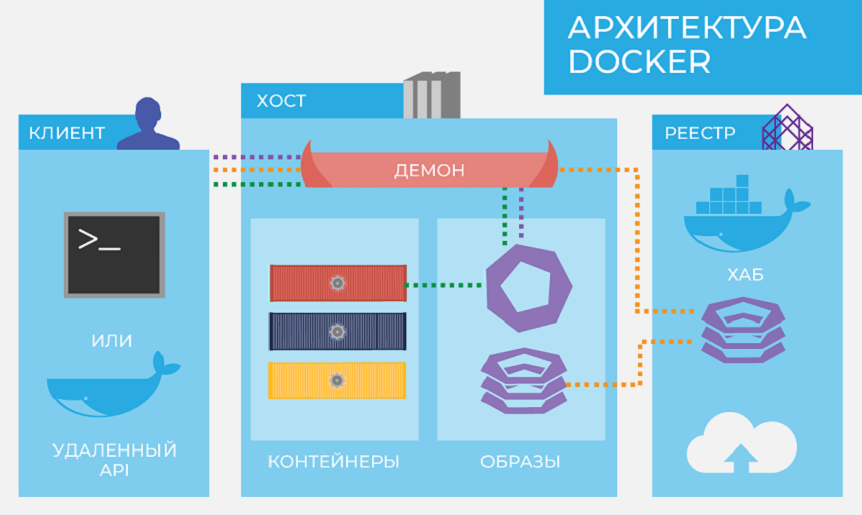


Рисунок 2.2 – Схема основных элементов Docker

Использование Docker позволяет быстрее и эффективнее доставлять или перемещать код, стандартизирует выполняемые приложениями операции и в целом экономит средства, оптимизируя использование ресурсов. Благодаря Docker пользователи получают объект, который с высокой надежностью можно запускать на любой платформе. Простой и понятный синтаксис Docker обеспечивает полный контроль над выполняемыми операциями. Повсеместное внедрение контейнеров подразумевает доступ к разнообразным инструментам и готовым приложениям, которые можно использовать с Docker.

Контейнеры Docker можно использовать в качестве основных компонентов для создания современных платформ и приложений. Docker упрощает сборку и запуск распределенных микросервисных архитектур, развертывание кода с помощью стандартизированных конвейеров непрерывной интеграции и доставки, создание высокомасштабируемых систем обработки данных и полностью управляемых платформ для разработчиков.

## **Диаграммы UML**

ДиаграммаUML–это графическое представление набора элементов,изображаемое в виде связанного графа с вершинами (сущностями) и ребрами (отношениями).

В языке UML вариант использования изображается в виде овала, помеченного именем представляемого варианта. Варианты использования могут быть связаны с участвующими в них действующими лицами (actors), изображаемыми в виде человечков и представляющими различные роли пользователей системы или внешние системы, взаимодействующие с ней.

Варианты использования могут быть связаны друг с другом тремя видами связей: обобщением (generalization), расширением (extend relationship) и включением (include relationship). Действующие лица также могут быть связаны друг с другом с помощью связей обобщения (generalization).

Диаграмму вариантов использования проекта можно увидеть в приложении А.

В системе определены роли администратора, пользователя и гостя. Пользователь имеет доступ к тому же функционалу что и гость, администратор к тому же что и пользователь.

Прецеденты системы:

* управление базой автомобилей;
* изменение профилей пользователей;
* сохранение фильтров;
* cохранение маршрута;
* просмотр карты автомобилей;
* вход в систему;
* выполнение регистрации;
* фильтрация автомобилей;
* просчёт маршрута;
* выбор автомобиля, включается в просчёт маршрута;
* построение маршрута, включается в просчёт маршрута.

Ниже будет описана диаграмма развёртывания/классов, её можно увидеть в приложении Б.

Для реализации авторизации в системе используется внешний компонент MS Identity. Так же приложение основано на платформе ASP.NET Core, и использует компонент ASP.NET Core MVC.

Пакет Identity Server содержит классы расширяющие базовые классы Identity. Также MongoIdentityUser агрегирует MongoIdentityRole (многие ко многим). В под пакете Static Pages перечислены страницы используемые для аутентификации. AuthenticateController агрегирует SignInManager из внешнего пакета Identity и расширяет ControllerBase из ASP .Net Core MVC.

На диаграмме также представлены пакеты баз данных, клиентского интерфейса и приложения обработки клиентской информации. Клиентский интерфейс и приложение обработки информации имеют зависимость по отношению к пакету Identity Server.

## **Сущности в базе данных и связи между ними**

Изучение предметной области и составленный впоследствии функционал показали, что можно выделить 5 сущностей.

Ниже приведены структуры объектов для хранения данных информационной системы.

БД UserData с данными пользователей имеет две коллекции: Filters и Routes.

Поля объектов Route:

* Id, идентификатор пользователя;
* Waypoints, массив точек маршрута.

Поля объектов Filter:

* Id, идентификатор пользователя;
* ProviderIds, массив идентификаторов провайдеров.

БД Configuration с конфигурацией приложения имеет одну коллекцию Tariffs.

Поля объектов Tariff:

* Id, идентификатор;
* Provider, провайдер каршеринга (состоит из Id, Name, LogoUrl);
* CarPrices. массив машин с ценами (состоит из Model, Brand, PerMinCost, PerMinParkingCost).

БД Identity с данными аутентификации, имеет две коллекции: Users и Roles.

Объекты User и Role используют GUID в качестве Id и являются расширениями объектов ASP.NET Core Identity.

# **Реализация приложения**

Приложение делится на две части сервер и клиент. Клиентская и серверная части располагаются на облачной платформе Azure, также используется облачный кластер MongoDB.

## **Серверная часть**

Для разработки серверной части приложения использовалась платформа ASP.NET Core. Серверная часть состоит из 5 микросервисов.

Типовая структура сервиса продемонстрирована на рисунке 3.1.

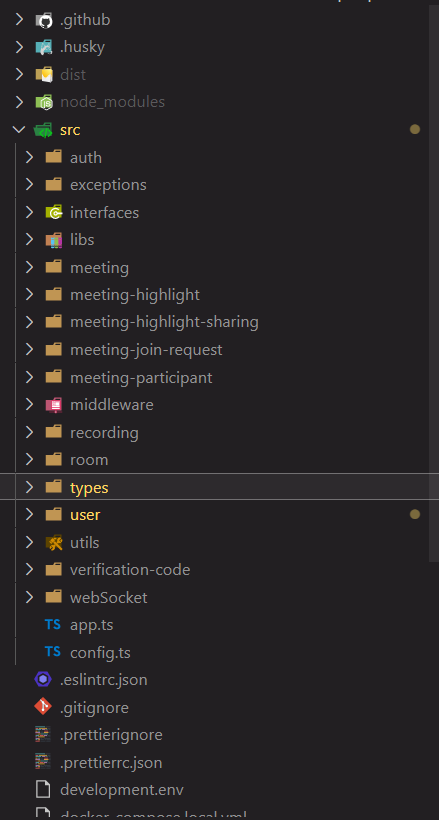


Рисунок 3.1 – Структура серверной части

Директория «node\_modules» хранит необходимые библиотеки для работы приложения. Директория «src» хранит необходимую логику для работы приложения. Директория «middleware» хранит файлы с middleware-функциями. Директория «services» содержит в себе файлы, которые используют для обработки http-методов модели и представления и отправляют в ответ клиенту некоторый результат. Директория «types» хранит файлы для определения типов, констант приложения. Директория «utils» хранит вспомогательные функции, использующиеся в проекте. Остальные папки — это совокупности файлов (модули), отражающих доменные объекты. Структура модуля «meeting» изображена на рисунке 3.2

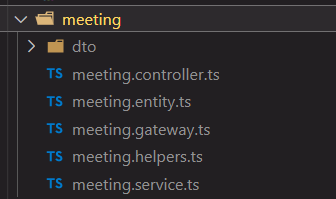


Рисунок 3.2 – Структура модуля «meeting»

Файлы с суффиксом controller – отвечают за обработку HTTP запросов. Файлы с суффиксом service – отвечают бизнес логикой и общаются с уровнем данных через репозитории предоставляемые typeorm. Директория dto отвечает за формат приходящих данных. Файлы, оканчивающиеся на entity отображают описание модели с которой работает typeorm. Файлы с суффиксом gateway отвечают за работу с сообщениями приходящими по протоколу WebScoket.

### **Конфигурация**

Секретные данные для работы приложения, такие как строка подключения к базе данных или секрет для JWT-токена, содержатся в .env файлах, которые в зависимости от переданной переменной среды NODE\_ENV выбираются с помощью npm-пакета cross-env. Env файл отвечает за то, в каком режиме будет работать сервер: production или development.

В файле tsconfig.json описана конфигурация Typescript.

Файлы eslint и prettier отвечают за настройку контроля качества кода которого необходимо придерживаться во время разработки программного продукта.

В файле app.ts вызывющая функцию по настройке и запуску сервера. В нем создается объект htpp сервера к которому затем присоединяется websocket сервер. Задаются контроллеры и помимо всего прочего, там настраивается CORS для корректного доступа с веб-клиента и устанавливается глобальный префикс.

Отдельно стоит уделить внимание форматам даты и времени, которые будут использоваться. Это важный момент, так как очевидно, что тип, представляющий время и дату, является сложным, а значит, что на разных платформах, в разных языках программирования, этот тип может быть интерпретирован совершенно по-разному, поэтому возникает необходимость введения определённого единого стиля. Таковым был выбран стандарт ISO 8601, так как он является самым распространённым и имеет поддержку во всех используемых языках программирования. Он выглядит так: «YYYY-MM-DDThh:mm:ss».

### **Реализация REST API. Контроллеры**

Классы-контроллеры являются представлением API. Есть условный шаблон, по которому описывается каждый контроллер. Каждый контроллер должен реализовывать интерфейс, показанный на рисунке 3.3

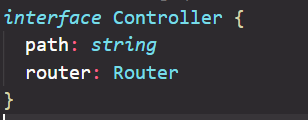


Рисунок 3.3 – Интерфейс контроллера

Поле path отвечает за url по которому данный контроллер будет вызван. Поле Router является объектов из пакета Express и отвечает за сопоставление HTPP запроса определенной функции.

Каждый контроллер также должен помечаться декоратором Service для обеспечения Dependency Injection за которое отвечает пакет typedi.

Для валидации тела запроса был разработан отдельный validation middleware с использованием пакета class-validator. Код данного обработчика представлен в листинге 3.1

|  |
| --- |
| function validationMiddleware(    type: any,    skipMissingProperties = false,    source: 'body' | 'query' = 'body',  ): RequestHandler {    return (req, res, next) => {      validate(plainToClass(type, req[source]), {        skipMissingProperties,      }).then((errors: ValidationError[]) => {        if (errors.length > 0) {          const message = errors            .map((error: ValidationError) => {              let errorConstrain: any = error.constraints              while (!errorConstrain && error.children) {                error = error.children[0]                errorConstrain = error.constraints              }              return Object.values(errorConstrain)            })            .join(', ')          next(new HttpException(400, message))        } else {          next()        }      })    }  } |

Листинг 3.1 – Реализация validation middleware

Пример использования PUT обработчика представлен в Листинге 3.2

|  |
| --- |
| this.router.put(        `${this.path}/change-password`,        validationMiddleware(ChangePasswordDto),        asyncWrapper(this.changePassword),      ) |

Листинг 3.2 – Обработчик PUT запроса

Благодаря этому обработчику, нет необходимости в том, чтобы осуществлять самостоятельную валидацию, данных, приходящих в виде функцию. Достаточно указать в аргументах метода, класс DTO в котором описать с помощью декораторов ограничения на входные данные. И middleware проверит входные данные и в случае если входные данные окажутся не валидными отправит ответ со статусом ошибки 400 и пояснительным текстом. В контроллеры с помощью механизма Dependency Injection библиотеки typedi встраиваются сервисы, которые в нем используются. Пример кода контроллера приведен в приложении В.

### **Сервисы.**

Сервисы – классы, которым контроллеры делегируют выполнение различных действий, после поступления на них запроса. Они также помечаются своим особым декоратором для того, чтобы приложение смогло создать по ним объекты и включить их в свой пул для последующего внедрения зависимостей.

Как было уже описано ранее, большинство сервисов лишь выполняют различные действия с репозиториями, иногда лишь делая небольшие преобразования в объектах, чтобы ORM-система смогла правильно их обработать. Работа сервисов совершается в неблокирующем асинхронном режиме. В конструктор сервиса так же встраиваются различные модели.

### **Репозитории. Взаимодействие с БД**

Для того, чтобы взаимодействовать с базой данных, используется ORM typeorm. Для этого создаются Модели – классы, представляющие собой таблицы в базе данных, у которых поля соответствуют колонкам этих таблиц. Эти классы принимают на вход методы репозиториев. Репозиторий в нашем случае – это сами классы моделей, которые позволяют работать с ними как с репозиториями.

Каждый класс модели помечается декоратором Entity в котором можно указать название таблицы, которой он будет соответствовать.

Описание полей класса также сопровождается декораторами в которых указывается: тип данных, ограничение целостности, ограничения типа данных.

### **DTO. Конвертация разных типов объектов**

DTO – важнейшая часть API-системы, так как эти объекты представляют то, как должны будут выглядеть запросы и ответы к ней. Легко понять тот факт, что если запросы будут по большей части одинаковыми, то ответы могут отличаться как по своему наполнению, так и по структуре. Например, если приходит запрос на получение одного пользователя, то не имеет смысла отправлять вместе с ним полную информацию о всех аукционах, в которых он поучаствовал, так как большинство из отправленных данных не будет использоваться получателем и канал передачи будет впустую нагружаться. Также будут возникать проблемы как с многочисленными зависимостями, если они будут тянуть друг друга, так и с циклическими зависимостям, которые будут вызывать переполнение стека. Одним из решений перечисленных проблем является определение класса для каждого из типов запросов. Главным его недостатком является то, что придётся писать достаточно большое количество мало чем отличающегося кода. Тем не менее, преимуществ такой подход несёт больше, поэтому был выбран именно он. Для реализации этого подхода использовались:

* классы для каждого тела запроса;
* классы, которые наследовали либо все поля либо часть из них.
* классы для тела ответа

Пример такого класса можно увидеть в приложении Г.

При валидации используется npm-пакет class-validator. Для преобразования ответа модели в DTO используется пакет class-transform.

### **JWT. Аутентификация и авторизация**

Как было написано ранее, для нужд аутентификации и авторизации будет использоваться JWT-токен. При аутентификации, он будет формироваться и отправляться аутентифицирующемуся клиенту, затем этот клиент должен будет его отправлять с каждым запросом, а задача сервера – проверять этот токен на валидность.

Для реализации аутентификации использовался пакет jsonwebtoken. При успешной авторизации сервер формирует два jwt токена access token и refresh token. Refresh token используется для обновления access token и его самого, чтобы пользователю не пришлось вводить свои данные заново.

### **Обмен сообщениями с клиентом в реальном времени**

 Для обмена сообщениями между браузером и веб-сервером в режиме реального времени используется протокол связи поверх TCP-соединения - Websocket.

Для работы с веб-сокетами был разработан класс, WebSocketServer. В конструктор принимает http-сервер и инитиализирует сервер по маршруту /api/websockets. В данном проекте использовалась библиотека Socket.IO. После инициализации сервера, происходит подписка на событие подключения сокета.

При подключении сокета срабатывает функция connection в которой сокет подписывается на события определенные в перечислении WebSocoketEvent. Перечисление WebSocoketEvent представлено на рисунке 3.4

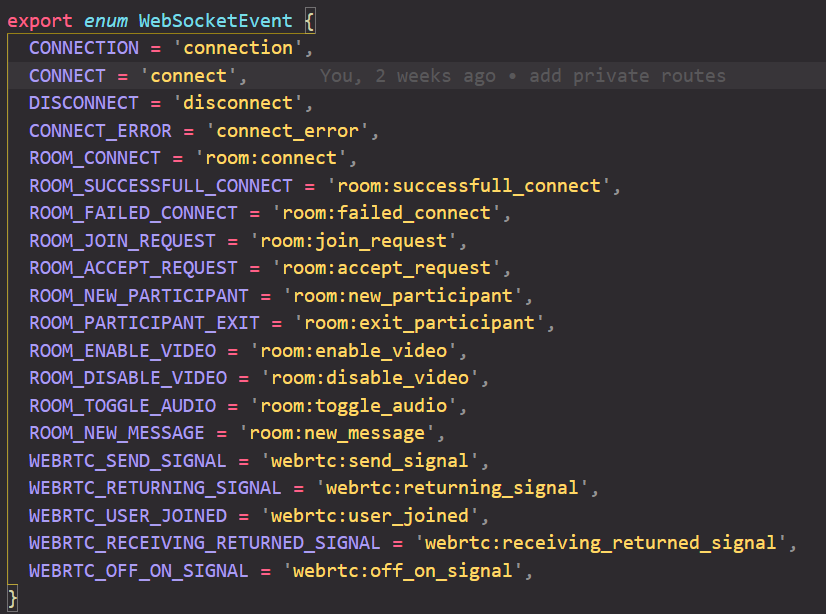


Рисунок 3.4 – Перечисление WebSocoketEvent

Обмен сообщениями в реальном времени используется для подключения пользователь в конференции, об оповещении о новых участниках встречи. Для обмена сообщениями в рамках комнаты.

Также реализация данного сервера используется в качестве сигнального сервера для работы технологии WebRTC.

## **Клиентская часть**

Для клиентской части приложения был использован фреймворк Gatsby.Js. [Gatsby.js](https://www.gatsbyjs.org/) - это [SSG](https://blog.cackle.me/static-site-generators-headless-cms-v-jamstack/), который позволяет построить или перестроить [pwa сайт](https://blog.cackle.me/progressive-web-app-pwa/) с очень быстрой загрузкой страниц. Загрузка страниц целиком и переход между ними измеряется в миллисекундах, а не секундах.

Данный фреймворк построен на JAMStack архитектуре: является SSG (Static Site Generator) и следующих современных технологиях, включающих React, Webpack, GraphQL, современный JavaScript. С помощью React создавать интерактивные пользовательские интерфейсы приятно и просто. Достаточно описать, как части интерфейса приложения выглядят в разных состояниях. React будет своевременно их обновлять, когда данные изменяются.

Декларативные представления сделают код более предсказуемым и упростят отладку. А создание инкапсулированных компоненты с собственным состоянием, позволяет объединить их в сложные пользовательские интерфейсы.

Поскольку логика компонента написана на JavaScript, а не содержится в шаблонах, можно с лёгкостью передавать самые разные данные по всему приложению и держать состояние вне DOM.

### **Хранение состояния**

MobX — это автономная библиотека, для управления фронтенд-состоянием приложения. MobX обеспечивает консистентность и согласованность внутреннего состояния фронтенд-приложения, предоставляя удобные инструменты для его изменения.

MobX позволяет сделать невозможным инконсистентность состояния. Стратегия достижения этого довольно проста: убедится, что, все что может быть вынуто из состояния, будет вынуто. Автоматически.

Для хранения состояния создаются соответствующие хранилища. Все свойства хранилища при обертывании объекта в фукнцию makeAutoObservable становятся отслеживаемыми библиотекой MobX. По умолчанию все поля класса становятся отслеживаемыми, а методы имеют типо actions. Для того чтобы подписать React компонент на изменения состояния необходимо обернуть его в функцию observer. Данные изменились – в компоненте возникла реакция. Таким образом, никаких специальных подписок и коллбэков – Mobx доставляет изменения сам.

Для того чтобы иметь доступ к данным располагаемым в хранилищах MobX был разработан специальный rootStore. Используя createContext функции библиотеки React передаем в нее экземпляр созданного главного хранилища. Затем для доступа к этому хранилищу внутри компонентов был разработан специальный React хук useStores. Для того, чтобы основное управляющее хранилище rootStore со старта имело доступ ко всем остальным данным, делаем инициализацию дочерних хранилищ внутри rootStore. Код отвечающий за объявление rootStore, а также создание React хука useStores доступно в приложении Д.

Реализация roomChatStore представлена на листинге 3.3

|  |
| --- |
| import { makeAutoObservable, runInAction } from 'mobx'  import { RootStore } from '.'  interface ChatMessage {    message: string    firstName: string    lastName: string    imageUrl: string    isMine: boolean,    date?: Date  }  class RoomChatStore {    isSetingsOpen = false    rootStore: RootStore    state = 'pending'    message: string = ''    messages: Array<ChatMessage> = []    constructor(rootStore: RootStore) {      this.rootStore = rootStore      makeAutoObservable(this)    }    newMessage(message: ChatMessage) {      runInAction(() => this.messages.push({...message, date: new Date()}))    }    sendMessage() {      this.rootStore.meetingStore.sendMessage(this.message)      this.message = ''    }    setMessage(message: string) {      this.message = message    }  }  export default RoomChatStore |

Листинг 3.3 – Реализация roomChatStore

В данном фрагменте можно увидеть вызов других хранилищ. А так же методы по изменению текущих свойств.

### **Маршрутизация**

Gatsby работает как SSR, и каждый маршрут соответствует локальному файлу.

Это означает, что при использовании маршрутизации на стороне клиента Gatsby будет искать файл (созданный в папке src/pages/\* или динамически созданный в gatsby-node.js), а ненайденный файл приведет к 404.

Чтобы реализовать маршрутизацию на стороне клиента, нужно указать, что определенный путь является только клиентским. Для этого используется встроенный в Gatsby роутер @reach/router. Пример страницы встречи.

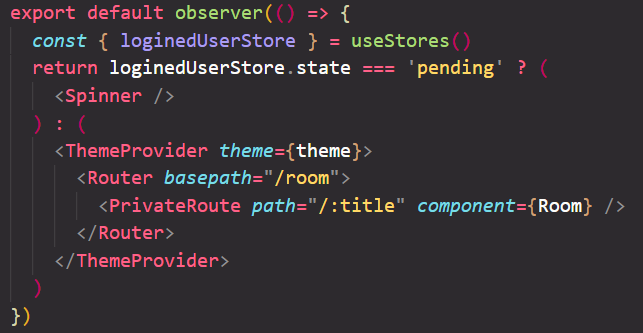


Рисунок 3.5 – Маршруты страницы комнаты

Чтобы попасть на страницу конференции необходимо быть авторизованным пользователем.

### **Обмен сообщениями с сервером в реальном времени.**

Для реализации двунаправленного соединения с сервером в режиме реального времени используется протокол связи поверх TCP-соединения - Websocket. Для работы с протоколом, как и на сервере была использована библиотека, упрощающая работу с сокетами socket-io-client. С нее было реализовано получение информации о участниках конференции, чат конференции, а также управление входящими запросами на подключение в комнату.

### **Видеосвязь в реальном времени.**

Для обеспечения онлайн видеосвязи используется в приложении используется протокол WebRTC.

WebRTC, сокращение от Web Real-Time Communication, является одновременно API и протоколом. Протокол WebRTC - это набор правил для двух агентов WebRTC для двунаправленной безопасной связи в реальном времени. API WebRTC позволяет разработчикам использовать протокол WebRTC. WebRTC API специфицирован только для JavaScript.

Аналогичные отношения были бы между HTTP и Fetch API. Протокол WebRTC - это HTTP, а API WebRTC - это Fetch API.

Групповая конференция строится как mesh сеть, где каждый новый узел соединяется с каждым. Для работы с протоколом используется библиотека упрощающая подключение между узлами simple-peer. Для установки соединения были разработаны функции, позволяющие создавать узел и передавать необходимую для соединения двух узлов информацию через сигнальный сервер, в качестве которого выступает серверная часть приложения. Код функций для установки подключения представлен на листинге 3.4

|  |
| --- |
| export function createPeer(socket: Socket, participant: MeetingParticipant, isCamDisabled, isMicDisabled) {    const peer = new Peer({      ...configuration,      initiator: true,      stream: participant.currentStream,    })    peer.on('signal', (signal) => {      socket.emit(WebSocketEvent.WEBRTC\_SEND\_SIGNAL, {        toSocketId: participant.socketId,        signal,        isCamDisabled,        isMicDisabled,      })    })    return peer  }  export function addPeer(socket: Socket, incomingSignal, participant: MeetingParticipant, isCamDisabled, isMicDisabled) {    const peer = new Peer({      ...configuration,      initiator: false,      stream: participant.currentStream,    })    peer.on('signal', (signal) => {      socket.emit(WebSocketEvent.WEBRTC\_RETURNING\_SIGNAL, {        signal,        toSocketId: participant.socketId,        isCamDisabled,        isMicDisabled,      })    })    peer.signal(incomingSignal)    return peer  } |

Листинг 3.4 – Функции для установки соединения

## **Развёртывание приложения в docker**

Docker может автоматически создавать образы читая инструкции из Dockerfile. Dockerfile представляет из себя текстовый документ содержащий все команды для сборки образа.

Для докерезации клиентского приложения нужно добавить конфигурацию для nginx. NGINX, является известным программным обеспечением с открытым исходным кодом для веб-сервера. В своём первоначальном выпуске оно функционировало для веб-обслуживания HTTP. Однако сегодня оно также служит обратным прокси-сервером, балансировщиком нагрузки HTTP и почтовым прокси-сервером. Конфигурация для nginx показана на листинге 3.5.

|  |
| --- |
| http {    upstream docker-app-dev { server meetqi\_app\_dev:80;  }    upstream docker-api-dev { server meetqi\_api\_dev:3016;  }    server {      listen 80;      listen 443 ssl;      server\_name dev.meetqi.com;      ssl\_certificate /etc/letsencrypt/live/dev.meetqi.com/fullchain.pem;      ssl\_certificate\_key /etc/letsencrypt/live/dev.meetqi.com/privkey.pem;      include /etc/letsencrypt/options-ssl-nginx.conf;      ssl\_dhparam /etc/letsencrypt/ssl-dhparams.pem;      location /.well-known/acme-challenge/ { root /var/www/certbot; }      location /api {          proxy\_http\_version 1.1;          proxy\_cache\_bypass $http\_upgrade;          proxy\_set\_header Upgrade $http\_upgrade;          proxy\_set\_header Connection 'upgrade';          proxy\_set\_header Host $host;          proxy\_set\_header X-Real-IP $remote\_addr;          proxy\_set\_header X-Forwarded-For proxy\_add\_x\_forwarded\_for;         proxy\_pass http://docker-api-dev;              }      location / {          proxy\_http\_version 1.1;          proxy\_cache\_bypass $http\_upgrade;          proxy\_set\_header Upgrade $http\_upgrade;          proxy\_set\_header Connection 'upgrade';          proxy\_set\_header Host $host;          proxy\_set\_header X-Real-IP $remote\_addr;          proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;          proxy\_pass http://docker-app-dev;      }    }  } |

Листинг 3.4 – Конфигурация nginx

Dockerfile, разработанный для клиентской части приложения, описан в листинге 3.5.

|  |
| --- |
| FROM node:16-alpine as builder  RUN \    apk add --no-cache python3 make g++ && \    apk add vips-dev fftw-dev --update-cache \    --repository http://dl-3.alpinelinux.org/alpine/edge/community \    --repository http://dl-3.alpinelinux.org/alpine/edge/main \   && rm -fR /var/cache/apk/\*  RUN apk add automake autoconf libtool nasm  WORKDIR /app  COPY package.json .  RUN yarn install && yarn cache clean  COPY . .  RUN ["yarn", "build"]  FROM nginx as prod  EXPOSE 80  COPY --from=builder /app/nginx/default.conf /etc/nginx/conf.d/default.conf  COPY --from=builder /app/public /usr/share/nginx/html |

Листинг 3.5 – Dockerfile клиентской части приложения

Dockerfile, разработанный для серверной части приложения, описан в приложении E.

Далее нужно создать image клиентского и серверного приложений для этого в консоли нужно прописать команду, приведённую в листинге 3.6, находясь в папке с определённым проектом.

|  |
| --- |
| docker build -t container\_name path\_to\_docker\_file |

Листинг 3.6 – Команда для сборки image

Docker Compose инструмент для создания и запуска многоконтейнерных Docker приложений. В Compose, вы используете специальный файл для конфигурирования ваших сервисов приложения. Конфигурация Docker Compose расположена в файле docker-compose.yml. Она представлена в листинге 3.7.

|  |
| --- |
| version: "3"  services:    meetqi\_postgres\_dev:      container\_name: meetqi\_postgres\_dev      image: postgres:13      ports:        - "5432:5432"      volumes:        - /data/postgres:/data/postgres      env\_file:        - docker.env      networks:        - shared-network    nginx:      container\_name: nginx-dev      image: nginx:latest      depends\_on: [        meetqi\_api\_dev, meetqi\_app\_dev      ]      volumes:        - ./nginx/default.conf:/etc/nginx/nginx.conf      ports:        - 8080:80      command: /bin/sh -c "nginx -g 'daemon off;'"      networks:        - shared-network    meetqi\_api\_dev:      container\_name: meetqi\_api\_dev      image: ivashyn/meetqi-backend:latest      depends\_on:        - meetqi\_postgres\_dev      networks:        - shared-network      ports:        - 3016:3016      restart: always    meetqi\_app\_dev:      container\_name: meetqi\_app\_dev      image: local-meetqi      ports:        - 5000:80      restart: unless-stopped      command: /bin/sh -c "nginx -g 'daemon off;'"      networks:        - shared-network  networks:    shared-network: |

Листинг 3.7 – Docker-compose.yml файл

Команда запуска контейнеров представлена в листинге 3.8.

|  |
| --- |
| Docker-compose up |

Листинг 3.8 – Команда запуска контейнеров

После этого приложение станет доступно для использования.

## **Непрерывная интеграция и непрерывная поставка**

GitHub Actions настроены в качестве системы непрерывной интеграции для серверной и клиентской части. Для этого в репозитории проета был создан файл .github/workflows/developmentCI.yaml. Фрагмент файла по авторизации в DockerHub, сборки и загрузки образа в облачное хранилище образов DockerHub представлен на листинге 3.9.

|  |
| --- |
| - name: Cache Docker layers                uses: actions/cache@v2                with:                    path: /tmp/.buildx-cache                    key: ${{ runner.os }}-buildx-${{ github.sha }}                    restore-keys: |                        ${{ runner.os }}-buildx-  - name: Login to Docker Hub                uses: docker/login-action@v1                with:                    username: ${{ secrets.DOCKER\_USERNAME }}                    password: ${{ secrets.DOCKER\_PAT }}  - name: Output                run: echo ${{ steps.prep.outputs.tags }} && echo ${{ steps.buildx.outputs.name }} && echo ${{ steps.buildx.outputs }}  - name: Build and push                id: docker\_build                uses: docker/build-push-action@v2                with:                    builder: ${{ steps.buildx.outputs.name }}                    context: ./                    file: ./Dockerfile                    target: prod                    push: true                    tags: ${{ steps.prep.outputs.tags }}                    cache-from: type=local,src=/tmp/.buildx-cache                    cache-to: type=local,dest=/tmp/.buildx-cache |

Листинг 3.9 – Docker-compose.yml файл

Непрерывная доставка осуществляется с помощью другого GitHub Action код которого представлен в приложении Ж.

# **Тестирование приложения**

Тестирование проекта выполнялось в течение разработки посредством таких средств, как отладка и логгирование. После введения нового либо изменения существующего компонента программы, он и зависящий от него код также подвергался тестированию. Анализ полученных результатов происходил по факту получения ошибок и проблемные места сразу исправлялись. После завершения разработки программы по нескольку раз были подвергнуты испытаниям все элементы управления в пользовательском интерфейсе, также они тестировались по ходу написания записки.

После написания приложения также проводились различные тесты, и проверки, такие как проверка на подключение в комнату, проверка на корректный чат, проверка корректного создания комнаты и добавления новой пометки, а также изменение профиля.

Валидация приложения – это один из основных гарантов надёжности приложения и всякий программист должен предусмотреть и предотвратить непредвиденное поведение пользователя.

При регистрации обрабатываются вводимые данные пользователя. Так как поле «email» является уникальным, в базе данных не могут храниться пользователи с одинаковыми логинами, если в базе данных уже существует пользователь с таким же логином, то клиента уведомят об ошибке, это продемонстрировано на рисунке 4.1.

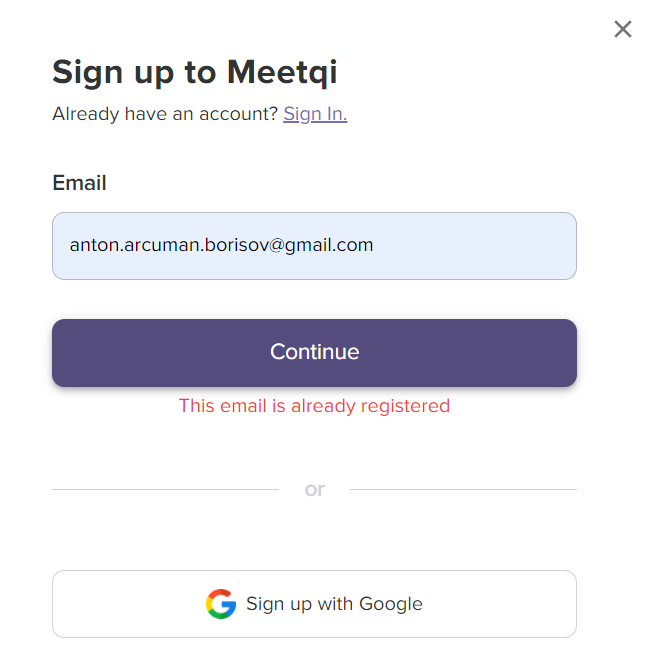


Рисунок 4.1 – Некорректная регистрация

При входе в систему также обрабатываются вводимые данные пользователя. Если пользователь неверно ввел данные в поля для входа, он получит уведомление о некорректно введенных данных, пример уведомления об ошибке предоставлен на рисунке 4.2.

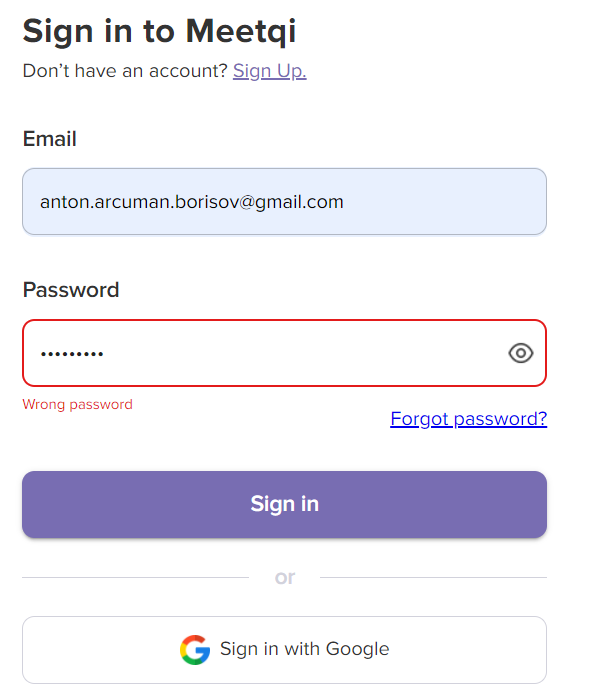
****

Рисунок 4.2 – Неверно введены данные

В итоге приложение стало хорошо отлаженным и готовым к полноценному использованию.

# **Руководство пользователя**

Данная глава содержит описание некоторых функций приложения для более легкого восприятия конечного пользователя.

## **Регистрация пользователя**

Для регистрации необходимо пройти несколько этапов. Первый этап это заполнить email, форма продемонстрирована на рисунке 5.1.

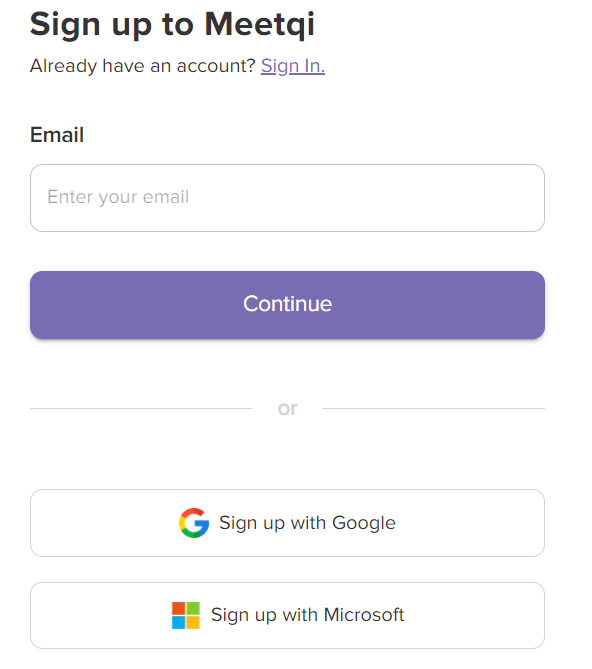


Рисунок 5.1 – Форма регистрации

После отправки email пользователю необходимо подтвердить свою почту путем перехода по ссылке из сообщений отправленого на адрес электронной почты. Пример письма продемонстрирована на рисунке 5.2.

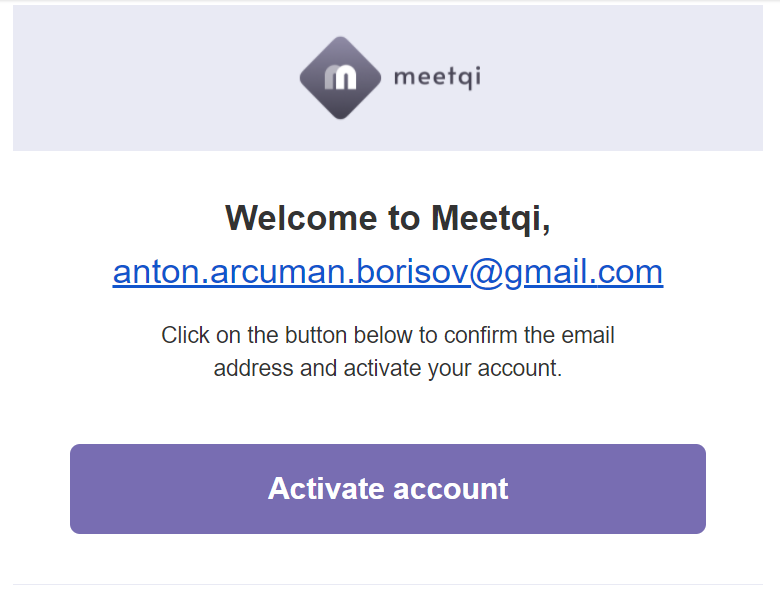


Рисунок 5.2 – Письмо пришедшее на почту

Следующим этапом будет ввод личной информации, а также пароля. Форма представлена на рисунке 5.3

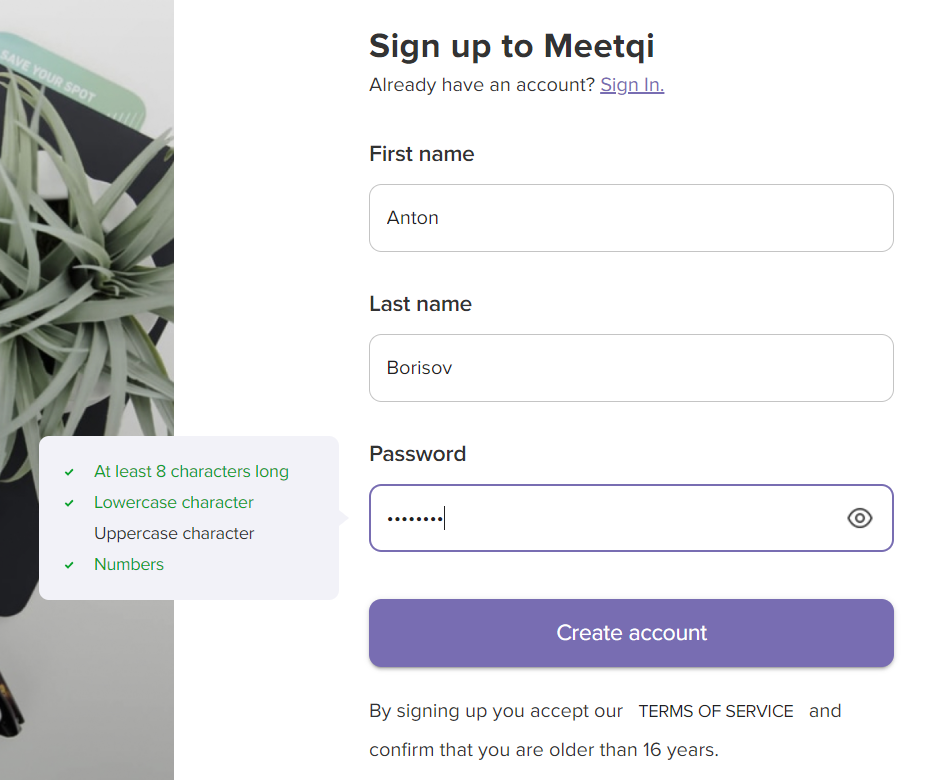


Рисунок 5.3 – Письмо пришедшее на почту

После отправки пользователем запроса на регистрацию, происходит валидация всех полей формы. Если валидация будет неуспешной, пользователь увидит сообщения об ошибках в соответствующих полях для ввода информации. В случае успешной регистрации пользователь переадресовывается на главную страницу приложения.

## **Аутентификация и авторизация пользователя**

Для аутентификации необходимо ввести пароль, логин и нажать на кнопку «Sign In», что позволит проверить, существует ли пользователь с введенными данными в системе. Все поля являются обязательными для заполнения.

Если аутентификация прошла успешно и соблюдены все выше описанные правила, то происходит перенаправление на главную страницу, что означает, что пользователь был успешно авторизован. Скриншот формы авторизации представлен на рисунке 5.2.

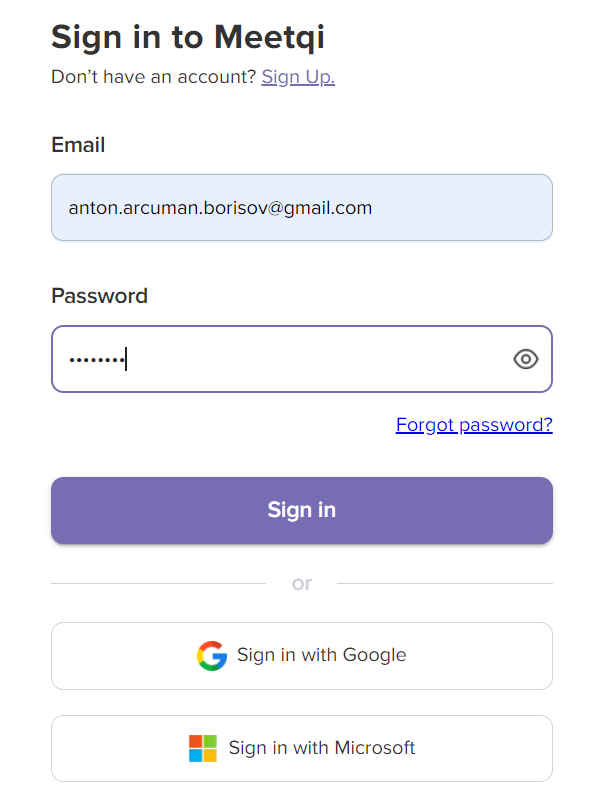


Рисунок 5.2 – Фрагмент экрана с формой для авторизации пользователя

При входе в систему авторизированному пользователю выдается токен, который записывается в локальное хранилище, тем самым позволяя разграничить возможности пользователей на сайте.

В данном курсовом проекте логика приложения задумана так, что пользователь, который не вошел в систему имеет доступ только к стартовой странице и форме регистрации и авторизации

## **Запуск конференции**

Для того, чтобы пользователь смог запустить конференцию ему необходимо вначале создать комнату(рисунок 5.4).

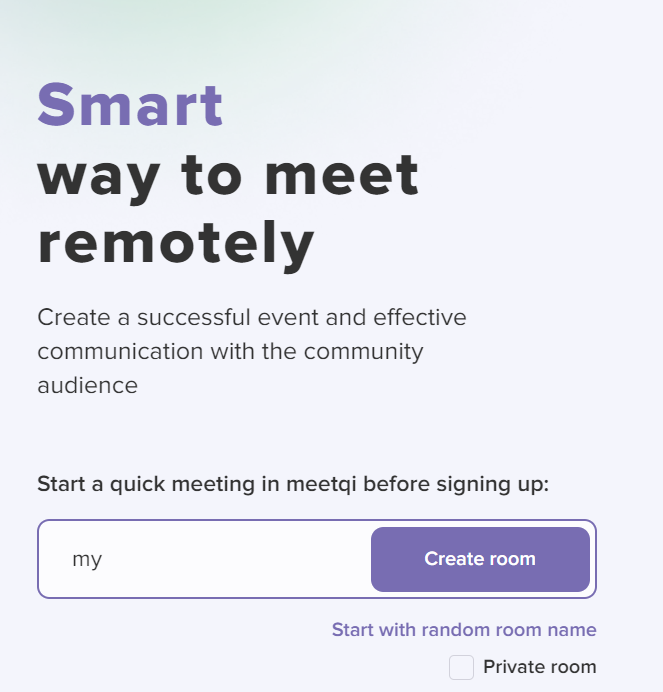


Рисунок 5.4 – Создание новой комнаты

После того как пользователь создал комнату, автоматически создастся встреча и он будет перенаправленный на экран подготовки митинга (рисунок 5.5).

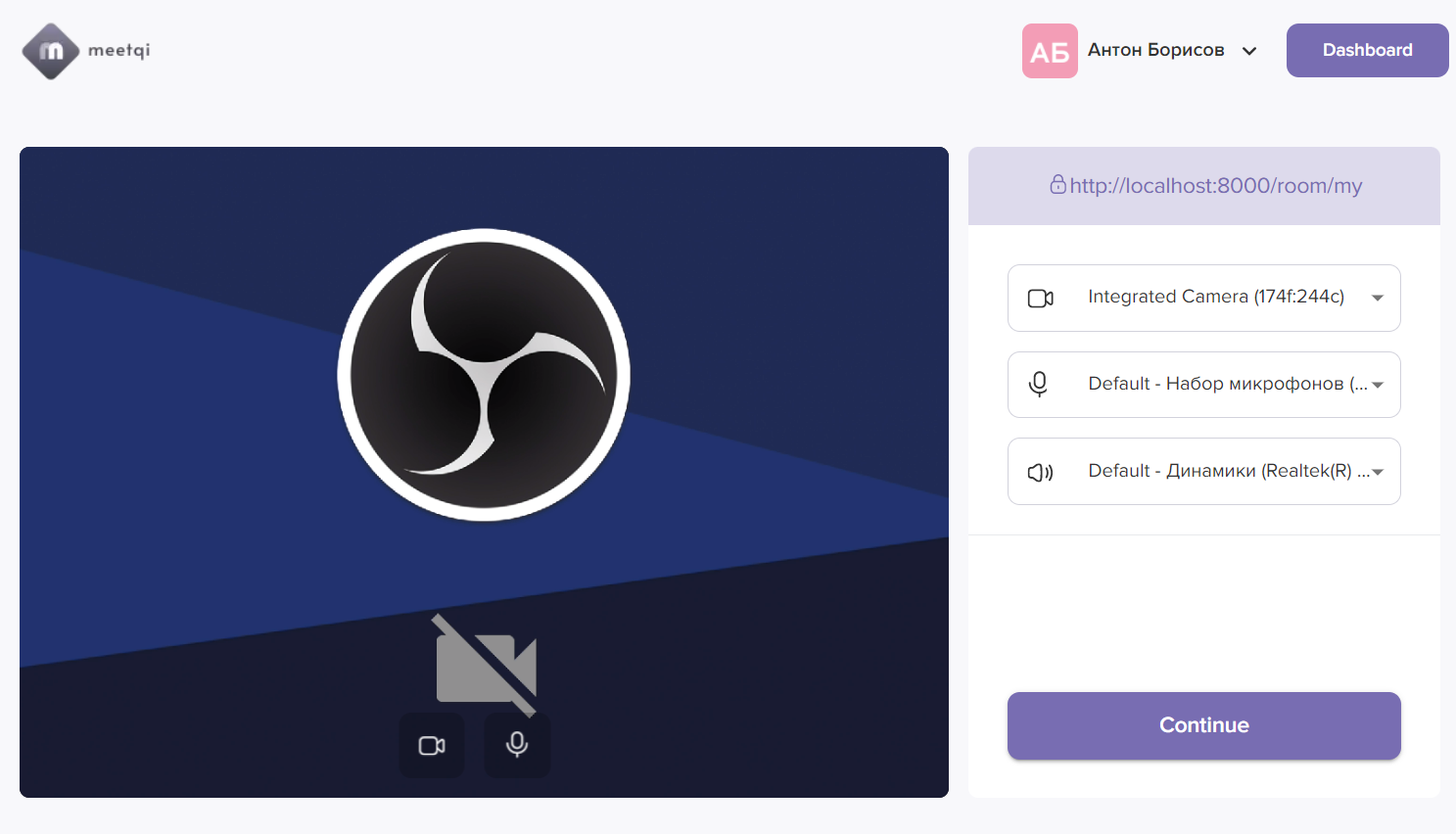


Рисунок 5.5 – Комната ожидания

На данном этапе пользователь может выбрать используемую камеру и микрофон. При нажатии кнопки «Continue» пользователь попадает на страницу встречи к которой могут подключиться также другие пользователи, которые отображаются в панели «Members» (рисунок 5.6).

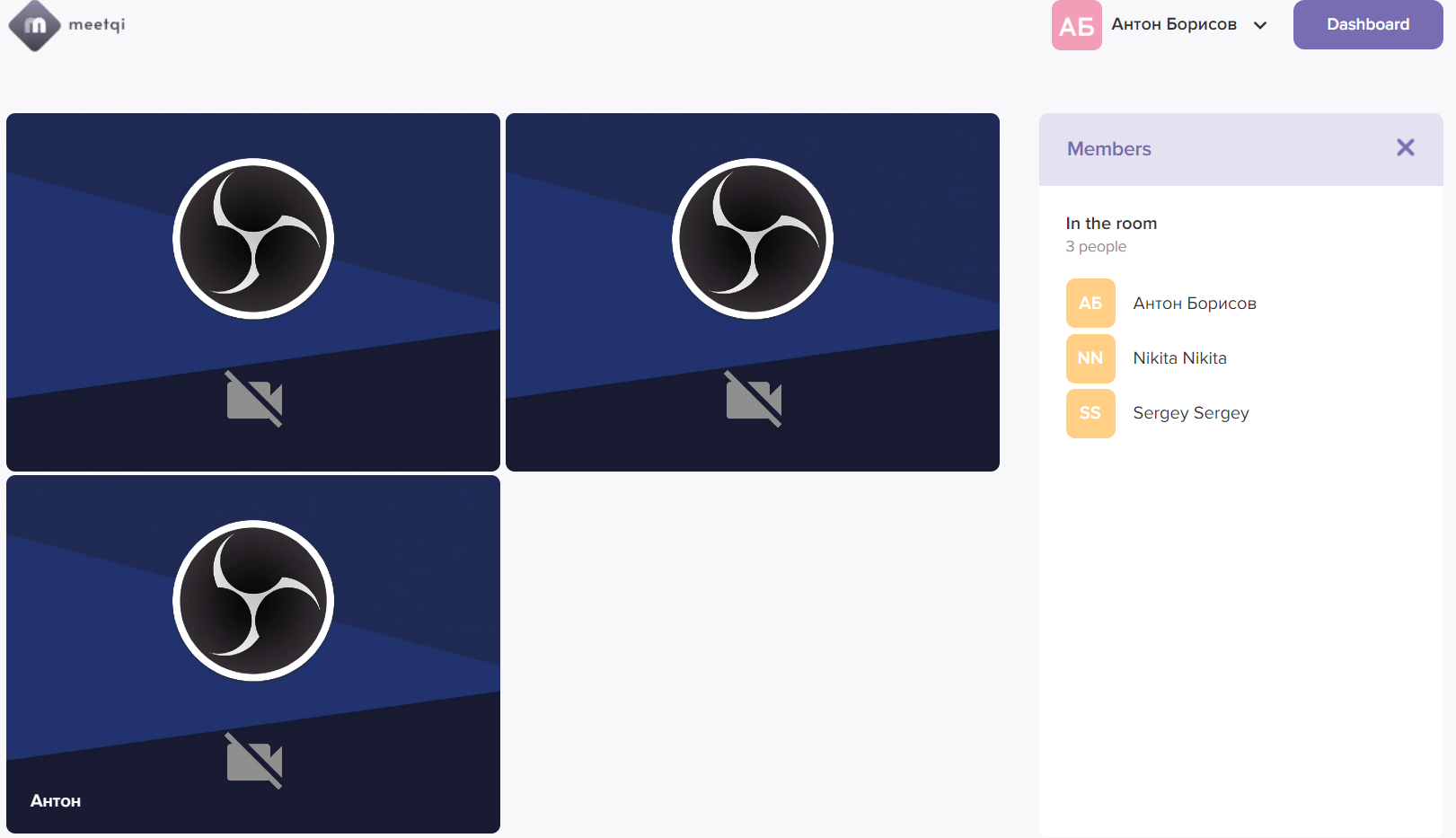


Рисунок 5.6 – Страница конференции с включенной вкладкой «Members»

Также у пользователя есть функционал чата с другими собеседниками. Интерфейс чата представлен на рисунке 5.7.

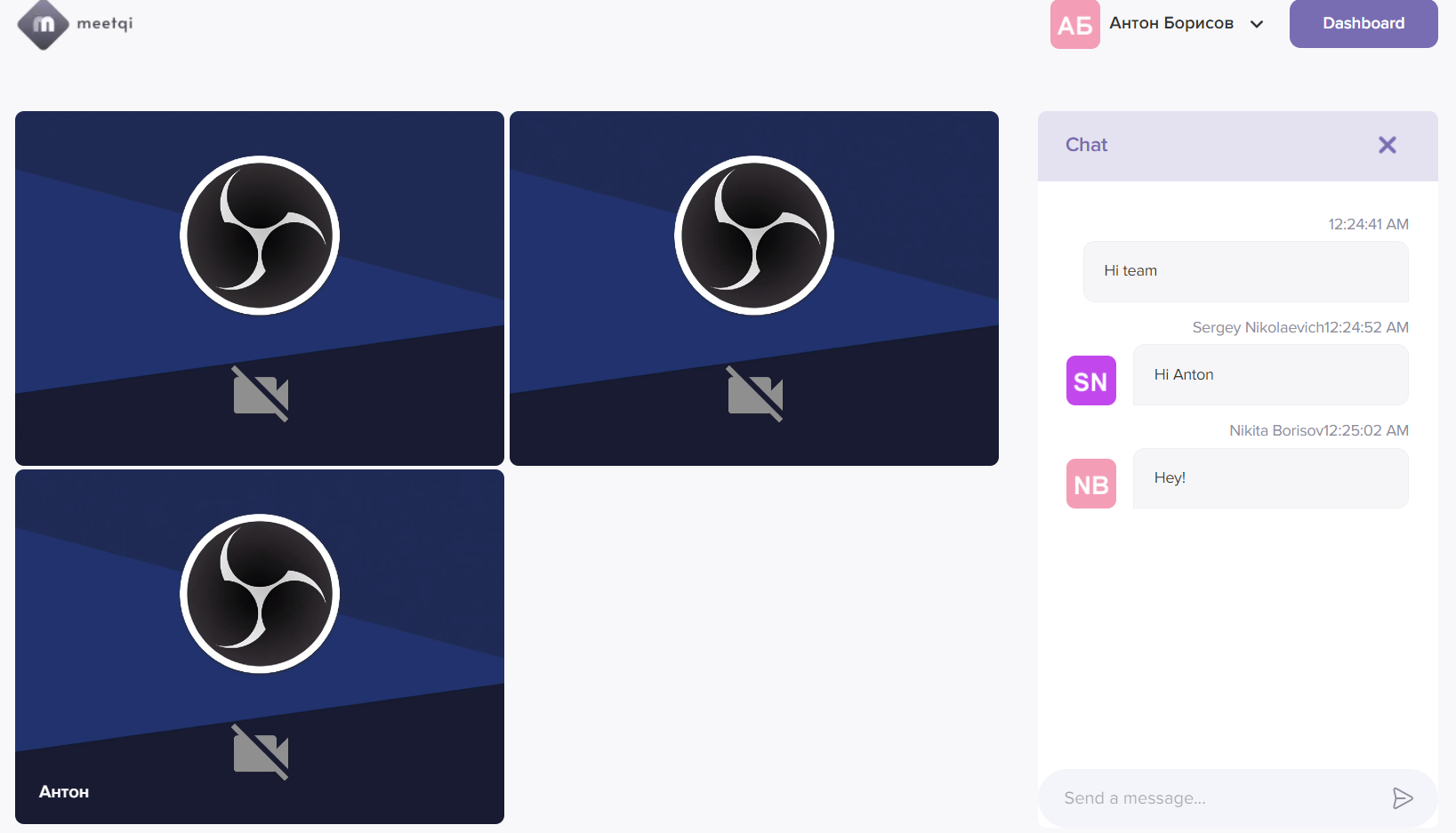


Рисунок 5.7 – Страница конференции с включенной вкладкой «Chat»

Также у пользователя есть возможность добавлять заметки и копировать ссылку чата, а также включать и выключать демонстрацию экрана

## **Страниц встреч**

На странице встреч пользователь может просмотреть все митинги в которых он участвовал, а также участников этих встреч и свои заметки. Пример описания встречи приведен на рисунке 5.8

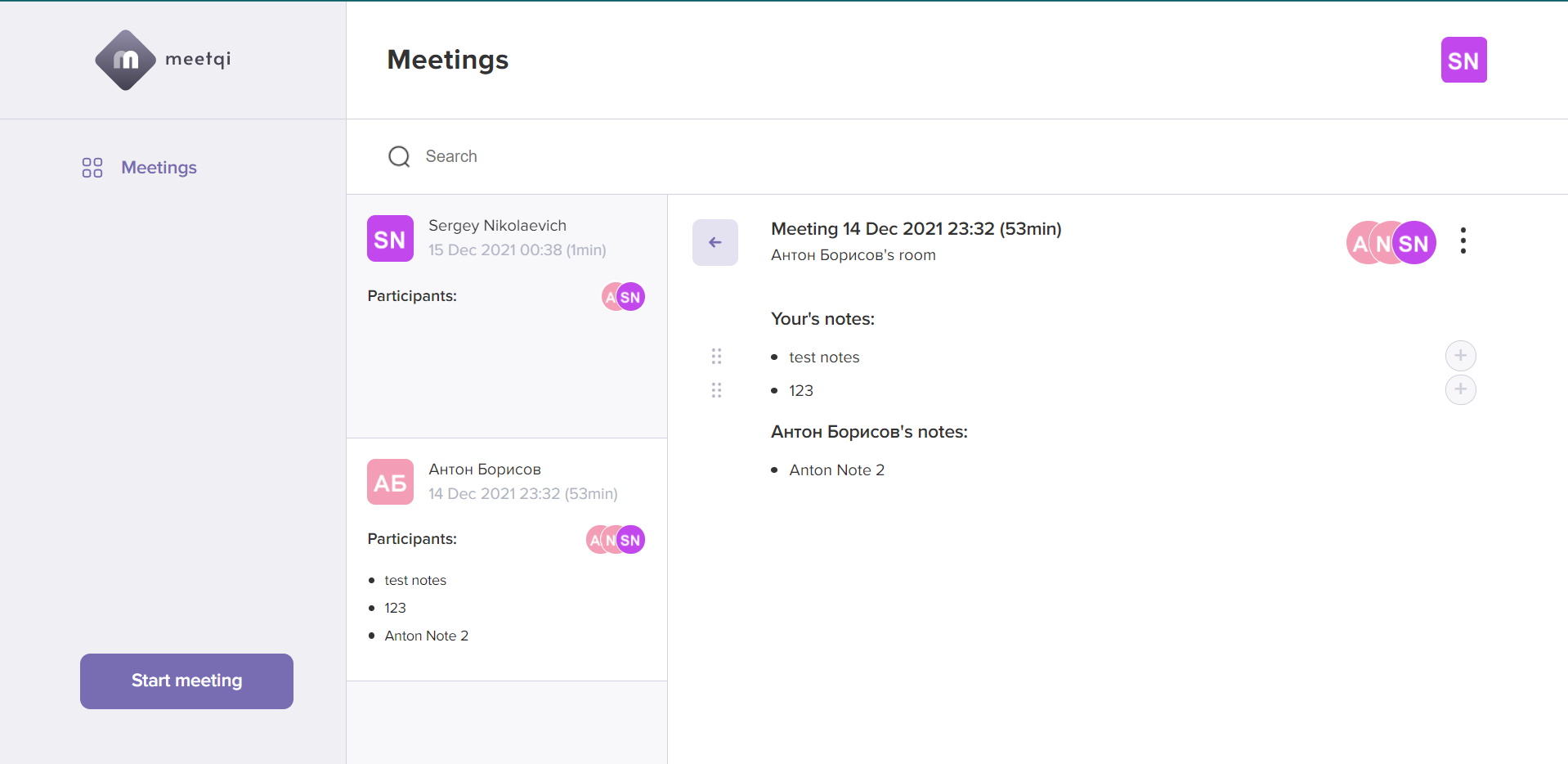


Рисунок 5.8 – Описание страницы встреч

При выборе определенной встречи, можно увидеть свои заметки, а также заметки которыми поделились с пользователем другие участники встречи.

# **Заключение**

При работе над курсовым проектом было проведено изучение предметной области, выявление в ней проблем и составление функционала программного средства, с их учётом, после чего последовало проектирование и разработка. Также в ходе работы проанализировано и проработано множество сторон и нюансов проектирования и разработки серверной части приложения и клиентской, например, выбор подходящих паттернов и технологий, составление и тестирование алгоритмов, структурирование проекта, построение пользовательского интерфейса.

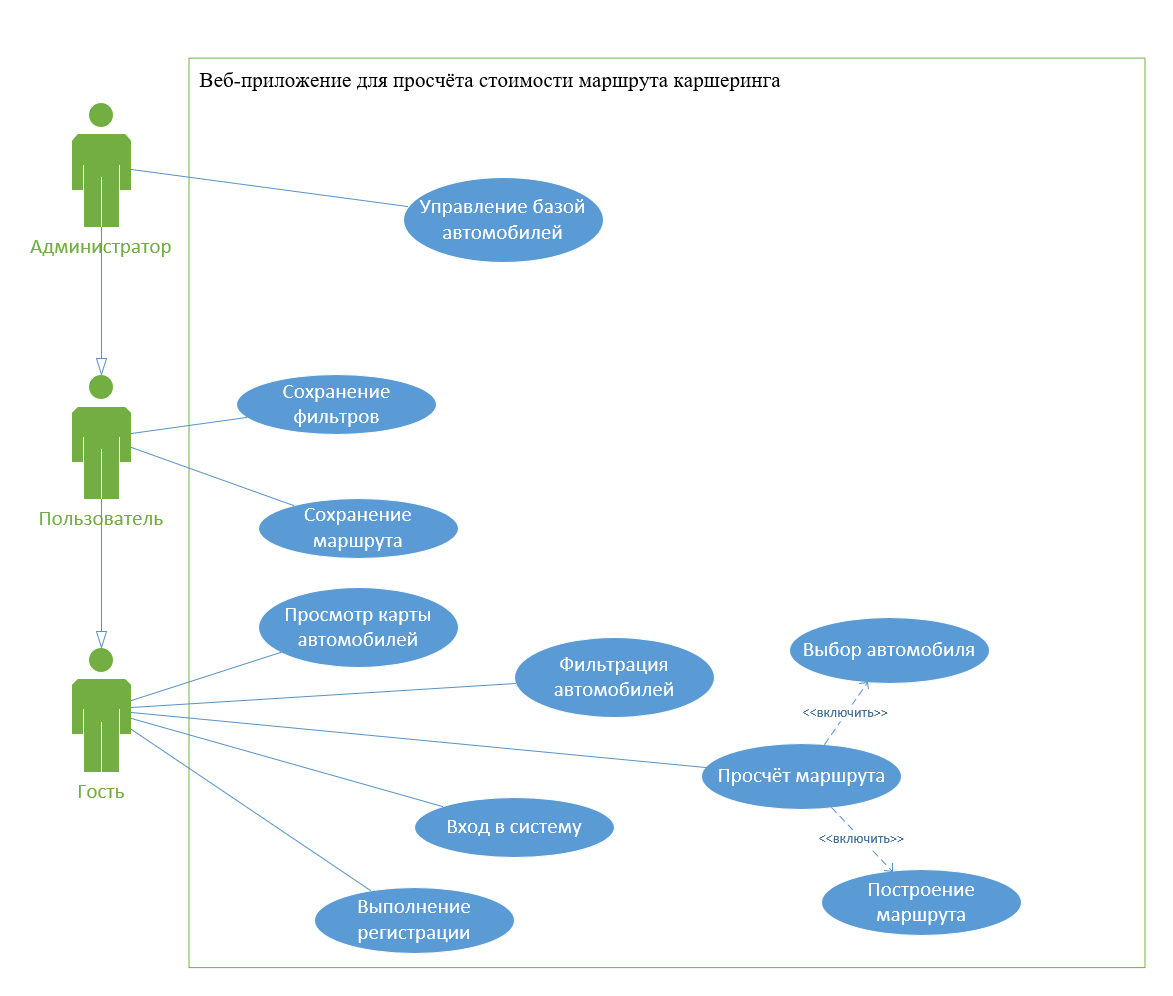
В итоге были реализован весь заявленный функционал. Как клиентская, так и серверная части проекта имеют хороший потенциал для будущих модификаций во многом благодаря грамотному структурированию, внимательности при написании кода, а также выборе известных и проверенных сопутствующих технологий. Было получено большое количество опыта по работе с такими фреймворками как GatsbyJs, React и ExpressJs, базой данных PostgreSQL, ORM системой Typeorm, Docker, Nginx и облачным сервисом AWS, улучшены навыки по проектированию, а также применению и связыванию друг с другом различных технологий.

# **Список используемых источников**

1. Асинхронный веб, или что такое веб-сокеты [Электронный ресурс]. — Режим доступа: <https://tproger.ru/translations/what-are-web-sockets/> Дата доступа: 13.10.2021.
2. Документация по TypeOrm [Электронный ресурс]. — Режим доступа: https://typeorm.io Дата доступа: 10.11.2021.
3. JavaScript [Электронный ресурс]. — Режим доступа:  <https://ru.wikipedia.org/wiki/JavaScript> Дата доступа: 01.11.2021.
4. Документация по React [Электронный ресурс]. — Режим доступа: https://ru.reactjs.org/ Дата доступа: 27.11.2021.
5. Что такое Node.js [Электронный ресурс]. — Режим доступа: <https://metanit.com/web/nodejs/1.1.php> Дата доступа: 02.11.2021.
6. Документация по WebRTC [Электронный ресурс]. — Режим доступа: https://webrtc.org/ Дата доступа: 04.11.2021.
7. Документация по GatsbyJs [Электронный ресурс]. — Режим доступа: https://gatsby.com/ Дата доступа: 22.11.2021.
8. Документация по Docker [Электронный ресурс]. — Режим доступа: https://www.docker.com / Дата доступа: 25.11.2021.
9. MaterialUI [Электронный ресурс]. — Режим доступа: https://material-ui.com/versions/Дата доступа: 27.10.2021.
10. Socket.IO [Электронный ресурс]. — Режим доступа: https://socket.io/ Дата доступа: 07.11.2021.
11. N-layer архитектура [Электронный ресурс]. — Режим доступа: <https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/n-tier> Дата доступа: 07.11.2021.

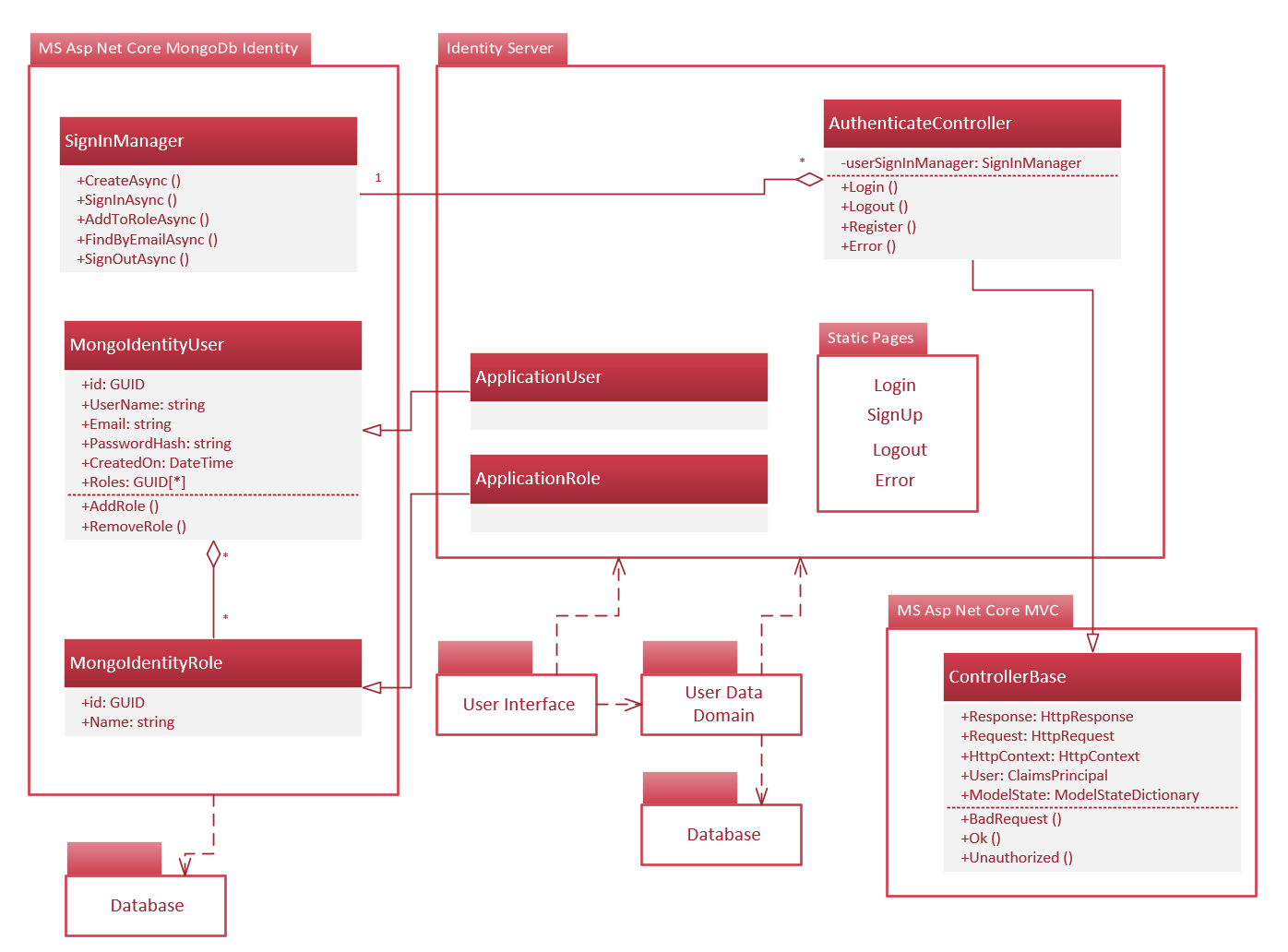
# **ПРИЛОЖЕНИЕ А**

UML диаграмма использования



# **ПРИЛОЖЕНИЕ Б**

UML диаграмма пакетов/классов



# **ПРИЛОЖЕНИЕ В**

import { Response, Router } from 'express'

import { Service, Container } from 'typedi'

import Controller from '../interfaces/controller.interface'

import validationMiddleware from '../middleware/validation.middleware'

import MeetingHighlightService from './meeting-highlight.service'

import HighlightNotFoundException from '../exceptions/highlight-not-found.exception'

import DeleteDto from './dto/delete.dto'

import { asyncWrapper } from '../utils/async-wrapper'

import AuthMiddleware from '../middleware/auth.middleware'

import { RequestWithUser } from 'src/types/custom'

@Service()

class MeetingHighlightController implements Controller {

    constructor(private meetingHighlightService: MeetingHighlightService) {

        meetingHighlightService = Container.get(MeetingHighlightService)

        this.initializeRoutes()

    }

    public path = '/highlights'

    public router = Router({ mergeParams: true })

    private initializeRoutes() {

        this.router.get(

            `${this.path}`,

            AuthMiddleware,

            asyncWrapper(this.getHighlights)

        )

        this.router.post(

            `${this.path}/share`,

            AuthMiddleware,

            asyncWrapper(this.shareHighlights)

        )

        this.router.post(

            `${this.path}`,

            AuthMiddleware,

            asyncWrapper(this.createHighlight)

        )

        this.router.delete(

            `${this.path}`,

            validationMiddleware(DeleteDto),

            AuthMiddleware,

            asyncWrapper(this.deleteHighlight)

        )

        this.router.put(

            `${this.path}`,

            AuthMiddleware,

            asyncWrapper(this.changeOrderHighlight)

        )

    }

    public getHighlights = async (

        request: RequestWithUser,

        response: Response

    ) => {

        const highlights =

            await this.meetingHighlightService.getHighlightsByMeeting(

                request.params.meetingId,

                request.user

            )

        response.send(highlights)

    }

    public createHighlight = async (

        request: RequestWithUser,

        response: Response

    ) => {

        const user: User = request.user

        const { highlight }: CreateDto = request.body

        const newHighlight = await this.meetingHighlightService.create(

            highlight,

            request.params.meetingId,

            user

        )

        response.send(newHighlight)

    }

    public changeOrderHighlight = async (

        request: RequestWithUser,

        response: Response

    ) => {

        const { highlights } = request.body

        const message = await this.meetingHighlightService.changeOrder(

            highlights

        )

        response.send(message)

    }

    public deleteHighlight = async (

        request: RequestWithUser,

        response: Response

    ) => {

        const user: User = request.user

        const { highlightId, isMine }: DeleteDto = request.body

        const highlight = await this.meetingHighlightService.delete(

            highlightId,

            isMine,

            user

        )

        if (!highlight) {

            throw new HighlightNotFoundException()

        }

        response.send(highlight)

    }

    public shareHighlights = async (

        request: RequestWithUser,

        response: Response

    ) => {

        const sharing = await this.meetingHighlightService.sharedHighlight(

            request.body.highlightId,

            request.body.sharedData,

            request.params.meetingId

        )

        response.send(sharing)

   }

}

export default MeetingHighlightController

# **ПРИЛОЖЕНИЕ Г**

import { IsString, IsOptional } from 'class-validator'

class RegisterUserDto {

  @IsString()

  @IsOptional()

  public firstName?: string

  @IsString()

  @IsOptional()

  public lastName?: string

  @IsString()

  @IsOptional()

  public password: string

  @IsString()

  @IsOptional()

  public termsLink?: string

  @IsString()

  @IsOptional()

  public verificationCode: string

}

export default RegisterUserDto

# **ПРИЛОЖЕНИЕ Д**

import React from 'react'

import Dashboard from './dashboard'

import ConferenceJoinStore from './conferenceJoinStore'

import ConferenceModalsStore from './conferenceModalsStore'

import MeetingStore from './meetingStore'

import LoginedUserStore from './loginedUserStore'

import RoomStore from './roomStore'

import RoomChatStore from './roomChatStore'

import JoinRoomRequestStore from './joinRoomRequestStore'

import HighlightsStore from './highlightsStore'

export class RootStore {

  conferenceJoinStore: ConferenceJoinStore

  conferenceModalsStore: ConferenceModalsStore

  meetingStore: MeetingStore

  dashboardStore: Dashboard

  loginedUserStore: LoginedUserStore

  roomChatStore: RoomChatStore

  joinRoomRequestStore: JoinRoomRequestStore

  roomStore: RoomStore

  highlightsStore: HighlightsStore

  constructor() {

    this.conferenceJoinStore = new ConferenceJoinStore(this)

    this.conferenceModalsStore = new ConferenceModalsStore(this)

    this.meetingStore = new MeetingStore(this)

    this.dashboardStore = new Dashboard(this)

    this.loginedUserStore = new LoginedUserStore(this)

    this.roomStore = new RoomStore(this)

    this.roomChatStore = new RoomChatStore(this)

    this.joinRoomRequestStore = new JoinRoomRequestStore(this)

    this.highlightsStore = new HighlightsStore(this)

  }

}

const StoresContext = React.createContext(new RootStore())

export const useStores = () => React.useContext(StoresContext)

# **ПРИЛОЖЕНИЕ Е**

FROM node:16-alpine AS BUILD\_IMAGE

RUN apk update && apk add --no-cache python3 py3-pip && apk add yarn curl bash make && rm -rf /var/cache/apk/\*

RUN apk add --update alpine-sdk && \

    apk add libffi-dev openssl-dev && \

    apk --no-cache --update add build-base

#  add libraries; sudo so non-root user added downstream can get sudo

RUN apk add --no-cache \

    g++ \

    cairo-dev \

    jpeg-dev \

    pango-dev \

    musl-dev \

    giflib-dev \

    pixman-dev \

    pangomm-dev \

    libjpeg-turbo-dev \

    freetype-dev

RUN curl -sfL https://install.goreleaser.com/github.com/tj/node-prune.sh | bash -s -- -b /usr/local/bi

WORKDIR /usr/src/app

RUN yarn --frozen-lockfile

COPY . .

RUN yarn install

RUN yarn build-dist

FROM node:16-alpine as prod

USER root

RUN apk add --no-cache \

    g++ \

    cairo-dev \

    jpeg-dev \

    pango-dev \

    musl-dev \

    giflib-dev \

    pixman-dev \

    pangomm-dev \

    libjpeg-turbo-dev \

    freetype-dev

RUN mkdir -p /home/node/app/

RUN mkdir -p /home/node/app/node\_modules

RUN mkdir -p /home/node/app/dist

RUN chown -R 1000:1000 /home/node/app

RUN chown -R 1000:1000 /home/node/app/node\_modules

RUN chown -R 1000:1000 /home/node/app/dist

WORKDIR /home/node/app

COPY --from=BUILD\_IMAGE /usr/src/app/package.json /home/node/app/

COPY --from=BUILD\_IMAGE /usr/src/app/development.env /home/node/app/

COPY --from=BUILD\_IMAGE /usr/src/app/dist /home/node/app/dist

COPY --from=BUILD\_IMAGE /usr/src/app/node\_modules /home/node/app/node\_modules

EXPOSE 3016

EXPOSE 10000-10100

CMD ["yarn", "start"]

# **ПРИЛОЖЕНИЕ Ж**

name: Deploy version dev

on:

  workflow\_dispatch:

    branches:

      - 'development'

  push:

    branches:

      - 'development'

jobs:

  CD:

    runs-on: ubuntu-latest

    steps:

      - name: Transfer docker-compose

        uses: actions/checkout@v2

      # Setup ke

      - run: set -eu

      - run: mkdir "$HOME/.ssh"

      - run: echo "${{ secrets.key }}" > "$HOME/.ssh/key"

      - run: chmod 600 "$HOME/.ssh/key"

      - run: cd docker-compose-dev

      - run: rsync -e "ssh -i $HOME/.ssh/key -o StrictHostKeyChecking=no" --rsync-path="sudo rsync" --archive --compress --delete . ubuntu@dev.meetqi.com:/var/www/docker

      - run: ssh -i $HOME/.ssh/key -o StrictHostKeyChecking=no ubuntu@dev.meetqi.com "docker-compose -f /var/www/docker/docker-compose-dev/docker-compose.yaml pull; docker-compose -f /var/www/docker/docker-compose-dev/docker-compose.yaml up  -d; docker image prune -f"