Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет Компьютерных систем и сетей
Кафедра Информатики

К защите допустить:

Заведующий кафедрой информатики

_____ Н. А. Волорова

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к дипломному проекту
на тему:

ОНЛАЙН КИНОТЕАТР

БГУИР ДП 1-40 04 01 00 029 ПЗ

Студент А. А. Красильников
Руководитель М. В. Стержанов
Консультанты:
от кафедры информатики М. В. Стержанов
по экономической части Т. А. Рыковская
Нормоконтролёр В. В. Шиманский

Рецензент

Минск 2020

РЕФЕРАТ

ОНЛАЙН КИНОТЕАТР : дипломный проект / А. А. Красильников. - Минск : БГУИР, 2020, - 72 с., чертежей (плакатов) - 6 л. формата А4.

Пояснительная записка 72 с., 9 рисунков, 2 таблиц, 16 формул и 17 источников.

Целью дипломного проекта является разработка удобного в использовании приложения видеоплеера. Основной особенностью является наличие механизма синхронизации возпроизведения, который позволяет совместно смотреть видео на расстоянии.

Для достижения цели дипломного проекта были рассмотрены различные варианты для реализации поставленной цели, рассмотрены аналогичные приложения, их плюсы и минусы, выработаны функциональные и нефункциональные требования.

Было разработано веб-приложение видеоплеер. Данное приложение позволяет пользователю создать комнату для совместного просмотра видео. Разработанный плеер будет автоматически синхронизировать процесс воспроизведения. Также реализованы текстовый чат, список воспроизведения и система поиска видео.

В разделе технико-экономического обоснования был произведён расчёт затрат на создание ПО и прибыли от разработки, получаемой разработчиком. Проведённые расчёты показали экономическую целесообразность проекта.

СОДЕРЖАНИЕ

ПЕРЕЧЕНЬ УСЛОВНЫХ ОБОЗНАЧЕНИЙ, СИМВОЛОВ И ТЕРМИНОВ

В настоящей пояснительной записке применяются следующие определения и сокращения.

API — Application Programming Interfaces.

JSON —текстовый формат обмена данными,основанный на JavaScript.

БД — база данных.

ПО — программное обеспечение.

HTML — HyperText Markup Language — язык гипертекстовой разметки.

MVC — Model View Controller.

NoSQL — not only SQL — не только SQL.

SQL — structured query language — язык структурированных запросов.

REST — Representation State Transfer — передача состояния представления.

JS — JavaScript.

ВВЕДЕНИЕ

Развитие мобильных сетей и повышение скорости интернета изменяет процесс общения между людьми. С друзьями можно мгновенно поделиться любым контентом, а новые способы общения на расстоянии продолжают развиваться.

Если обратиться к рейтингу самых посещаемых сайтов по версии SimilarWeb за 2019 год, можно заметить что пользователи проявляют повышенный интерес к сервисам предоставляющим возможность просматривать видео. Подобную востребованность видео материалов можно объяснить тем, что они просты для восприятия и способны удовлетворить большой список человеческих потребностей: от обучающих материалов до фильмов и сериалов. Подобный интерес стал возможен благодаря развитию веб-технологий, которые на сегодняшний день позволяют пользователю получать и обрабатывать данные на такой скорости, что просмотра видео стал для многих ежедневной рутиной.

За последние несколько лет произошло изменение того как люди предпочитают смотреть фильмы, сериалы и другие видео. Всё больше людей используют интернет и различные веб-сервисы для просмотра кино, которые пришли на смену классическому телевидению и кинотеатрам. До сих пор такие сервисы показывают рост и новые игроки пытаются занять свою нишу на рынке.

Также для современного общества очень важным является процесс социализации. Поэтому постоянно появляется много новых способов для общения и провождения времени с друзьями и близкими. Один из вариантов времяпрепровождения — совместный просмотр видео. Особенно остро стоит вопрос социализации и совместного времяпрепровождение в условиях глобальной пандемии COVID-19. Поэтому совместный просмотр видео на расстоянии актуален как никогда.

1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ. ПОСТАНОВКА ЗАДАЧИ

В рамках данного раздела будет произведён обзор предметной области данного дипломного проекта. Также будут рассмотрены возможные варианты реализаций и необходимые для этого технологий, сервисы, предлагающие аналогичные возможности, их плюсы и минусы и будут сформулированы основные требования к разрабатываемому ПО.

1.1 Цель дипломного проекта

Для определения ключевых функций приложения была произведён анализ статистики самых популярных веб-ресурсов и приложений у пользователей. Из статистики SimilarWeb за 2019 год, можно сделать вывод, что значительная часть ресурсов, исключая поисковые системы, например Google, связаны с просмотром видео. После изучения полученных результатов было решено, что необходимо создать видеоплеер, который позволит пользователю смотреть видео с кем-то на расстоянии.

1.2 Формат разрабатываемого приложения

В самом начале необходимо определиться с тем, как данное приложение будет функционировать. Под этим понимается необходимость выбора между разработкой настольного/мобильного или веб-приложения.

Под настольным/мобильным приложением имеется ввиду программа, которая для своей работы не требует наличия других программ на системе пользователя, кроме системных библиотек, характерных для ОС. Веб-приложение для своей работы требует наличие веб-браузера. Рассмотри плюсы и минусы этих реализаций ПО.

Для настольных приложений характерны высокая степень интеграции с системой пользователя и более низкое потребление системных ресурсов, так как веб-приложения для своей работы полагаются на веб-браузер. Веб-приложения способны без особого труда работать на разных операционных системах, так как напрямую от ОС веб-приложения не зависят. Это является огромным достоинством веб-приложений. Настольные приложения также можно реализовать используя кроссплатформенные библиотеки и фреймворки. Данные библиотеки накладывают определённые ограничения на итоговое приложение, так как в них часто отсутствует специфичные для ОС возможности. Веб-приложения также обладают таким ограничением, но степень раз-

вития веб-технологий и вовлечённость крупных разработчиков ОС в развитие веба позволяет закрыть на эти ограничения глаза. Также веб-приложения проще перенести на мобильные платформы, так как обычно для этого требуется только адаптация пользовательского интерфейса. С настольными приложениями так поступить невозможно из-за сильной зависимости от ОС. Благодаря развитию веб-технологий сегодня можно создать веб-приложение, которое будет неотличимо от настольного. Всё вышеперечисленное, а также многочисленные оптимизации веб-браузеров позволяют закрыть глаза на преимущества настольных приложений.

1.3 Анализ аналогов

1.3.1 YouTube

Видеохостинг от компании Google, предоставляющий пользователям услуги хранения и распространения видеофайлов. Также имеется возможность проведения прямых трансляций. YouTube обладает обширными возможностями: комментарии, система рейтинга, монетизация видео и многое другое. Из важных для проекта возможностей стоит выделить, функцию "Премьера". Она позволяет пользователю загрузить видеоролик у установить время когда начнётся его воспроизведение для остальных пользователей. Это похоже на прямой эфир. Все зрители видеоролика во время "Премьеры"будут наблюдать одно и тоже. На рисунке 1.1 можно видеть пример использования данной функции. Также стоит отметить полезную возможность, которая позволяет предоставить доступ к видеоролику только людям у которых есть ссылка на него.

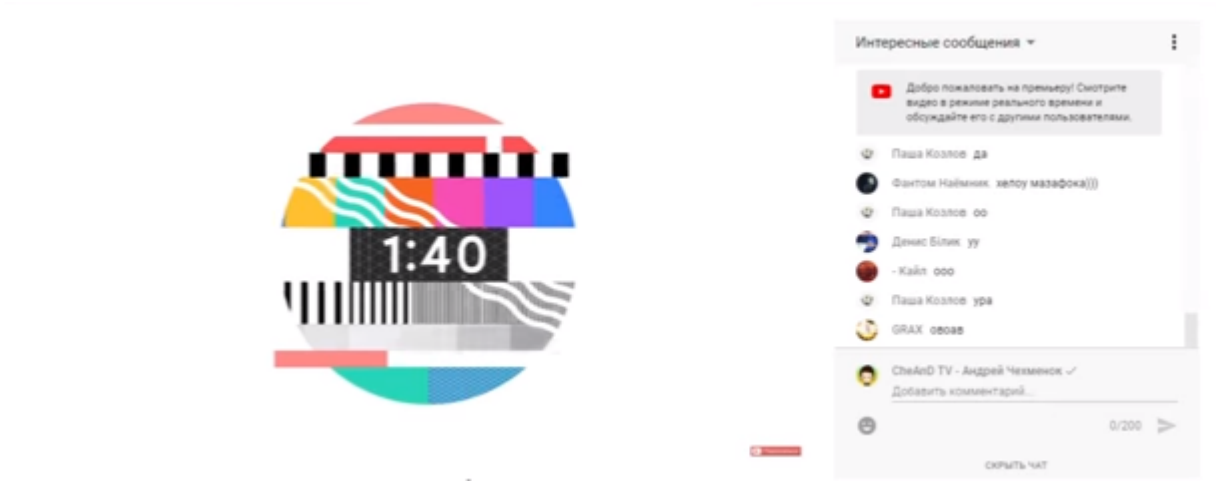

Рисунок 1.1 – YouTube Премьера

Плюсы:

– множество функциональных возможностей для обработки видео;
– система комментариев и рекомендации;
– возможность проведения прямых трансляций и "Премьер";
– возможность доступа к видео только по ссылке.

Минусы:

– региональные ограничения на некоторые возможности и видеоролики;
– необходимость предварительной загрузки видео на сервис.

1.3.2 Twitch

Стриминговый сервис, который позволяет пользователям транслировать видеопоток со своего компьютера другим пользователям сайта с минимальной задержкой. Вначале сервис использовался для транслирования видеоигр, но сейчас не ограничивается данной тематикой. Основной особенностью сервиса является высокая степень взаимодействия и социализации зрителей во время трансляций. Это достигается при помощи текстового чата, который имеет у каждой трансляции (пример на рисунке 1.2).

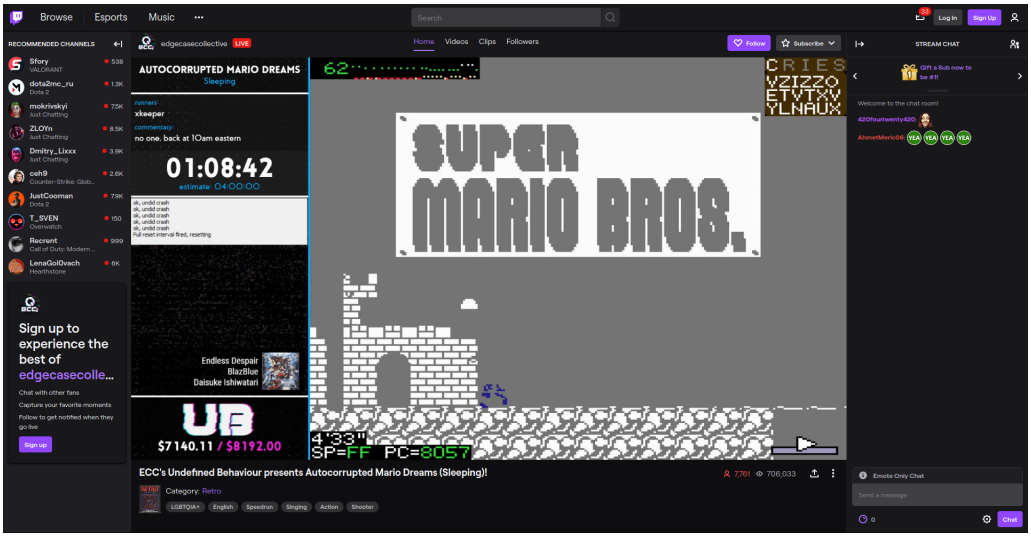

Рисунок 1.2 – Прямая трансляция на Twitch

Плюсы:

– пользовательский чат во время трансляции.

Минусы:

– отсутствие контроля у большинства пользователей, так как за трансляцию отвечает только один пользователь;
– наличие хорошего интернет соединения, оборудования и программного обеспечения для ведения трансляции;

– приватные трансляции доступны только для пользователей получивших одобрение со стороны сервиса.

1.3.3 Rave

Приложение, которое “сделает интереснее всё: от киноночей до местных музыкальных вечеринок”. Rave объединяет совместный просмотр видео, голосовое общение и текстовые сообщения. Создатели позиционируют Rave в большей степени как приложение для вечеринок: функция RaveDJ позволяет сводить песни при помощи компьютера и искусственного интеллекта.

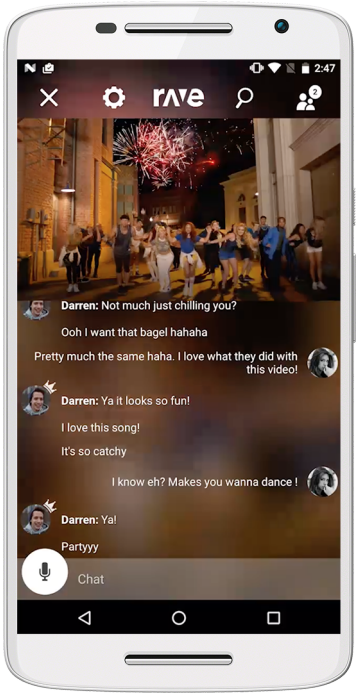

Рисунок 1.3 – Приложение Rave

Плюсы:
– текстовый и голосовой чат;
– большой список поддерживаемых сервисов;
– настройки режимов воспроизведения.

Минусы:
– требуется наличие учётной записи;
– отсутствует версия для персональных компьютеров.

1.4 Постановка задачи

В рамках данного дипломного проекта необходимо разработать веб-видеоплеер для совместного просмотра видео на расстоянии. При разработке необходимо выполнить следующие задачи:

– разработать веб-видеоплеер, который будет синхронизировать воспроизведение видео у пользователей;

– разработать пользовательский интерфейс для приложения;

– реализовать поддержку воспроизведения видеофайлов со сторонних ресурсов: YouTube, Vimeo.

1.5 Требования к приложению

Одним из самых важных этапов при разработке является сбор и организация требований. Хорошо описанные требования существенно снижают риски при проектировании архитектуры, разработке и тестировании программного продукта.

1.5.1 Нефункциональные требования

– минимизировать затраты на разработку первой версии приложения;

– веб-приложение должно работать на большинстве современных браузеров, в их число входят: Google Chrome 80 версии, Mozilla Firefox 74 версии, Safari 12 версии;

– поддержка двух последних версий поддерживаемых браузеров;

– идентичные функциональные возможности на разных веб-браузерах;

– идентичный пользовательский интерфейс на разных веб-браузерах;

1.5.2 Функциональные требования

– плеер способен поддерживать множество разных сеансов;

– плеер должен воспроизводить mp4 и webm видео;

– при паузе видео у одного пользователя оно останавливается у всех;

– при перемотке видео у одного пользователя оно перематывается у всех;

– плеер можно открыть в полноэкранном режиме;

1.6 Перспективы развития приложения

В дальнейшем приложение может быть улучшено при помощи введения дополнительных возможностей:

– выбор качества для видеофайла;

– создание собственного библиотеки видеоматериалов;

– добавление поддержки прямых трансляций;

– расширение списка поддерживаемых сторонних сервисов;

– добавление голосового чата.

2 ИСПОЛЬЗУЕМЫЕ ТЕХНОЛОГИИ

2.1 Синхронизация данных

При разработке видеоплеера было необходимо определиться с теми технологиями и методами синхронизации и обмена данными, которые доступны для реализации в веб-приложении. Были рассмотрены следующие варианты: стриминг видео, обмен синхронизирующих сообщений между приложениями, обмен синхронизирующих сообщений через сервер.

2.1.1 Стриминг

Разберёмся с понятием стриминга. В данном случае речь идёт об HTML5-стриминге, также существует понятие HTML5-видео. Различия заключаются в том, то для HTML5-видео используется готовый видео файл, а для HTML5-стриминга постоянный видеопоток (видео на YouTube — это HTML5-видео, а трансляция на Twitch — HTML5-стриминг).

Недостатком данного метода является наличие большого количества кодеков, транспортных и видео протоколов, которые имеют свои нюансы, ограничения и проблемы совместимости. Также данный метод не подходит для данного проекта, так как данная система накладывает ограничения на видеофайлы, которые пользователь сможет использовать, потому что сначала их необходимо загрузить на сервер, откуда они будут передаваться остальным пользователям. Это также ограничивает возможности синхронизации, так как при стриминге для обработки запросов плеера (пауза, перемотка) для множества сессий потребуются большие серверные мощности. Также это накладывает ограничения на скорость исходящего интернет соединения пользователя, необходимое для передачи видеопотока на сервер.

2.1.2 Обмен сообщений между приложениями

Второй вариант заключается в том, что каждое приложение является независимым и для синхронизации оно обменивается сообщениями о состоянии видео с другими приложениями. Данный метод позволяет снизить нагрузку на сервер, так как основные действия будут происходить на устройстве пользователя. Остаётся вопрос обмена синхронизирующими сообщениями. Для этих целей подходит технология Web Real-Time Communications. Web Real-Time Communications (WebRTC) - это технология, которая позволяет веб-приложениям и сайтам осуществлять захват и передачу аудио и/или видео потоков. Также имеется возможность обмениваться данными произвольного типа между браузерами, без необходимости посредника в этом

процессе. На данный момент реализация WebRTC в современных браузерах неполная, что выражается в различной степени реализованности как функций, так и кодеков WebRTC. WebRTC предлагает ряд интерфейсов для передачи медиа потоков и произвольных данных, контроля соединения, управление идентификацией и многое другое.

В основе WebRTC лежит ряд протоколов, которые обеспечивают возможность P2P соединения:

- Interactive Connectivity Establishment (ICE);
- Session Traversal Utilities for NAT (STUN);
- Session Traversal Utilities for NAT (STUN);
- Network Address Translation (NAT);
- Traversal Using Relays around NAT (TURN);
- Session Description Protocol (SDP).

Network Address Translation (NAT) - механизм, позволяющий заменить внутренний IP адрес и порт в пакетах, отправляемых с локального устройства, на внешний адрес маршрутизатора для доступа во внешнюю сеть. Существует несколько способов трансляции адреса: статический, при котором адреса соотносятся один к одному (это необходимо когда данное устройство должно быть доступно из внешней сети), динамически, когда внутреннему адресу ставиться один из доступных адресов, и перегруженный, при котором адреса нескольких устройств заменяются на один и тот же адрес, но с различными номерами портов. Существует несколько типов NAT, которые способны оказать трудности в процессе установки соединения между устройствами. Из них нам интересен симметричный NAT.

Симметричный NAT - трансляция, при которой каждый запрос с определённого внутреннего адреса и порта на конкретный внешний адрес и порт преобразуется в уникальный внешний адрес и порт. Если устройства с одинаковым адресом и портом отправят запрос на разные адреса и порты, им будут присвоены разные адреса. При данном типе NAT компьютер способен получать пакеты только от тех источников, которым он уже отправлял запросы, иначе говоря, запрос из внешней сети от неизвестного отправителя будет проигнорирован.

Interactive Connectivity Establishment (ICE) - это технология, используемая для нахождения кратчайшего пути коммуникации между двумя компьютерами. Данный протокол необходим для тех приложений, в которых обработка сообщений через центральный сервер является неэффективным как в плане скорости передачи данных, так и из-за дополнительных финансовых затрат.

Для установки соединения необходимо знать свой IP адрес и ограничения NAT. В этом помогает вспомогательный сервер, который реализует Session Traversal Utilities for NAT (STUN) протокол. Если роутер реализует симметричный NAT, то организация соединения по публичным IP адресам становиться невозможной. В таком случае необходимо использовать дополнительный сервер, реализующий Traversal Using Relays around NAT (TURN). Работа данного сервера заключается в пересылке данных в обход симметричного NAT. Этот подход имеет свои издержки и его принято использовать при отсутствии иных способов установить соединение.

Установки соединения между браузерами, используя WebRTC, также является нежелательным методом для данного проекта, потому что для того, чтобы все пользователи могли использовать видеоплеер, необходимо много дополнительной ресурсов на разработку и поддержание системы установки соединения. Минусом данного способа является его ненадёжность и высокие затраты на реализацию механизма обмена сообщений. Также данный метод является ненадёжным в ситуации потери соединения из-за независимости приложений, что является нежелательным для данного дипломного проекта.

2.1.3 Обмен сообщений с сервером

Для реализации поставленной цели было решено использовать метод, при котором приложения будут обмениваться сообщениями через серве. Это позволит избежать потери данных при сохранении приемлемой скорости. Приложения будут точно зантъ необходимое состояние видео, так как верная информация об этом будет всегда находиться на сервере. Также данный метод позволяет при рассинхронизации принудительно синхронизировать клиенты, так как имеется единый источник информации. Описание способа обмена сообщений через червер представлено в разделе 2.2.

2.2 Технологии для сервера

Для того, чтобы упростить разработку приложения, было решено использовать сервис Firebase от компании Google. Данный сервис предлагает множество модулей, которые очень полезны для веб-разработки, создания мобильных или настольных приложений. Данный сервис избавляет от нужды в собственном сервере, так как Firebase реализует практически все возможности, которые необходимы от стандартного сервера. Главным инструментом в составе Firebase является их база данных. На данный момент Firebase предлагает несколько различных вариантов баз данных: Realtime Database и

Firestore. Их главное отличие заключается в форме хранения данных и доступа к ним.

Realtime Database использует JSON файлы для хранения информации. Это является крайне неэффективным, так как для получения определённых данных сначала нужно получить все данные из файла, а только потом можно произвести поиск. Это увеличивает потребность приложения в ресурсах системы, что недопустимо для данного проекта.

Firestore — облачная NoSQL база данных реального времени. Это означает то, что, при использовании данной базы данных, пользователь может "подписаться"на определённые документы. В таком случае при изменении содержимого документа пользователь сразу получит новые данные. Благодаря данному механизму можно быть уверенным, что конечный пользователь получит самую свежую информацию, что является важным для данного проекта. Firestore поддерживает следующие типы данных:

- текстовая строка;
- числа;
- булевые значения;
- массивы;
- даты;
- словари.

Так как Firestore является NoSQL базой данных, то для организации данных в ней используются не таблицы, а документы. Отличие заключается в том, что документ не имеет жёсткой структуры, как таблицы, что позволяет хранить в документах данные произвольных типов и менять их значение и структуру по мере необходимости. Данный способ позволяет более гибко взаимодействовать с данными, но лишает дополнительной надёжности, свойственной SQL базам данных.

Для организации документов используются коллекции. Коллекция — это просто набор документов. Они не обязаны быть одинаковыми, но данный случай крайне нежелателен. Каждый документы в коллекции имеет идентификатор, уникальный для данной коллекции. В качестве идентификатора может выступать любая подходящая текстовая строка. Документы также способны содержать внутри себя вложенные коллекции, однако данная возможность не особо полезна. Также имеется возможность создания пользовательских функций и триггеров для взаимодействия с базой данных, что позволяет выполнять определённые операции при изменении, создании или удалении документов и коллекций.

Firebase также предлагает встроенную систему аутентификации поль-

зователей, которая поддерживает различные источники для идентификации: почта, профиль Google и др.

Также плюсом Firebase является возможность его интеграции с другими сервисами компании Google и дополнительная защита от прекращения работы собственного сервера.

2.3 Недостатки Firebase

Основные возможности, которые предоставляет Firebase связанные с хранением, изменением и получением информации из встроенной базы данных. Firebase не имеет возможности для создания API, поэтому для решения вспомогательных задач, не связанных с базой данных необходим дополнительный API-сервер, который будет заниматься обработкой запросов и обращаться к Firestore по мере необходимости. Это позволит избежать проблем, связанных с невозможность выполнения некоторых операций на компьютере конечного пользователя. Так как самые ресурсозатратные операции выполняет Firebase, данный сервер может выполнять небольшой список операций, что очень полезно для данного проекта.

2.4 Выбор языка программирования

Так как было решено решено разработать веб-приложение, разработку необходимо вести на языке, который имеет поддержку для выполнения внутри браузера. К таковым можно отнести: JavaScript, TypeScript, Kotlin. По своей сути все варианты языков это JavaScript, так как браузер работает именно с ним, но выбор того или иного языка может сказаться на процессе разработки и дальнейшей поддержке.

Kotlin является наиболее молодым из тройки и из-за данного обстоятельства может вызвать трудности у разработчиков не знакомых с проектом, так как на текущий момент основное применение kotlin находит в android разработке. Помимо этого механизм работы kotlina для веб-разработки создаёт лишние трудности, ввиду необходимости трансляции кода на Kotlin в рабочий JavaScript код.

TypeScript - язык программирования от компании Microsoft, как альтернатива JavaScript, расширяющая его возможности. Основной особенностью TypeScript можно считать систему типизации, так как в отличии от обычного JavaScript, она статическая. Однако реализация типизации в TypeScript не решает проблемы динамической типизации JavaScript полностью, так как всё равно возможны случаи, когда переменная может иметь тип отличный от

объявленного. Также стоит отметить, что многие возможности TypeScript в том или ином виде приходят и в JavaScript, что делает TypeScript не лучшим выбором в долгосрочной перспективе.

2.5 JavaScript

Таким образом был сделан выбор в сторону языка JavaScript. Основной целью JavaScript является придание статическим HTML страницам динамичности, например отправка форм без перезагрузки страницы, изменение разметки страницы и стилей элементов на лету и многое другое. Программы написанные на JavaScript называются скриптами, они представляют из себя простой текстовый файл. Для работы JavaScript скриптов не нужна дополнительная подготовка или компиляция для работы. За выполнение JavaScript кода отвечает отдельная программа, которую обычно называют движком или «виртуальной машиной JavaScript». За счёт это JavaScript скрипты можно выполнять не только в браузере, но и на сервере при наличии JavaScript движка, который займется выполнением кода. Движок берёт код из скрипта, преобразует его в машинный код и затем выполняет его.

Современные браузеры в своём составе имеют собственные встроенные JavaScript движки для выполнения скриптов, например V8 в Chrome или SpiderMonkey в FireFox[1]. Хоть в основном реализации различных JavaScript движком похожи, но в них всё равно присутствуют различия, которые способны сказаться на работе JS кода в разных браузерах. Также могут быть различия в разных версиях одного и того же движка. Подобные нюансы необходимо учитывать при разработке веб-приложений. Для решения подобных проблем в JavaScript присутствуем механизм полифилов, позволяющий использовать новые и недоступные функции, на разных или старых браузерах. У JavaScript имеется собственный открытый стандарт ECMAScript, который должен быть реализован движками, но данный процесс происходить не моментально.

JavaScript отличается от многих современных языков программирования. Эти отличия многими воспринимаются неоднозначно, так как они способны предложить богатую функциональность, но также эти отличия могут стать причиной непредвиденных неполадок в работе приложения. К таким отличиям можно отнести систему наследования. В отличии многих других языков, в которых для наследования используется концепция классов, в JavaScript наследование опирается на понятие прототипа. Прототип — это объект, который имеет набор свойств и методов. Это значит, что объекты одного типа

могут иметь разный набор свойств. Даже встроенные типы, структуры данных и функции представляют из себя обычные объекты с определенным набором свойств и функций. Для получения возможностей массива объекту достаточно указать объект, реализующий методы массива как прототип. Также при помощи данной особенность можно расширить возможности базовых типов, расширяя их прототипы. В стандарте ECMAScript 6 ввели синтаксис для классического описания классов, но данное нововведение является синтаксическим сахаром, так как никак не меняет механизмы наследования, используемые в JavaScript.

Другой особенностью JavaScript является контекст выполнения кода (значение переменной this). Данная переменная ссылается на текущий объект при выполнении кода (окно браузера или конкретный объект).То, какой объект является текущим, может вызвать трудности. Если в других языках при выполнении вложенных методов классов переменная, ссылающаяся на основной объект, остаётся прежней (self в Python), то в подобной ситуации в JavaScript произойдёт смена контекста выполнения на глобальный. В результате этого произойдёт потеря текущего объекта при выполнении функции, что может привести к непредвиденным ошибкам. На данный момент для решения данной проблемы в стандарте ECMAScript 6 ввели стрелочные функции. Их основной особенностью является то, что данные функции сохраняю контекст, в котором их вызывают.

Также среди возможностей JavaScript можно выделить:

- объекты с возможностью интроспекции;
- функции как объекты первого класса;
- автоматическое приведение типов;
- автоматическая сборка мусора;
- анонимные функции.

2.6 Построение пользовательского интерфейса

Далее встаёт вопрос визуализации пользовательского интерфейса. Любой сайт представляет из себя набор HTML документов, CSS стилей и JS скриптов. HTML, CSS отвечают за построение, структуру и визуальное оформление содержимого сайта. Хоть для создания динамического веб-приложения можно использовать только HTML, CSS и JavaSript, это подход вызовет дополнительные трудности при разработке большого количества динамический элементов и их дальнейшей поддержке. Поэтому для создания пользовательского интерфейса было решено использовать специализированные для этого

библиотеки и фреймворки. Среди них можно выделить:

– AngularJS
– VueJS
– ReactJS

От AngularJS было решено отказаться по некоторым причинам: слишком большие размеры фреймворка, вызванные тем, что Angular предлагает много функциональных возможностей, однако эти возможности излишни для данного дипломного проекта. Также, в сравнении с React и Vue, Angular обладает проблемами с производительностью, сложен в тестировании и отладке кода из-за своих архитектурных решений.

Vue и React разделяют многие общие идеи и функции. В данном случае выбор библиотеки обоснован личными предпочтениями разработчика. React обладает большей поддержкой из-зв своей популярности и предлагает больше возможностей чем Vue, оставаясь небольшим по размеру. Также использование React может облегчить дальнейшую разработку мобильной версии приложения, благодаря React Native. Эти две библиотеки, хоть и предназначены для разных вещей (веб и мобильная разработка соответственно), обладают схожей архитектурой и принципами работы. Этот факт является крайне полезным при портировании существующего веб-приложения на мобильные платформы IOS и Android.

2.7 ReactJS

ReactJS - JavaScript библиотека для создания пользовательских интерфейсов, разработанная компанией Facebook. React обладает хорошей производительностью и архитектурой для создания динамичных веб-приложений. Среди основных преимуществ стоит выделить: виртуальный DOM и JSX.

2.7.1 Виртуальный DOM

Основной концепцией и архитектурной особенностью React является виртуальный DOM[2]. Для взаимодействия с HTML документом используется Document Object Model — платформо независимый интерфейс, позволяющий программам и скриптам получить доступ к содержимому HTML, XHTML, XML документов, модифицировать их содержимое и оформление. Элементы документа представляются в качестве узлов со связями "родитель-потомок". Проблема DOM заключается в его неспособности эффективно работать с современными динамическими пользовательскими интерфейсами,

так как обработка большого количества элементов может занимать слишком много времени. Данную проблему можно решить, используя различные ухищрения, но глобально это не решает проблемы разработки современных динамических интерфейсов. Решить эту проблему призвана концепция виртуального DOM. Виртуальный DOM не является общепринятой технологией или стандартом, это подход, который заключается в том, что взаимодействия происходят с DOM не напрямую. Код оперирует не DOM, а его легковесной копией. После обновления копии происходит процесс согласования реального DOM и виртуального. Во время данного процесса происходит сравнение текущего дерева элементов и нового, затем для изменённых элементов происходит перестроение и изменение структуры поддерева DOM. React берет на себя процесс сравнения деревьев, стараясь обеспечить наилучшую производительность и скорость. Однако React не решает всех проблем автоматически. Разработчику необходимо контролировать поток данных и проектировать интерфейс так, чтобы изменяемые элементы не имели больших поддеревьев DOM. Это позволит избежать излишних перестроек реального DOM, что значительно увеличивает производительность приложения.

2.7.2 JSX

Для описание пользовательского интерфейса React используют JSX — расширение языка JavaScript. Он напоминает стандартный язык шаблонов, но обладает всей силой JavaScript. React не разделяет разные компоненты приложения, такие как представления и логика. Вместо этого React использует абстрактную структуру "компонент которая объединяет в себе логику пользовательского интерфейса и сам интерфейс. Компонент является представлением DOM элементов в его виртуальной копии. При работе приложения JSX преобразуется в специальную функцию для создания DOM элемента, которая затем транслируется при помощи Babel в обычный JavaScript.

2.8 Технологии для серверной части

Ни одно веб-приложение невозможно без серверной части, которая отвечает за работу с базой данных, обработку данных и выполнение роли API, и данный проект не исключение. При выборе средств разработки основными критериями были: знакомство разработчика с данными инструментами, возможность быстрого создания минимально необходимой функциональности, без необходимости поддержки большой кодовой базы, легковесность, возможность расширения в дальнейшем при наличии необходимости. Наилучшим выбором оказался язык программирования Python и библиотека Flask.

2.8.1 Python

Python - высокоуровневый, многоплатформенный язык программирования общего назначения, является одним из наиболее востребованных языков, используемых при создании приложений, системных средств, в различных научных целях и веб-разработке[3]. Данный язык ориентирован на повышение скорости разработки и читаемость кода. Среди достоинств и особенностей языка можно выделить:

- скорость разработки;
- динамическая типизация;
- множество сторонних модулей и библиотек;
- высокая масштабируемость;
- многочисленное сообщество;
- встраиваемость.

Python поддерживает различные парадигмы программирования, среди них: структурное, объектно-ориентированное, функциональное, императивное и аспектно-ориентированное программирование. Python является динамически типизированным языком программирования, это означает что тип переменной определяется в момент присваивания значения этой переменной. Данная особенность позволяет ускорить процесс разработки при работе с данными переменных типов и изменяющемся окружении[4]. Другим важным плюсом Python является собственный пакетный менеджер, при помощи которого можно добавить сторонние модули для расширения возможностей языка, например модуль numpy добавляет множество классов и функций для выполнения математических исследований. Одним из главных преимуществ Python для веб-разработки является наличие большого количества популярных и функциональных библиотек. В сравнении со многими другими языками, например Java или C#, Python сочетает в себе как продвинутые возможности для разработки веб-приложений, так и простоту их создания, не нуждаясь в громоздких и сложных фреймворках.

При выборе были рассмотрены и недостатки языка, важнейшим из которых был недостаток в скорости. Python является языком с полной динамической типизацией, автоматическим управлением памятью. Если на первый взгляд это может казаться преимуществом, то при разработке программ с повышенным требованием к эффективности, Python может значительно проигрывать по скорости своим статическим братьям (C/C++, Java, Go). Что касается динамических собратьев (PHP, Ruby, JavaScript), то здесь дела обстоят намного лучше, Python в большинстве случаев выполняет код быстрее за счет предварительной компиляции в байт-код и значительной части стандартной

библиотеки, написанной на C.

2.8.2 Flask

В качестве библиотеки для реализации серверной части был выбран Flask. Flask позиционирует себя как микро-фреймворк с возможностью к расширению. Это выражается в том, что сама библиотека содержит очень небольшой набор базовых возможностей, а при необходимости тех или иных дополнительных функций нужно использовать сторонние совместимые инструменты. В противовес данному подходу можно выделить другую крайне популярную Python библиотеку — Django. Она предлагает разработчику практически полный набор всего необходимо для написания веб-приложения, при этом накладывая ряд архитектурных ограничений. Однако для того, чтобы получить необходимую для данного проекта функциональность (API-сервер) в Django, необходимо использовать дополнительную библиотеку — Django REST. Она добавляет полноценный возможности REST API-сервера. В совокупности Django и Django REST требуют крайне много ресурсов на разработку и поддержание, особенно при требовании небольшой функциональности. Поэтому Flask оказался предпочтительнее из-за своего размера и расширяемости.

3 АРХИТЕКТУРА И ПРОЕКТИРОВАНИЕ ПРИЛОЖЕНИЯ

3.1 Компоненты и Flux

Так как для разработки был выбран ReactJS, то при разработке было решено придерживаться характерных для данной библиотеки архитектурных подходов. React использует компонентно-ориентированную архитектуру. Она заключается в том, что визуальное представление, данные, от которых зависит пользовательский интерфейс, и логика интерфейса выносятся в единую сущность — компонент. Данный подход можно сравнить с MVC. Отличие заключается в том, что только V - View полностью представлен в компонентно-ориентированной архитектуре. Задачи Model и Controller, которые непосредственно связаны с View, перенесены в компонент.

Для того, чтобы заполнить пробелы компонентно-ориентированной архитектуры, компания Facebook предлагает использовать вместе с React такой архитектурный подход, как Flux. Схема данных архитектур представлена на рисунке 3.1.

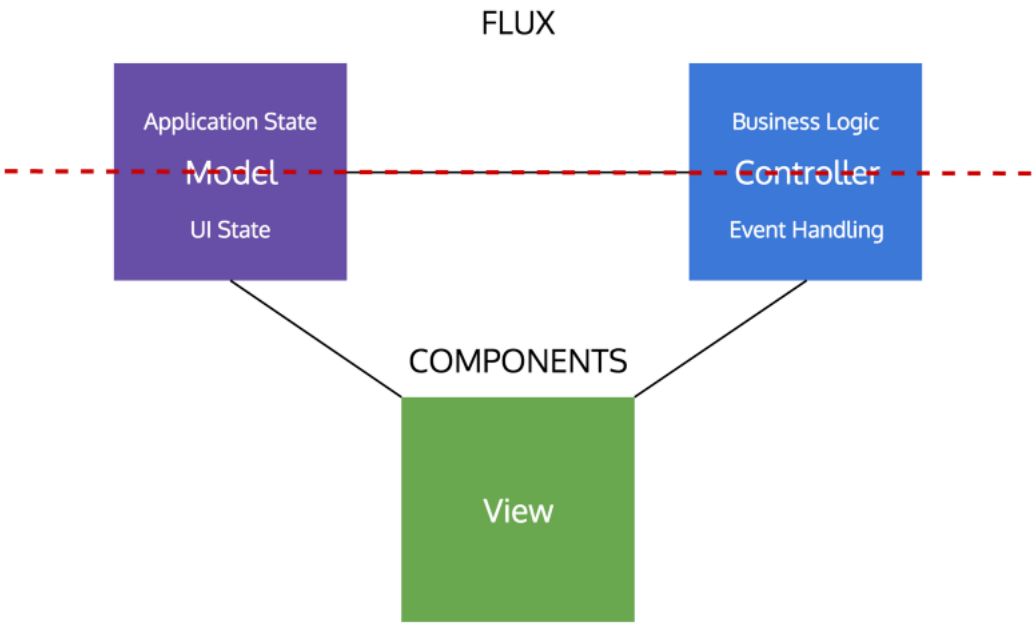

Рисунок 3.1 – Flux и компонентно-ориентированная архитектура

Главной особенностью Flux является односторонняя направленность передачи данных между компонентами Flux-архитектуры. Данный подход делает поток данных предсказуемым и позволяет легче производить тестирование и поиск ошибок. Flux-архитектура содержит три основных слоя:

– action (действие);
– store (хранилище);
– view (представление).

Действие представляет собой набор из имени и необязательной полезной нагрузки. Хранилище является источником состояния приложения, который при получении действий обновляет своё состояние. Состояние из хранилища используется представлениями для презентации информации пользователю. Популярной библиотекой реализующей Flux-архитектуру является Redux[5]. Она используется в данном проекте для реализации Flux-архитектуры.

3.2 Serverless архитектура

Serverless-вычисления (бессерверные-вычисления) - модель облачных вычислений, в которых платформа динамически руководит выделением вычислительных ресурсов. В данном случае бессерверный не означает отсутствие сервера как такового, под этим понимается то, что пользователю конкретной платформы не нужно заниматься созданием и настройкой собственного сервера. Данный подход позволяет значительно сэкономить ресурсы и уменьшить срок разработки, так как многие важные аспекты серверной части на себе берёт платформа. В рамках данного проекта такой платформой выступает Firebase.

3.3 Организация и описание модулей приложения

Разработанное веб-приложение можно разделить на три составляющие:
– Клиентское приложения на ReactJS;
– Модуль Firebase для работы с базой данных;
– API сервер для обработки запросов;

Клиентское приложение является основным модулем для данного проекта. В рамках клиентского приложения реализованы основные функциональные возможности данного проекта:

– создание профиля пользователя и управление пользовательскими данными;
– просмотр и поиск доступных комнат;
– создание комнаты;
– подключение к комнате;
– управление воспроизведением при помощи плеера.

Модуль Firebase отвечает за управление данными, их организацию и процесс передачи данных клиентскому приложению и Python сервису. Также Firebase используется для авторизации и управления пользователями и их данными.

Python сервис представляет из себя API сервер, основная функция которого заключается в обработке текста, который пользователь вводит при добавлении видео в текущую комнату. Сервер, анализируя полученные данные, старается найти подходящий вариант видеоролика среди поддерживаемых сервисов. Для ускорения выдачи результатов в будущем, часть данных сохраняется в Firebase.

На рисунке 3.2 изображена схема взаимодействия вышеперечисленных модулей.

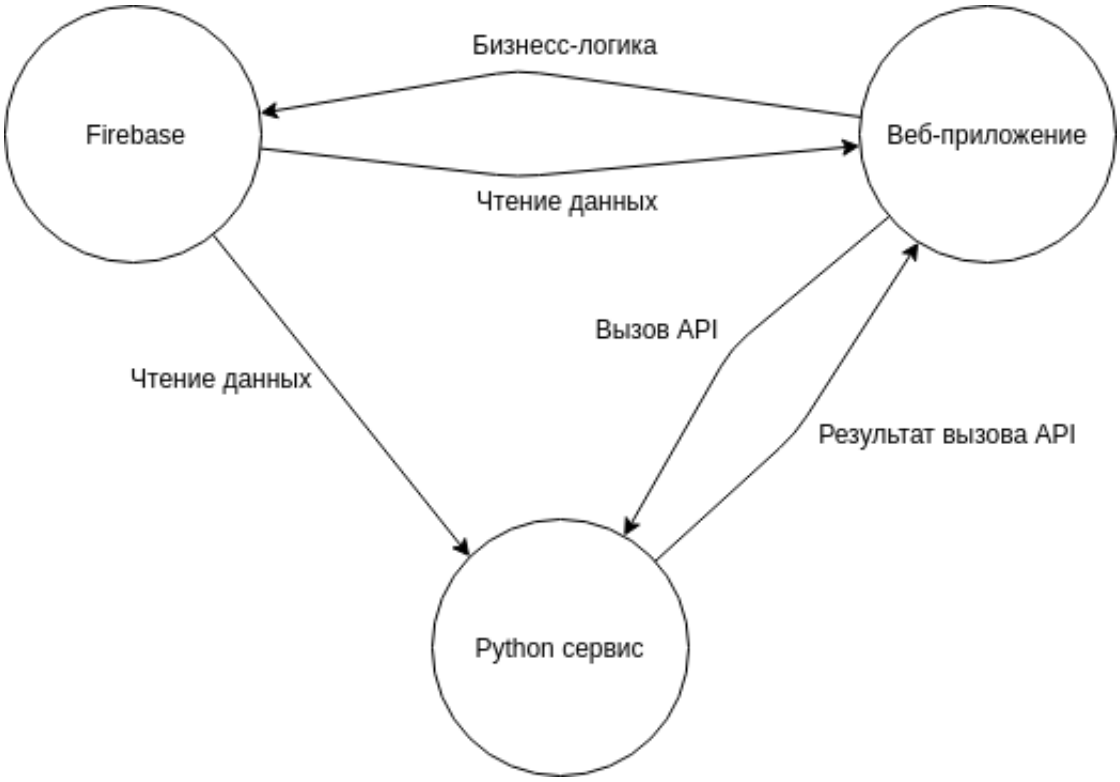

Рисунок 3.2 – Структура модулей приложения

3.4 Описание сущностей данных

Любое приложение используется для взаимодействия с данными, это может быть как работа с определёнными документами, так данные и сущности необходимы для реализации бизнес-логики. Рассмотрим основные сущности данных, их организацию и связи, необходимы для работы веб-приложения. Для организации данных используется несколько ключевых коллекций:

– rooms;

– roominfo;
– playlists;
– messages.

Так управление пользователями осуществляет Firebase, то как таковой модели данных для неё не требуются, но для полноты картины ниже представлена структура модели пользователя:

– uid (тип строки, уникальный идентификатор пользователя);
– displayName (тип строки, произвольное имя пользователя);
– email (тип строки, электронный адрес пользователя).

3.4.1 Коллекция rooms

Данная коллекция содержит документы, которые необходимы для синхронизации состояния воспроизведения у разных клиентских приложений. Основная информация, которая описывает состояние видеоролика, это:

– источник текущего видеоролика;
– время текущего видеоролика;
– состояние воспроизведения (на паузе видео или нет).

Исходя из необходимого набора данных, документы коллекции rooms имеют три поля:

– src (имеет тип строки, хранит ссылку на источник видеофайла);
– time (имеет тип числа, хранит текущее время видео в секундах);
– paused (имеет булевый тип, если true, то видео должно быть на паузе и наоборот).

3.4.2 Коллекция roominfo

Данная коллекция служит для объединения информации характерной для сущности комнаты, которая необходима для её идентификации и поиска. Среди доступных полей имеются:

– name (имеет тип строки, содержит название комнаты);
– hasPassword (имеет булевый тип, используется для отображения формы для ввода пароля при входе в комнаты, если имеет значение true);
– password (имеет тип строки, хранит в себе пароль указанный пользователем при создании комнаты).

3.4.3 Коллекция playlists

Данная коллекция хранит документы, которые являются представлением текущего списка видеороликов для конкретной комнаты. Документы данной коллекции имеют следующую структуру:

– videos (имеет тип массива, который содержит объекты, которые описывают сущность видеоролика).

У объектов видеороликов нет представления в виде документов, но эта структура данных крайне важна, так как в ней содержатся основные данные необходимые для воспроизведения и идентификации видеоролика. Данный объект имеет следующую структуру:

– name (имеет тип строки, содержит название видеоролика);

– src (имеет тип словаря, содержит набор пар ключ значение для различных источников данного видео);

– userInput (имеет тип строки, содержит значение, при помощи которого был добавлен видеоролик).

3.4.4 Коллекция messages

Данная коллекция содержит документы, которые являются представлением пользовательского чата конкретной комнаты. Документы данной коллекции имеют следующую структуру:

– messages (имеет тип массива, который содержит объекты, которые описывают сущность сообщения).

Объекты сообщения похожи по своей сути на объекты видео из подраздела 3.4.3. Данные объекты имеют следующую структуру:

– username (имеет тип строки, содержит имя пользователя, отображаемое в чате);

– userId (имеет тип строки, содержит уникальный идентификатоп пользователя);

– message (имеет тип строки, содержит пользовательское сообшение отправленное в чат)

– date (имеет тип даты, содержит дату отправления сообщения пользователем);

3.4.5 Организация документов из разных коллекций

Документы в перечисленных коллекциях не содержат явного идентификатора комнаты среди своих полей, это вызвано тем, что у каждого документа коллекции должен быть уникальный для неё идентификатор, по которому можно обратиться к документу. При создании комнаты в первую очередь происходит инициализация документа из коллекции roominfo. Новому документу в качестве идентификатора ставится сгенерированный UUID. При удачном создании этого документа, происходит создание документов в других коллекциях (эти документы будут иметь такой же уникальный иденти-

фикатор).

Благодаря этому после получения списка идентификаторов комнат, мы имеем возможность получить необходимую в данный момент информацию из необходимой коллекции по известному идентификатору. Это сделано для минимизации данных, которые пользователю придётся получать с сервера во время синхронизации видео. Структура документов и их связи представлена на рисунке 3.3.

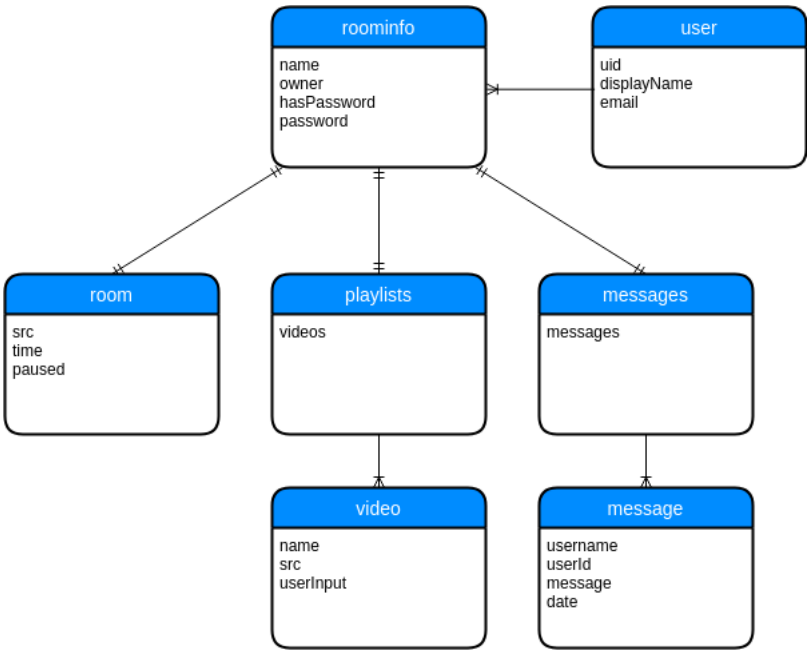

Рисунок 3.3 – Структура данных

3.5 API сервер

Изначально стояла задача разработать веб-приложение без написания собственного сервера, это позволило бы снизить трудозатраты на разработку и дальнейшую поддержку. Однако в ходе проектирования выяснилось, что реализовать возможность по поиску видео без использования сервера невозможно из-за ограничений сторонних видеосервисов. Из-за данного ограничения было решено разработать вспомогательный API сервер на языке Python. Данный сервер нужен для обработки информации и выполнения операций, которые невозможно выполнить при помощи Firebase или в самом веб-приложении.

На данный момент его единственная задача заключается в том, чтобы обработать текст введённый пользователем при добавлении видео в список воспроизведения. Затем ему необходимо определить то, чем является введенные данные: текстом или ссылкой. Если данные являются ссылкой на

страницу содержащую видеоролик и наше веб-приложение поддерживает данный видеосервис, сервер запрашивает информацию о видео напрямую с видеосервиса, если такая возможность доступна. В противном случае сервер старается получить данную информации другими способами (Для YouTube необходима сторонний модуль для Python youtube-dl). Если введенные пользователем данные являются текстом, происходит поиск по данному тексту на поддерживаемых видеосервисах. При нахождении подходящего видеоролика, данные полученные от видесервиса преобразуются в формат, который использует веб-приложение. Эти данные должны соответствовать структуре описанной в подразделе 3.4.3. После преобразования данных они отправляются обратно клиенту в формате JSON. Схема алгоритма по обработке пользовательского запроса представлена на рисунке 3.4.

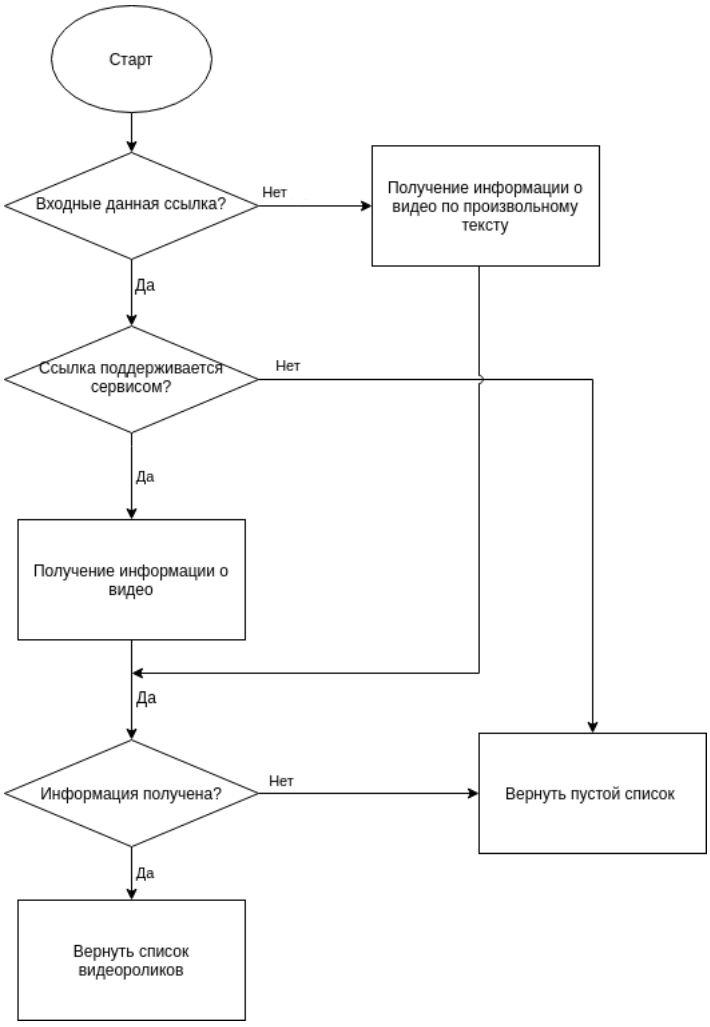

Рисунок 3.4 – Алгоритм поиска видео

4 РЕАЛИЗАЦИЯ ВЕБ-ПРИЛОЖЕНИЯ

По итогу разработки было реализовано веб-приложение, которое позволяет пользователю организовать комнату для совместного просмотра видео. Примеры пользовательского интерфейса представлены на рисунках 4.1 и 4.2. Для воспроизведения видео используется тег <video>, который был введён в стандарте HTML5.

Данный стандарт определяет для тега <video> ряд атрибутов и событий, для контроля воспроизведения. Проблема заключается в том, что стандартные элементы управления тега <video> не обладают всеми необходимыми возможностями для реализации данного проекта. Также при помощи JS кода нет никакой возможности изменить события при взаимодействии со стандартные элементы управления. Можно только добавить собственную логику, которая будет работать после выполнения стандартной. Данное ограничение делает невозможным реализацию хорошей синхронизации, так как у пользователя инициировавшего событие оно произойдёт сразу, а у остальных после получения данных с сервера.

Для решения проблемы синхронизации был реализован собственный пользовательский интерфейс (см. рис. 4.2), который позволяет полностью манипулировать порядком выполняемых действий. Реализация данного пользовательского интерфейса представлена в листинге 4.1.

Листинг 4.1 – Компонент Player

```
import React, { Component } from 'react';
import { MdPlayArrow, MdPause, MdFullscreen, MdQueue } from 'react-icons/
md';
import './player.scss';
import TimeIndicator from './TimeIndicator';
import RangeBar from './Slider';
import Hotkeys from 'react-hot-keys';
import VolumeSlider from './VolumeSlider';
import { PlayerBackend } from '../PlayerBackend/default';
import PlayList from './PlayList';
import Notifications from './Notifications';
import Chat from './Chat';

const pauseIcon = <MdPause />;
const playIcon = <MdPlayArrow />;

class Player extends Component {
   constructor(props) {
       super(props);
       this.processor = new PlayerBackend();
       this.playerElemetn = React.createRef();
       this.state = {
```

```
            isFullscreen: false,
            playbackIcon: playIcon,
            playbackState: false,
            playlist: false,
            showVideoChooser: false,
            showChat: false,
        }
    }

    componentDidMount() {
        this.vid_title = document.getElementById('video-title');
        this._connectToPlayerBackend();
        document.onfullscreenchange = () => this.setState({ isFullscreen: !
this.state.isFullscreen });
        if (this.props.externalSetUp) {
            for (const func of this.props.externalSetUp) {
                func(this.processor);
            }
        }
    }

    _connectToPlayerBackend = () => {
        this.processor.video = document.getElementById('video-player');
        this.setState({ time: this.processor.time, volume: this.processor.
volume })

        this.processor.video.addEventListener('timeupdate', () => this.
setState({ time: this.processor.time }));
        this.processor.video.addEventListener('volumechange', () => this.
setState({ volume: this.processor.volume }));

        for (const event in this.props.events) {
            this.processor.addPreAction(this.props.events[event], event);
        }
    }

    playback = () => {
        this.processor.dispatch('changePlayback', this.processor.paused);
        this.processor.dispatch('setTime', this.processor.time);
        let newIcon = this.processor.paused ? pauseIcon : playIcon;
        this.setState({ playbackIcon: newIcon, playbackState: this.
processor.paused });
    }

    fullscreen = () => {
        if (this.state.isFullscreen) {
            this.leaveFullscreen();
        } else {
            this.enterFullscreen();
        }
    }

    enterFullscreen = () => {
```

```
        const elem = this.playerElemetn.current;
        if (elem.requestFullscreen) {
            elem.requestFullscreen();
        } else if (elem.mozRequestFullScreen) {
            elem.mozRequestFullScreen();
        } else if (elem.webkitRequestFullscreen) {
            elem.webkitRequestFullscreen();
        } else if (elem.msRequestFullscreen) {
            elem.msRequestFullscreen();
        }
    }

    leaveFullscreen = () => {
        if (document.exitFullscreen) {
            document.exitFullscreen();
        } else if (document.mozCancelFullScreen) {
            document.mozCancelFullScreen();
        } else if (document.webkitExitFullscreen) {
            document.webkitExitFullscreen();
        } else if (document.msExitFullscreen) {
            document.msExitFullscreen();
        }
    }

    setTime = (timeValue) => {
        if (!timeValue.isNaN) {
            this.processor.dispatch('setTime', timeValue * this.processor.
duration);
        }
    }

    setVolume = (volumeValue) => {
        this.processor.dispatch('setVolume', volumeValue);
    }

    changeMute = () => {
        this.processor.dispatch('changeMute');
    }


    togglePlaylist = () => {
        this.setState({ playlist: !this.state.playlist });
    }

    toogleChat = (e) => {
        this.setState({ showChat: !this.state.showChat });
    }

    render() {
        return (
            <div className='player-wrapper' id='player' ref={this.
playerElemetn}>
                <Hotkeys keyName='space' onKeyUp={this.playback} />
```

```
                <Hotkeys keyName='c' onKeyUp={this.toogleChat} />
                <div className='player-layer'>
                    <video id='video-player' />
                </div>

                <div className='player-layer mouse-show'>
                    <div id='player-controls' className='player-controls'>
                        <div className='player-controls-row'>
                            <RangeBar value={this.processor.timeProgress}
handle_change={this.setTime} style={{ 'margin-top': '5px', 'margin-bottom'
: '5px' }} />
                        </div>
                        <div className='player-controls-row'>
                            <div onClick={this.playback} className="player-
button player-button-play">
                                {this.state.playbackIcon}
                            </div>
                            <VolumeSlider volume={this.state.volume}
volumeHandler={this.setVolume} toggleMute={this.changeMute} />
                            <TimeIndicator time={this.processor.time}
duration={this.processor.duration} />
                            <div className='player-button' onClick={this.
togglePlaylist}>
                                <MdQueue />
                            </div>
                            <div id='fullscreen-btn' onClick={this.
fullscreen} className="player-button player-button-fullscreen">
                                <MdFullscreen />
                            </div>
                        </div>
                    </div>
                </div>
                <Notifications />
                <PlayList processor={this.processor} visibility={this.state
.playlist} playlistLogic={this.props.playlistLogic} libraryLogic={this.
props.libraryLogic} toogleVideoChooser={this.toogleVideoChooser} />
                {this.state.showChat &&
                    <div className='player-layer'>
                        <Chat hide={this.toogleChat}/>
                    </div>
                }
            </div>
        );
    }
}

export default Player;
```

Данный компонент содержит внутри себя несколько дочерних компонентов: PlayList, VolumeSlider, TimeIndicator, RangeBar. Это сделано с целью оптимизации работы с деревом DOM, так как данные элементы должны регулярно обновляться.

Элементы управления компонента Player обрабатывают определённые события (пауза, перемотка и т.д.). Для непосредственного управления тегом <video> реализован дополнительный класс PlayerBackend (см. листинг 4.2), который содержит логику для управления процессом воспроизведения. При помощи данного класса можно управлять видео напрямую, вызывая событие сразу, или использовать метод dispatch, который занимается обработкой последовательности событий. Данный метод необходим для того, чтобы можно было модифировать стандартную логику тега <video>. При реализации данного метода ыл рассмотрен механизм работы библиотеки Redux, для того чтобы данный метод соответствовал Flux архитектуре. Для метода dispatch доступно следующие действия (actions):

– changePlayback (ставит видео на паузу или возобновляет воспроизведение);
– setTime (перематывает видео);
– setVolume (изменяет громкость видео);
– changeMute (заглушает видео);
– setSrc (устанавливает источник видео).

Метод dispatch позволяет выполнить дополнительное действие перед стандартным. Для этого необходимо после создания объекта данного класса вызвать метод addPreAction, который принимает название действия и функцию, которую необходимо выполнить перед данным действием. Благодаря этому разработанный плеер можно использовать и без синхронизации, так как синхронизирующая логика добавляется при помощи метода addPreAction.

Листинг 4.2 – Класс PlayerBackend

```
class PlayerBackend {
 constructor(video) {
     this._video = video;
     this._preActions = {};
     this._actions = {
         'changePlayback': this.changePlayback,
         'setTime': this.setTime,
         'setVolume': this.setVolume,
         'changeMute': this.changeMute,
         'setSrc': this.setSource
     }
 }

 set video(value) {
     this._video = value;
     this._video.addEventListener('error', (e) => { console.error('error', e)
   });
     this._video.addEventListener('waiting', (e) => { console.error('waiting',
    e) });
```

```
        this._video.addEventListener('stalled', (e) => { console.error('stalled',
    e) });
}

get video() {
    return this._video;
}

addPreAction = (value, action) => {
    if (this._preActions[action] === undefined) {
        this._preActions[action] = []
    }
    this._preActions[action].push(value);
}

resetPreActions() {
    this._preActions = {}
}

resetPostActions() {
    this._preActions = {}
}

// Playback
_play = () => {
    this._video.play().catch((e) => console.error('playback', e));
}

_pause = () => {
    this._video.pause();
}

get paused() {
    return this._video.paused;
}

changePlayback = (paused) => {
    if (paused !== undefined) {
        paused ? this._play() : this._pause();
    } else {
        this.paused ?  this._play() : this._pause();
    }
}

// Volume
set volume(value) {
    this._video.volume = value;
}

setVolume = (value) => this._video.volume = value;

get volume() {
    return this._video.volume;
```

```
    }

    get muted() {
        return this._video.muted;
    }

    changeMute = () => {
        this._video.muted = !this._video.muted;
    }

    // Time
    setTime = (value) => this._video.currentTime = value;
     set time(value) {
        this._video.currentTime = value;
    }

    setTime = (value) => this._video.currentTime = value;

    get time() {
        return this._video ? this._video.currentTime || 0 : 0;
    }

    get timeProgress() {
        return this._video ? this._video.currentTime / this._video.duration || 0
       : 0;
    }

    get duration() {
        return this._video ? this._video.duration || 0 : 0;
    }

    //   Src
    set source(value) {
        this._video.src = value;
    }

    setSource = (value) => this._video.src = value;

    dispatch = (action, value) => {
        let _preActions = this._preActions[action] || [];
        const _actionWrapper = promiseWrapper(this._actions[action], value);

        let promise = new Promise(resolve => resolve());
        for (const _preAction of _preActions) {
            promise = promise.then(promiseWrapper(_preAction, value));
        }

        promise.then(promiseWrapper(this._actions[action], value));
    }
}
```

4.1 Реализация синхронизирующей логики

Так как класс PlayerBackend позволяет выполнять основные операции для управления видео без синхронизации, для выполнения синхронизации необходимо использовать дополнительный класс, который предоставит данную возможность. Реализация основных методов для синхронизации и работы с Firebase представлена в листинге 4.3.

Листинг 4.3 – Методы для взаимодействия с Firebase

```
changePlayback = (roomId, paused) => {
 let room_ref = this.db.collection('rooms').doc(roomId);
 room_ref.update({ paused })
 return room_ref.get().then(() => 'skip');
}

listenPlayback = (roomId, dispatcher) => {
 return this.db.collection('rooms').doc(roomId).onSnapshot(snap => {
   if (snap.data()) {
     dispatcher.firebaseData.paused = snap.data().paused;
     dispatcher.changePlayback(!snap.data().paused);

   }
 });
}

getPlayback = (roomId, dispatcher) => {
 let roomRef = this.db.collection('rooms').doc(roomId);
 return roomRef.get().then(value => dispatcher.changePlayback(!value.data().
   paused));
}

setTime = (roomId, time) => {
 let room_ref = this.db.collection('rooms').doc(roomId);
 room_ref.update({ time })
 return room_ref.get().then(() => 'skip');
}

listenTime = (roomId, dispatcher) => {
 return this.db.collection('rooms').doc(roomId).onSnapshot(snap => {
   if (snap.data()) {
     if (dispatcher.firebaseData.time !== snap.data().time || dispatcher.
   firebaseData.src != snap.data().src) {
       dispatcher.firebaseData.time = snap.data().time;
       dispatcher.setTime(snap.data().time)
     }

   }
 });
}

getTime = (roomId, dispatcher) => {
```

```
  let roomRef = this.db.collection('rooms').doc(roomId);
  return roomRef.get().then(value => dispatcher.setTime(value.data().time || 0)
    );
}

listenSrc = (roomId, dispatcher) => {
  return this.db.collection('rooms').doc(roomId).onSnapshot(snap => {
    if (snap.data()) {
      if (dispatcher.firebaseData.src !== snap.data().src) {
        dispatcher.firebaseData.src = snap.data().src;
        dispatcher.setSource(snap.data().src)
      }
    }
  });
}

setSrc = (roomId, src) => {
  let room_ref = this.db.collection('rooms').doc(roomId);
  let roomInfoRef = this.db.collection('roominfo').doc(roomId);
  room_ref.update({ src })
  roomInfoRef.update({ src });
  return room_ref.get().then(() => 'skip');
}

listenPlaylist = (roomId, controller) => {
  let playlistRef = this.db.collection('playlists').doc(roomId);
  return playlistRef.onSnapshot(snap => {
    if (snap.data()) {
      controller(snap.data().videos)
    }
  })
}

addItemsToPlaylist = (playlistId, items) => {
  let playlistRef = this.db.collection('playlists').doc(playlistId);
  return playlistRef.update({ videos: firebase.firestore.FieldValue.arrayUnion
    (...items) });
}

removeItemsFromPlaylist = (playlistId, items) => {
  let playlistRef = this.db.collection('playlists').doc(playlistId);
  playlistRef.update({
    videos: firebase.firestore.FieldValue.arrayRemove(...items)
  })
}

setPlaylist = (playlistId, playlist) => {
  let playlistRef = this.db.collection('playlists').doc(playlistId);
  return playlistRef.update({ videos: playlist });
}

addUserToRoom = (roomId) => {
  let user = this.auth.currentUser;
```

```
 console.log('user', user)
 if (user) {
   let statsRef = this.db.collection('stats').doc(roomId);
   statsRef.update({
     users: firebase.firestore.FieldValue.arrayUnion({ id: user.uid, status:
   null })
   });
 }
}

removeUserFromRoom = (roomId) => {
 let user = this.auth.currentUser;
 if (user) {
   let statsRef = this.db.collection('stats').doc(roomId);
   statsRef.update({
     users: firebase.firestore.FieldValue.arrayRemove({ id: user.uid, status:
   null })
   });
 }
}

createRoom = (name, usePassword, password) => {
 if (!this.auth.currentUser.isAnonymous) {
   return this.db.collection('roominfo').where('owner', '==', this.auth.
   currentUser.uid).get().then((querySnap) => {
     if (querySnap.size < 1) {
       return this.db.collection('roominfo').add({
         name,
         usePassword,
         password,
         users: [],
         hidden: false,
         src: '',
         owner: this.auth.currentUser.uid,
       }).then(docRef => {
         this.db.collection('rooms').doc(docRef.id).set({ paused: true, src: '
   ', time: 0, })
         this.db.collection('messages').doc(docRef.id).set({ messages: [] });
         this.db.collection('videos').doc(docRef.id).set({ videos: [] });
       });
     } else {
       throw Error("You can't create more rooms")
     }
   })

 }
}

listenRooms = (controller) => {
 return this.db.collection('roominfo').onSnapshot(snap => {
   let rooms = [];
   snap.forEach(doc => {
     rooms.push({ ...doc.data(), id: doc.id });
```

```
    })
    controller(rooms);
  });
}

listenRoom = (roomId, controller) => {
  return this.db.collection('roominfo').doc(roomId).onSnapshot(snap => {
    if (snap.data()) {
      controller(snap.data());
    }
  })
}

getRoom = (roomId) => {
  return this.db.collection('roominfo').doc(roomId).get().then(doc => {
    return doc.data();
  });
}

getUserRooms = (userId) => {
  return this.db.collection('roominfo').where('owner', '==', userId || '').get
    ().then((querySnap) => {
    const rooms = []
    querySnap.forEach(doc => {
      rooms.push(doc.data());
    })
    return rooms;
  })
}

enterRoom = (roomId, password) => {
  let docRef = this.db.collection('roominfo').doc(roomId)
  return docRef.get().then(doc => {
    let result = true;
    if (doc.data().usePassword && password !== doc.data().password) {
      result = false;
    }
    return result;
  })
}

findRoom = (query) => {
  return this.db.collection('roominfo').get().then(snap => {
    let result = [];

    snap.forEach(docRef => {
      if (docRef.data().name.includes(query)) {
        result.push(docRef.data());
      }
    });

    return result;
  })
```

```
}

listenChat = (roomId, controller) => {
 return this.db.collection('messages').doc(roomId).onSnapshot(snap => {
   controller(snap.data().messages);
 });
}
```

Для того, чтобы пользователь получал синхронизирующие данные от других пользователей используются методы: listenPlayback, listenTime и listenSrc. Данные методы инициализируют соединение с Firebase и при помощи методов из класса PlayeBackend изменяют состояние воспроизведения видео в зависимости от текущих данных на сервере. Данные методы гарантируют, что все пользователи комнаты будут получать информацию о состоянии воспроизведения с сервера. Данные методы вызываются при подключении пользователя к комнате. Для отправки запроса об изменении состояния воспроизведения используются методы: changePlayback, setTime и setSrc. Данные методы берут значение текущего состояния видео и оправляют эти значения в Firebase. При помощи идентификатора комнаты данные значения записываются в необходимый документ коллекции rooms. Данные из этих документов затем получают методы listenPlayback, listenTime и listenSrc. Которые затем вызывают необходимые действия класса PlayerBackend. Для вызова методов changePlayback, setTime и setSrc из модуля Firebase используется метод dispatch класса PlayerBackend. Это позволяет выполнять синхронизирующие вызовы только когда пользователь находиться в комнате.

Для того чтобы установить синхронизирующую логику для компонента Player используется компонент обёртка FirebasePlayer (см. листинг 4.4). Данный компонент устанавливает идентификатор комнаты для методов Firebase и затем передаёт их компоненту Player, где те устанавливаются как обработчики действий в классе PlayerBackend.

Листинг 4.4 – Компонент FirebasePlayer

```
class FirebasePlayer extends PureComponent {
    constructor(props) {
        super(props);
        if (props.roomId) {
            this.events = {
                'changePlayback': this.firebasePlayback,
                'setTime': this.firebaseTime,
                'setSrc': this.firebaseSource,
            }
            this.setUp = [this.setUpFirebaseData, this.
firebaseListenSource, this.firebaseListenPlayback, this.firebaseListenTime
]
            this.libraryLogic = {
```

```
                'getLibraryItem': this.firebaseGetLibraryItem
            }
            this.playlistLogic = {
                'listenPlaylist': this.firebaseListenPlaylist,
                'addItems': this.firebaseAddItemsToPlaylist,
                'removeItems': this.firebaseRemoveItemsFromPlaylist,
                'setPlaylist': this.firebaseSetPlaylist,
            }
        }
    }

    componentDidMount() {
        if (this.props.roomId) {
            this.props.firebase.sign_in_anon().then(() => {
                this.props.firebase.addUserToRoom(this.props.roomId);
            })
            window.addEventListener('beforeunload', () => this.props.
firebase.removeUserFromRoom(this.props.roomId));

        }
    }

    componentWillMount() {
        if (this.props.roomId) {
            window.removeEventListener('beforeunload', () => this.props.
firebase.removeUserFromRoom(this.props.roomId));
            this.props.firebase.removeUserFromRoom(this.props.roomId);

        }
    }

    firebaseSetPlaylist = (playlist) => {
        return this.props.firebase.setPlaylist(this.props.roomId, playlist
);
    }

    firebaseRemoveItemsFromPlaylist = (items) => {
        return this.props.firebase.removeItemsFromPlaylist(this.props.
roomId, items);
    }

    firebasePlayback = (value) => {
        return this.props.firebase.changePlayback(this.props.roomId, !
value)
    }

    firebaseListenPlayback = (dispatcher) => {
        return this.props.firebase.listenPlayback(this.props.roomId,
dispatcher);
    }

    fetchPlayback = (dispatcher) => {
        return this.props.firebase.getPlayback(this.props.roomId,
```

```
dispatcher);
    }

    firebaseTime = (value) => {
        return this.props.firebase.setTime(this.props.roomId, value)
    }

    firebaseListenTime = (dispatcher) => {
        return this.props.firebase.listenTime(this.props.roomId,
dispatcher);
    }

    setUpFirebaseData = (dispatcher) => {
        dispatcher.firebaseData = {}
    }

    firebaseSource = (value) => {
        return this.props.firebase.setSrc(this.props.roomId, value);
    }

    firebaseListenSource = (dispatcher) => {
        return this.props.firebase.listenSrc(this.props.roomId, dispatcher
);
    }

    firebaseGetLibraryItem = (id) => {
        return this.props.firebase.getLibraryItem(id).then(value => value)
    }

    firebaseListenPlaylist = (controller) => {
        return this.props.firebase.listenPlaylist(this.props.roomId,
controller)
    }

    firebaseAddItemsToPlaylist = (items) => {
        return this.props.firebase.addItemsToPlaylist(this.props.roomId,
items)
    }

    render() {
        return (
            <div className='f-player'>
                <Player events={this.events} externalSetUp={this.setUp}
libraryLogic={this.libraryLogic} playlistLogic={this.playlistLogic} />
            </div>
        )
    }
}
```

4.2 Клиентское веб-приложение

Клиентское приложение можно разделить на две структурные единицы: основная страница и страница комнаты. Примеры пользовательского интерфейса данных страниц представлены на рисунках 4.1 и 4.2

Рисунок 4.1 – Главная страница приложения

Рисунок 4.2 – Страница комнаты

Основная страница содержит в себе следующий набор функциональных возможностей:

– авторизация пользователей;
– изменение пользовательских настроек;
– отображение доступных комнат;
– поиск комнат;
– предпросмотр комнат;
– переход на страницу комнаты.

Станица плеера имеет следующие функции:

– проверка пароля комнаты;
– сохранение пароля при удачном вводе, чтобы пропустить его ввод в дальнейшем;
– воспроизведение видеоролика;
– остановка видеоролика;
– перемотка видеоролика;
– изменение громкости видеоролика;
– открытие видеоролика в полноэкранном режиме;
– отображение уведомлений о текущих действиях плеера;
– добавление видеоролика в список воспроизведения;
– добавление нескольких роликов в список воспроизведения;
– удаление ролика из списка воспроизведения;
– удаление всех роликов из списка воспроизведения;
– выбор следующего ролика для воспроизведения из списка воспроизведения;
– автоматическое включение следующего ролика из списка воспроизведения;
– синхронизация времени ролика между пользователями текущей комнаты;
– синхронизация состояния воспроизведения между пользователями текущей комнаты;
– синхронизация источника текущего видеоролика между пользователями текущей комнаты;
– отправка и получение сообщений в чате комнаты.

4.3 Учётная запись пользователя

Для работы пользователя с веб-приложением необходимо иметь учетную запись. Каждый пользователь при первом открытии страницы приложения автоматически получает учетную запись без необходимости ввода каких-либо данных. Данная учетная запись имеет статус анонимной. Это сделано

для того чтобы любой пользователь мог сразу начать пользоваться сервисом. Пользователи с анонимной учетной записью имеют ряд ограничений:

– невозможно создать собственную комнату;
– невозможно сменить имя пользователя;

Для того чтобы избавиться от ограничений пользователю необходимо создать собственную учетную запись. Для этого ему необходимо предоставить адрес электронной почты, выбрать имя пользователя и пароль (для подтверждения пароля пользователь должен ввести его два раза). В дальнейшем пользователь при посещении сайта будет сразу использовать созданную им учетную запись. Для сохранения пользовательской информации в рамках текущей сессии, она записывается в хранилище состояний, предоставленный JavaScript библиотекой Redux.

4.4 Список комнат

Каждый пользователь имеет возможность просмотра списка комнат на главной странице. Данные для данного списка загружаются из базы данных и автоматически обновляются при их изменении. Также пользователь имеет возможность производить поиск комнат, для этого имеется строка поиска в верхней части страницы.

Список комнат представлен в виде набора плиток. На данных плитках находится основная информация о данной комнате: название, защищённость паролем, а также окно предпросмотра текущего видеоролика. При нажатии на плитку комнаты пользователь перенаправляется на страницу данной комнаты.

4.5 Комната

Перед тем как получить доступ к комнате происходит дополнительная проверка. Если комната защищена паролем, то пользователь сначала попадает на страницу с формой для ввода пароля. В случае ввода неправильного пароля пользователь получает уведомление об ошибке и ему предлагается ввести пароля снова. При вводе верного пароля, он сохраняется в локальном хранилище браузера для повторного использования и пользователь получает доступ к комнате.

4.6 Варианты использования

Далее будут рассмотрены основные варианты использования разработанного веб-приложения.

4.6.1 ВИ Создание учётной записи

Описание ВИ: Пользователь имеет возможность при желании создать личную учётную запись.

Предусловия основного потока действий нет.

Основной поток действий:

– пользователь нажимает кнопку "Create account";

– пользователь вводит электронную почту, имя пользователя, пароль и подтверждение пароля.

Ограничения: различные ограничения, установленные провайдером аутентификации Firebase.

4.6.2 ВИ Использование своей учётной записи

Описание ВИ: Пользователь имеет возможность использовать личную учётную запись.

Предусловия основного потока действий: у пользователя имеется учётная запись см. 4.6.1.

Основной поток действий:

– пользователь нажимает кнопку "Create account";

– пользователь нажимает кнопку "Existing user";

– пользователь вводит электронную почту и пароль.

Ограничения: различные ограничения, установленные провайдером аутентификации Firebase.

4.6.3 ВИ Создание комнаты

Описание ВИ: Пользователь имеет возможность создать комнату для совместного просмотра.

Предусловия основного потока действий: пользователь должен быть авторизован.

Основной поток действий:

– пользователь нажимает кнопку "Create";

– пользователь вводит название комнаты и пароль;

– клиент получает список комнат и обновляет интерфейс.

Ограничения: имя комнаты должно быть непустой строкой.

4.6.4 ВИ Поиск комнаты

Описание ВИ: Пользователь имеет возможность искать комнату по её имени для совместного просмотра.

Предусловия основного потока действий: Комната существует. Для создания комнаты см. 4.6.3.

Основной поток действий:

– пользователь начинает писать название комнаты;

– при изменении текста в строке поиска система делает поиск по имени комнаты;

– клиент получает список комнат и обновляет свой интерфейс.

Ограничения: имя комнаты должно быть непустой строкой.

4.6.5 ВИ Подключение к комнате

Описание ВИ: Пользователь имеет возможность присоединиться к комнате для совместного просмотра.

Предусловия основного потока действий: комната должна существовать см. 4.6.3.

Основной поток действий:

– пользователь нажимает кнопку плитку комнаты из списка;

– пользователь перенаправляется на страницу комнаты.

Ограничения: при наличии у комнаты пароля пользователь должен ввести его, для подключения.

4.6.6 ВИ Управление видеоплеером

Описание ВИ: Пользователь имеет возможность управлять видеоплеером комнаты во время совместного просмотра.

Предусловия основного потока действий: комната должна существовать см. 4.6.3, Пользователь присоединен к комнате см. 4.6.5.

Основной поток действий:

– Пользователь имеет следующие опций для управления плеером: запуск и остановка воспроизведения видео, перемотка, настройка громкости, открытие полноэкранного режима.

Ограничения: Для взаимодействия с плеером должно быть выбрано видео для воспроизведения см. 4.6.7.

4.6.7 ВИ Добавление видео по ссылке

Описание ВИ: Пользователь имеет возможность добавить видео в список воспроизведения комнаты для совместного просмотра.

Предусловия основного потока действий: комната должна существовать см. 4.6.3, Пользователь присоединен к комнате см. 4.6.5.

Основной поток действий:

– пользователь открывает меню списка воспроизведения;
– пользователь нажимает кнопку "Add";
– пользователь вводит ссылку на ролик в строку поиска;
– клиент получает список видео и обновляет интерфейс;
– пользователь нажимает кнопку "Add"у нужного видеоролика.

Ограничения: ссылка должна поддерживаться приложением.

4.6.8 ВИ Удаление видео из списка воспроизведения

Описание ВИ: Пользователь имеет возможность удалить видео в список воспроизведения комнаты для совместного просмотра.

Предусловия основного потока действий: комната должна существовать см. 4.6.3, Пользователь присоединен к комнате см. 4.6.5, в списке воспроизведения должны быть видеоролики см. 4.6.7.

Основной поток действий:

– пользователь открывает меню списка воспроизведения;
– пользователь нажимает кнопку "крестик"у необходимого видеоролика.

Альтернативный поток действий:

– пользователь открывает меню списка воспроизведения;
– пользователь нажимает кнопку "Reset".

Ограничений нет.

4.6.9 ВИ Использование чата

Описание ВИ: Пользователь имеет возможность общаться с другими пользователями, используя текстовый чат.

Предусловия основного потока действий: комната должна существовать см. 4.6.3, Пользователь присоединен к комнате см. 4.6.5.

Основной поток действий:

– пользователь открывает текстовый чат;
– пользователь вводит сообщение в текстовое поле;
– пользователь нажимает кнопку "Send"или клавишу Enter.

Ограничения: сообщение должно быть непустой строкой.

4.7 Тестирование приложения

Для проведения тестирования веб-приложения использовалось ПО из проекта Selenium. В рамках данного проекта разработан набор различных программ, которые помогают автоматизировать процесс тестирования веб-приложений. Для данного проекта использовался Selenium WebDriver - универсальный интерфейс для взаимодействия с драйвером браузера, веб-браузер Firefox и драйвер geckodriver. Для проведения тестов использовалась библиотека для языка Python и расширение для браузера Firefox — Selenium IDE.

Также было проведено ручное тестирование механизма синхронизации, так как при помощи Selenium их протестировать затруднительно. В рамках ручного тестирования было проведено дымовое тестирование возможностей видеоплеера и механизмов синхронизации.

5 ТЕХНИКО-ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ

5.1 Описание функций, назначения и потенциальных пользователей ПО

Разрабатываемый в дипломном проекте программный модуль предназначен для автоматизации организации и проведения совместного просмотра видео материалов по сети интернет.

Основными функциями разрабатываемого программного продукта являются:

– возможность использовать сервис неавторизованным пользователям;

– возможность регистрации профиля для доступа к дополнительным функциям;

– организация “комнат” для просмотра видео;

– поиск комнат по их атрибутам: название комнаты, название текущего видео ролика;

– возможность присоединиться к комнате для просмотра видео;

– управление воспроизведением видео: пауза, продолжить, перемотка, изменение громкости, переключение в полноэкранный режим;

– синхронизация видео между всеми пользователями в комнате (при нажатии паузы или перемотки, она происходит у всех пользователей в комнате);

– возможность писать сообщения в текстовый чат для текущей комнаты;

Данный программный продукт предполагает использование базы данных для хранения данных пользователей, информации и видео материалах и комнатах.

При разработке использовались язык программирования JavaScript и фреймворка ReactJS для создания клиентской части приложения и пользовательского интерфейса. Также был использован сервис Google Firebase, для базы данных текущих пользовательских сессий и язык программирования Python для написания серверной логики.

Разрабатываемый программный продукт адресован для пользователей, которые из-за различных обстоятельств, например пандемии, не имеют возможности личной встречи для просмотра кино с друзьями и близкими.

5.2 Расчет затрат на разработку ПО

Для разработки данного программного модуля необходимы следующие специалисты: ведущий программист, программист, дизайнер и тестировщик. Ведущий программист занимается. Трудоёмкость работ, вид работ и ставки представлены в таблице 5.1.

Затраты на основную заработную плату команды разработчиков. Расчет основной заработной платы участников команды осуществляется по формуле:

$$З_О = \sum_{i=1}^{n} З_i \cdot t_i \,, \tag{5.1}$$

где n — количество исполнителей, занятых разработкой конкретного ПО;
$З_i$ — часовая заработная плата i-го исполнителя (р.);
t_i — трудоемкость работ, выполняемых i-м исполнителем (ч).

Затраты на дополнительную заработную плату команды разработчиков определяется по формуле:

$$З_Д = \frac{З_О \cdot Н_Д}{100\%} \,, \tag{5.2}$$

где $З_О$ — затраты на основную заработную плату, (р.);
$З_i$ — норматив дополнительной заработной платы, % (НД = 40%).

$$З_Д = \frac{9571{,}2 \cdot 0{,}4}{100} = 38{,}28 \text{ р.} \tag{5.3}$$

Отчисления на социальные нужды (в фонд социальной защиты населения и на обязательное страхование) определяются в соответствии с действующими законодательными актами по формуле:

$$Р_{СОЦ} = \frac{(З_О + З_Д) \cdot Н_{СОЦ}}{100} \,, \tag{5.4}$$

где $Н_{СОЦ}$ — норматив отчислений на социальные нужды, % (НСОЦ = 35%).

$$Р_{СОЦ} = \frac{(9571{,}2 + 38{,}28) \cdot 0{,}35}{100} = 33{,}63 \text{ р.} \tag{5.5}$$

Таблица 5.1 – Расчет затрат на основную заработную плату команды разработчиков

№	Участник команды	Вид выполняемой работы	Часовая тарифная ставка, р.	Трудоемкость работ, ч	Зарплата по тарифу, р.
1	2	3	4	5	6
1	Ведущий программист	Разработка, архитектуры приложения, разработка функцио-нальности приложения	6,84	640	4377,6
2	Программист	Разработка функцио-нальности приложения, поддержка	4,86	640	3110,4
3	Дизайнер	Разработка дизайна пользователь-ского интерфейса	2,67	160	427,2
4	Тестировщик	Тестирование	3,45	480	1656
Итого затраты на основную заработную плату разработчиков					9571,2

Прочие затраты включаются в себестоимость разработки ПО в процентах от затрат на основную заработную плату команды разработчиков по формуле:

$$З_{ПЗ} = \frac{З_О \cdot Н_{ПЗ}}{100}, \tag{5.6}$$

где $Н_{ПЗ}$ — норматив прочих затрат, % (НПЗ = 150%).

$$З_{ПЗ} = \frac{9571{,}2 \cdot 1{,}5}{100} = 143{,}57 \text{ р.} \tag{5.7}$$

Полная сумма затрат на разработку программного обеспечения находится путем суммирования всех рассчитанных статей затрат. Итоговые данные представлены в таблице 5.2.

Таблица 5.2 – Затраты на разработку программного обеспечения

Статья затрат	Сумма, р.
Основная заработная плата команды разработчиков	9571,2
Дополнительная заработная плата команды разработчиков	38,28
Отчисления на социальные нужды	33,63
Прочие затраты	143,57
Общая сумма затрат на разработку	9786,68

Общая сумма затрат на разработку программного модуля составит 9786,68 рублей.

5.3 Экономический эффект при разработке ПО

Экономический эффект организации-разработчика программного обеспечения в данном случае представляет собой прибыль (чистая прибыль) от его продажи множеству потребителей. Прибыль рассчитывается по формулам:

$$\text{П}_\text{Ч} = \text{П} - \frac{\text{П} \cdot \text{Н}_\text{П}}{100}, \tag{5.8}$$

$$\text{П} = \text{Ц} \cdot N - \text{НДС} - \text{З}_\text{Р}, \tag{5.9}$$

$$\text{НДС} = \frac{\text{Ц} \cdot N \cdot \text{Н}_\text{ДС}}{100\% + \text{Н}_\text{ДС}}. \tag{5.10}$$

где Ц — цена реализации ПО заказчику (р.);
НДС — сумма налога на добавленную стоимость (р.);
$\text{Н}_\text{ДС}$ — ставка налога на добавленную стоимость согласно действующему законодательству, % (Ндс = 20%);
$\text{Н}_\text{П}$ — ставка налога на прибыль, % (Нп =18%);
$\text{З}_\text{Р}$ — Общая сумма затрат на разработку программного модуля(р.).

Основной доход планируется получать с продажи подписки, которая дает пользователю дополнительные возможности. Стоимость ежемесячной

подписки 10 р. Тогда при 150 подписчиках в течении года, что является пессимистическим прогнозом, имеем:

$$\text{НДС} = \frac{10 \cdot 150 \cdot 0{,}2}{1{,}2} = 250 \text{ р.} \tag{5.11}$$

$$\text{П} = (10 \cdot 12) \cdot 150 - 250 - 9786{,}68 = 7963{,}32 \text{ р.} \tag{5.12}$$

$$\text{П}_\text{ч} = \frac{7963{,}32 - (7963{,}32 \cdot 0{,}18)}{100} = 7948{,}98 \text{ р.} \tag{5.13}$$

Получаем 7948,98 рублей чистой прибыли за год по пессимистическим прогнозам. Вычислим уровень рентабельности по формуле:

$$\text{У}_\text{Р} = \frac{\text{П}_\text{ч}}{\text{З}_\text{Р}} \cdot 100\%\,, \tag{5.14}$$

$$\text{У}_\text{Р} = \frac{7948{,}98}{9786{,}68} \cdot 100\% = 81\%. \tag{5.15}$$

Уровень рентабельности затрат равен 81% при планируемой реализации в течении одного года, что выше текущей процентной ставки по банковским депозитным вкладам (13%, с поправкой на риск), следовательно разработка данного программного модуля является оправданной.

$$\text{Т}_\text{ОК} = \frac{9786{,}68}{7948{,}98} = 1{,}5 \text{ года.} \tag{5.16}$$

При сохранении данного уровня прибыли разработка приложения окупиться в течение полутора лет.

5.4 Заключение

В результате технико-экономического обоснования сервиса онлайн кинотеатра были произведены расчеты затрат на разработку, предполагаемой прибыли, рентабельности затрат и срока окупаемости. В результате были получены следующие значения:

– Затраты на разработку составляют 9786,68 рублей

– Пессимистический уровень чистой месячной прибыли составляет 7948,98 рублей

– Уровень рентабельность затрат на разработку равен 81%

– Окупаемость разработки в течение полутора лет

Эти данные говорят что разработка данного программного модуля является

оправданной. Расчёты рентабельности затрат были произведены с учётом изначальной заинтересованности пользователей, следовательно показатель рентабельности затрат может значительно превышать полученное значение. Особенно велика вероятность повышенного спроса в текущих реалиях, так как из-за пандемии COVID-19, люди вынуждены находиться в самоизоляции, а данный программный модуль позволит им решить проблему совместного просмотра кино.

ЗАКЛЮЧЕНИЕ

В рамках данного дипломного проекта был рассмотрен вопрос синхронизации воспроизведения видео по сети интернет, рассмотрены сервисы, которые предоставляют подобную функциональность, и различные способы синхронизации. Также было разработано веб-приложение, которое предоставляет пользовательский интерфейс для создания комнат и синхронизации процесса воспроизведения видео. Помимо это были реализованы дополнительные возможности: чат, уведомления и список воспроизведения.

При разработке приложения использовались современные технологии, методики разработки и архитектурные решения: код приложения разбит на отдельные модули (компоненты), которые выполняют задачи необходимые только для данного компонента. Был изучен и использован сервис Firebase, в частности Firebase Firestore, который позволяет повысить интерактивность приложения и ускорить разработку, минимизируя затраты. Также были изучены и применены различные средства для взаимодействия и получения видео с таких сторонних видеосервисов, как YouTube и Vimeo.

В результате цель дипломного проекты была дастигнута. Было создано программное обеспечение, выполняющее минимально необходимый набор возможностей. Однако было оставлено множество мест для улучшения разработанного ПО в дальнейшем. К таким улучшениям можно отнести: поддержку прямых трансляций, улучшение при работе с медленным интернет соединением, расширением возможностей чата (отправка изображений, видео и gif-файлов) и расширение возможностей по администрированию комнаты. В дальнейшем планируется улучшить данное приложение, введя дополнительные возможности и улучшая текущие, а также добавить полноценную поддержку мобильных платформ.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Современный учебник JavaScript [Электронный ресурс]. — Электронные данные. — Режим доступа: http://learn.javascript.ru/. — Дата доступа: 08.05.2013.

2. Документация React [Электронный ресурс]. — Электронные данные. — Режим доступа: https://ru.reactjs.org/docs/getting-started.html. — Дата доступа: 08.05.2013.

3. Micha Gorelick, Ian Ozsvald. High Performance Python / Ian Ozsvald Micha Gorelick. — O'Reilly Media, 2014. — 370 с.

4. David Beazley, Brian K. Jones. Python Cookbook / Brian K. Jones David Beazley. — O'Reilly Media, 2013. — 706 с.

5. Redux API – Redux Docs [Электронный ресурс]. — Электронные данные. — Режим доступа: https://react-redux.js.org/api/connect. — Дата доступа: 08.05.2013.

6. Флэнаган, Дэвид. JavaScript. Подробное руководство / Дэвид Флэнаган. — Символ, 2018. — 1080 с.

7. Крокфорд, Д. JavaScript: сильные стороны / Д. Крокфорд. — Питер, 2012. — 176 с.

8. Chris Aquino, Todd Gandee. Front-End Web Development / Todd Gandee Chris Aquino. — Big Nerd Ranch Guides, 2016. — 478 с.

9. Лоусон Брюс, Шарп Реми. Изучаем HTML5. Библиотека специалиста / Шарп Реми Лоусон Брюс. — Питер, 2012. — 304 с.

10. W3Schools Online Web Tutorials [Электронный ресурс]. — Электронные данные. — Режим доступа: https://www.w3schools.com/. — Дата доступа: 08.05.2013.

11. Веб-документайия MDN [Электронный ресурс]. — Электронные данные. — Режим доступа: https://developer.mozilla.org/. — Дата доступа: 08.05.2013.

12. Mardan, Azat. React Quickly / Azat Mardan. — MANNING, 2019. — 336 с.

13. React State and Lifecycle [Электронный ресурс]. — Электронные данные. — Режим доступа: https://reactjs.org/docs/state-and-lifecycle.html. — Дата доступа: 08.05.2013.

14. Sheriff, Paul D. Designing for Web or Desktop? [Электронный ресурс]. — Электронные данные. — Режим доступа: https://nurelm.com/webbased-vs-desktop-software/. — Дата доступа: 08.05.2013.

15. Bychkov, Dmitriy. Desktop vs. WebApplications: A Deeper Lookand

Comparison [Электронный ресурс]. — Электронные данные. — Режим доступа: http://www.seguetech.com/desktop-vs-web-applications/. — Дата доступа: 08.05.2013.

16. Downey, Allen B. Think Python / Allen B. Downey. — O'Reilly Media, 2015. — 292 с.

17. Горовой В.Г. Грицай А.В., Пархименко В.А. Экономическое обоснование проекта по разработке программного обеспечения / Пархименко В.А. Горовой В.Г., Грицай А.В. — кафедра экономики БГУИР, 2018. — 12 с.

ПРИЛОЖЕНИЕ А
Текст программы

```
import React, { Component } from 'react';
import { MdPlayArrow, MdPause, MdFullscreen, MdQueue } from 'react-icons/
md';
import './player.scss';
import TimeIndicator from './TimeIndicator';
import RangeBar from './Slider';
import Hotkeys from 'react-hot-keys';
import VolumeSlider from './VolumeSlider';
import { PlayerBackend } from '../PlayerBackend/default';
import PlayList from './PlayList';
import Notifications from './Notifications';
import Chat from './Chat';

const pauseIcon = <MdPause />;
const playIcon = <MdPlayArrow />;

class Player extends Component {
    constructor(props) {
        super(props);
        this.processor = new PlayerBackend();
        this.playerElemetn = React.createRef();
        this.state = {
            isFullscreen: false,
            playbackIcon: playIcon,
            playbackState: false,
            playlist: false,
            showVideoChooser: false,
            showChat: false,
        }
    }

    componentDidMount() {
        this.vid_title = document.getElementById('video-title');
        this._connectToPlayerBackend();
        document.onfullscreenchange = () => this.setState({ isFullscreen: !
this.state.isFullscreen });
        if (this.props.externalSetUp) {
            for (const func of this.props.externalSetUp) {
                func(this.processor);
            }
        }
    }

    _connectToPlayerBackend = () => {
        this.processor.video = document.getElementById('video-player');
        this.setState({ time: this.processor.time, volume: this.processor.
volume })
```

```
        this.processor.video.addEventListener('timeupdate', () => this.
setState({ time: this.processor.time }));
        this.processor.video.addEventListener('volumechange', () => this.
setState({ volume: this.processor.volume }));

        for (const event in this.props.events) {
            this.processor.addPreAction(this.props.events[event], event);
        }
    }

    playback = () => {
        this.processor.dispatch('changePlayback', this.processor.paused);
        this.processor.dispatch('setTime', this.processor.time);
        let newIcon = this.processor.paused ? pauseIcon : playIcon;
        this.setState({ playbackIcon: newIcon, playbackState: this.
processor.paused });
    }

    fullscreen = () => {
        if (this.state.isFullscreen) {
            this.leaveFullscreen();
        } else {
            this.enterFullscreen();
        }
    }

    enterFullscreen = () => {
        const elem = this.playerElemetn.current;
        if (elem.requestFullscreen) {
            elem.requestFullscreen();
        } else if (elem.mozRequestFullScreen) {
            elem.mozRequestFullScreen();
        } else if (elem.webkitRequestFullscreen) {
            elem.webkitRequestFullscreen();
        } else if (elem.msRequestFullscreen) {
            elem.msRequestFullscreen();
        }
    }

    leaveFullscreen = () => {
        if (document.exitFullscreen) {
            document.exitFullscreen();
        } else if (document.mozCancelFullScreen) {
            document.mozCancelFullScreen();
        } else if (document.webkitExitFullscreen) {
            document.webkitExitFullscreen();
        } else if (document.msExitFullscreen) {
            document.msExitFullscreen();
        }
    }

    setTime = (timeValue) => {
        if (!timeValue.isNaN) {
```

```
            this.processor.dispatch('setTime', timeValue * this.processor.
duration);
        }
    }

    setVolume = (volumeValue) => {
        this.processor.dispatch('setVolume', volumeValue);
    }

    changeMute = () => {
        this.processor.dispatch('changeMute');
    }


    togglePlaylist = () => {
        this.setState({ playlist: !this.state.playlist });
    }

    toogleChat = (e) => {
        this.setState({ showChat: !this.state.showChat });
    }

    render() {
        return (
            <div className='player-wrapper' id='player' ref={this.
playerElemetn}>
                <Hotkeys keyName='space' onKeyUp={this.playback} />
                <Hotkeys keyName='c' onKeyUp={this.toogleChat} />
                <div className='player-layer'>
                    <video id='video-player' />
                </div>

                <div className='player-layer mouse-show'>
                    <div id='player-controls' className='player-controls'>
                        <div className='player-controls-row'>
                            <RangeBar value={this.processor.timeProgress}
handle_change={this.setTime} style={{ 'margin-top': '5px', 'margin-bottom'
: '5px' }} />
                        </div>
                        <div className='player-controls-row'>
                            <div onClick={this.playback} className="player-
button player-button-play">
                                {this.state.playbackIcon}
                            </div>
                            <VolumeSlider volume={this.state.volume}
volumeHandler={this.setVolume} toggleMute={this.changeMute} />
                            <TimeIndicator time={this.processor.time}
duration={this.processor.duration} />
                            <div className='player-button' onClick={this.
togglePlaylist}>
                                <MdQueue />
                            </div>
                            <div id='fullscreen-btn' onClick={this.
```

```
  fullscreen} className="player-button player-button-fullscreen">
                                    <MdFullscreen />
                                </div>
                            </div>
                        </div>
                    </div>
                    <Notifications />
                    <PlayList processor={this.processor} visibility={this.state
  .playlist} playlistLogic={this.props.playlistLogic} libraryLogic={this.
  props.libraryLogic} toogleVideoChooser={this.toogleVideoChooser} />
                    {this.state.showChat &&
                        <div className='player-layer'>
                            <Chat hide={this.toogleChat}/>
                        </div>
                    }
                </div>
            );
        }
    }

    export default Player;

    class PlayerBackend {
constructor(video) {
    this._video = video;
    this._preActions = {};
    this._actions = {
        'changePlayback': this.changePlayback,
        'setTime': this.setTime,
        'setVolume': this.setVolume,
        'changeMute': this.changeMute,
        'setSrc': this.setSource
    }
}

set video(value) {
    this._video = value;
    this._video.addEventListener('error', (e) => { console.error('error', e)
  });
    this._video.addEventListener('waiting', (e) => { console.error('waiting',
   e) });
    this._video.addEventListener('stalled', (e) => { console.error('stalled',
   e) });
}

get video() {
    return this._video;
}

addPreAction = (value, action) => {
    if (this._preActions[action] === undefined) {
        this._preActions[action] = []
    }
```

```
        this._preActions[action].push(value);
    }

    resetPreActions() {
        this._preActions = {}
    }

    resetPostActions() {
        this._preActions = {}
    }

    // Playback
    _play = () => {
        this._video.play().catch((e) => console.error('playback', e));
    }

    _pause = () => {
        this._video.pause();
    }

    get paused() {
        return this._video.paused;
    }

    changePlayback = (paused) => {
        if (paused !== undefined) {
            paused ? this._play() : this._pause();
        } else {
            this.paused ?  this._play() : this._pause();
        }
    }

    // Volume
    set volume(value) {
        this._video.volume = value;
    }

    setVolume = (value) => this._video.volume = value;

    get volume() {
        return this._video.volume;
    }

    get muted() {
        return this._video.muted;
    }

    changeMute = () => {
        this._video.muted = !this._video.muted;
    }

    // Time
    setTime = (value) => this._video.currentTime = value;
```

```
  set time(value) {
      this._video.currentTime = value;
  }

  setTime = (value) => this._video.currentTime = value;

  get time() {
      return this._video ? this._video.currentTime || 0 : 0;
  }

  get timeProgress() {
      return this._video ? this._video.currentTime / this._video.duration || 0
    : 0;
  }

  get duration() {
      return this._video ? this._video.duration || 0 : 0;
  }

  //  Src
  set source(value) {
      this._video.src = value;
  }

  setSource = (value) => this._video.src = value;

  dispatch = (action, value) => {
      let _preActions = this._preActions[action] || [];
      const _actionWrapper = promiseWrapper(this._actions[action], value);

      let promise = new Promise(resolve => resolve());
      for (const _preAction of _preActions) {
          promise = promise.then(promiseWrapper(_preAction, value));
      }

      promise.then(promiseWrapper(this._actions[action], value));
  }
}

changePlayback = (roomId, paused) => {
  let room_ref = this.db.collection('rooms').doc(roomId);
  room_ref.update({ paused })
  return room_ref.get().then(() => 'skip');
}

listenPlayback = (roomId, dispatcher) => {
  return this.db.collection('rooms').doc(roomId).onSnapshot(snap => {
    if (snap.data()) {
      dispatcher.firebaseData.paused = snap.data().paused;
      dispatcher.changePlayback(!snap.data().paused);

    }
  });
```

```
}

getPlayback = (roomId, dispatcher) => {
 let roomRef = this.db.collection('rooms').doc(roomId);
 return roomRef.get().then(value => dispatcher.changePlayback(!value.data().
    paused));
}

setTime = (roomId, time) => {
 let room_ref = this.db.collection('rooms').doc(roomId);
 room_ref.update({ time })
 return room_ref.get().then(() => 'skip');
}

listenTime = (roomId, dispatcher) => {
 return this.db.collection('rooms').doc(roomId).onSnapshot(snap => {
   if (snap.data()) {
     if (dispatcher.firebaseData.time !== snap.data().time || dispatcher.
    firebaseData.src != snap.data().src) {
       dispatcher.firebaseData.time = snap.data().time;
       dispatcher.setTime(snap.data().time)
     }

   }
 });
}

getTime = (roomId, dispatcher) => {
 let roomRef = this.db.collection('rooms').doc(roomId);
 return roomRef.get().then(value => dispatcher.setTime(value.data().time || 0)
    );
}

listenSrc = (roomId, dispatcher) => {
 return this.db.collection('rooms').doc(roomId).onSnapshot(snap => {
   if (snap.data()) {
     if (dispatcher.firebaseData.src !== snap.data().src) {
       dispatcher.firebaseData.src = snap.data().src;
       dispatcher.setSource(snap.data().src)
     }
   }
 });
}

setSrc = (roomId, src) => {
 let room_ref = this.db.collection('rooms').doc(roomId);
 let roomInfoRef = this.db.collection('roominfo').doc(roomId);
 room_ref.update({ src })
 roomInfoRef.update({ src });
 return room_ref.get().then(() => 'skip');
}

listenPlaylist = (roomId, controller) => {
```

```
 let playlistRef = this.db.collection('playlists').doc(roomId);
 return playlistRef.onSnapshot(snap => {
   if (snap.data()) {
     controller(snap.data().videos)
   }
 })
}

addItemsToPlaylist = (playlistId, items) => {
 let playlistRef = this.db.collection('playlists').doc(playlistId);
 return playlistRef.update({ videos: firebase.firestore.FieldValue.arrayUnion
   (...items) });
}

removeItemsFromPlaylist = (playlistId, items) => {
 let playlistRef = this.db.collection('playlists').doc(playlistId);
 playlistRef.update({
   videos: firebase.firestore.FieldValue.arrayRemove(...items)
 })
}

setPlaylist = (playlistId, playlist) => {
 let playlistRef = this.db.collection('playlists').doc(playlistId);
 return playlistRef.update({ videos: playlist });
}

addUserToRoom = (roomId) => {
 let user = this.auth.currentUser;
 console.log('user', user)
 if (user) {
   let statsRef = this.db.collection('stats').doc(roomId);
   statsRef.update({
     users: firebase.firestore.FieldValue.arrayUnion({ id: user.uid, status:
   null })
   });
 }
}

removeUserFromRoom = (roomId) => {
 let user = this.auth.currentUser;
 if (user) {
   let statsRef = this.db.collection('stats').doc(roomId);
   statsRef.update({
     users: firebase.firestore.FieldValue.arrayRemove({ id: user.uid, status:
   null })
   });
 }
}

createRoom = (name, usePassword, password) => {
 if (!this.auth.currentUser.isAnonymous) {
   return this.db.collection('roominfo').where('owner', '==', this.auth.
   currentUser.uid).get().then((querySnap) => {
```

```
      if (querySnap.size < 1) {
        return this.db.collection('roominfo').add({
          name,
          usePassword,
          password,
          users: [],
          hidden: false,
          src: '',
          owner: this.auth.currentUser.uid,
        }).then(docRef => {
          this.db.collection('rooms').doc(docRef.id).set({ paused: true, src: '
    ', time: 0, })
          this.db.collection('messages').doc(docRef.id).set({ messages: [] });
          this.db.collection('videos').doc(docRef.id).set({ videos: [] });
        });
      } else {
        throw Error("You can't create more rooms")
      }
    })

  }
}

listenRooms = (controller) => {
  return this.db.collection('roominfo').onSnapshot(snap => {
    let rooms = [];
    snap.forEach(doc => {
      rooms.push({ ...doc.data(), id: doc.id });
    })
    controller(rooms);
  });
}

listenRoom = (roomId, controller) => {
  return this.db.collection('roominfo').doc(roomId).onSnapshot(snap => {
    if (snap.data()) {
      controller(snap.data());
    }
  })
}

getRoom = (roomId) => {
  return this.db.collection('roominfo').doc(roomId).get().then(doc => {
    return doc.data();
  });
}

getUserRooms = (userId) => {
  return this.db.collection('roominfo').where('owner', '==', userId || '').get
    ().then((querySnap) => {
    const rooms = []
    querySnap.forEach(doc => {
      rooms.push(doc.data());
```

```
    })
    return rooms;
  })
}

enterRoom = (roomId, password) => {
  let docRef = this.db.collection('roominfo').doc(roomId)
  return docRef.get().then(doc => {
    let result = true;
    if (doc.data().usePassword && password !== doc.data().password) {
      result = false;
    }
    return result;
  })
}

findRoom = (query) => {
  return this.db.collection('roominfo').get().then(snap => {
    let result = [];

    snap.forEach(docRef => {
      if (docRef.data().name.includes(query)) {
        result.push(docRef.data());
      }
    });

    return result;
  })
}

listenChat = (roomId, controller) => {
  return this.db.collection('messages').doc(roomId).onSnapshot(snap => {
    controller(snap.data().messages);
  });
}

class FirebasePlayer extends PureComponent {
    constructor(props) {
        super(props);
        if (props.roomId) {
            this.events = {
                'changePlayback': this.firebasePlayback,
                'setTime': this.firebaseTime,
                'setSrc': this.firebaseSource,
            }
            this.setUp = [this.setUpFirebaseData, this.firebaseListenSource,
    this.firebaseListenPlayback, this.firebaseListenTime]
            this.libraryLogic = {
                'getLibraryItem': this.firebaseGetLibraryItem
            }
            this.playlistLogic = {
                'listenPlaylist': this.firebaseListenPlaylist,
                'addItems': this.firebaseAddItemsToPlaylist,
```

```
                'removeItems': this.firebaseRemoveItemsFromPlaylist,
                'setPlaylist': this.firebaseSetPlaylist,
            }
        }
    }

    componentDidMount() {
        if (this.props.roomId) {
            this.props.firebase.sign_in_anon().then(() => {
                this.props.firebase.addUserToRoom(this.props.roomId);
            })
            window.addEventListener('beforeunload', () => this.props.firebase.
removeUserFromRoom(this.props.roomId));

        }
    }

    componentWillMount() {
        if (this.props.roomId) {
            window.removeEventListener('beforeunload', () => this.props.
firebase.removeUserFromRoom(this.props.roomId));
            this.props.firebase.removeUserFromRoom(this.props.roomId);

        }
    }

    firebaseSetPlaylist = (playlist) => {
        return this.props.firebase.setPlaylist(this.props.roomId, playlist);
    }

    firebaseRemoveItemsFromPlaylist = (items) => {
        return this.props.firebase.removeItemsFromPlaylist(this.props.roomId,
items);
    }

    firebasePlayback = (value) => {
        return this.props.firebase.changePlayback(this.props.roomId, !value)
    }

    firebaseListenPlayback = (dispatcher) => {
        return this.props.firebase.listenPlayback(this.props.roomId,
dispatcher);
    }

    fetchPlayback = (dispatcher) => {
        return this.props.firebase.getPlayback(this.props.roomId, dispatcher);
    }

    firebaseTime = (value) => {
        return this.props.firebase.setTime(this.props.roomId, value)
    }

    firebaseListenTime = (dispatcher) => {
```

```
        return this.props.firebase.listenTime(this.props.roomId, dispatcher);
    }

    setUpFirebaseData = (dispatcher) => {
        dispatcher.firebaseData = {}
    }

    firebaseSource = (value) => {
        return this.props.firebase.setSrc(this.props.roomId, value);
    }

    firebaseListenSource = (dispatcher) => {
        return this.props.firebase.listenSrc(this.props.roomId, dispatcher);
    }

    firebaseGetLibraryItem = (id) => {
        return this.props.firebase.getLibraryItem(id).then(value => value)
    }

    firebaseListenPlaylist = (controller) => {
        return this.props.firebase.listenPlaylist(this.props.roomId,
    controller)
    }

    firebaseAddItemsToPlaylist = (items) => {
        return this.props.firebase.addItemsToPlaylist(this.props.roomId, items
    )
    }

    render() {
        return (
            <div className='f-player'>
                <Player events={this.events} externalSetUp={this.setUp}
    libraryLogic={this.libraryLogic} playlistLogic={this.playlistLogic} />
            </div>
        )
    }
}
```