

Министерство науки и высшего образования Российской Федерации
Санкт-Петербургский политехнический университет Петра Великого
Название института

Работа допущена к защите
Должность руководителя М
_____ И.О. Фамилия
«_____» _____ 202X г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
ДИПЛОМНАЯ РАБОТА БАКАЛАВРА
РАЗРАБОТКА СИСТЕМЫ АДАПТИВНОГО СТРИМИНГА ВИДЕО В
НЕСКОЛЬКИХ СИНХРОНИЗИРОВАННЫХ ПОТОКАХ С
РАСПРЕДЕЛЁННОЙ СЕГМЕНТИРОВАННОЙ ЗАГРУЗКОЙ КОНТЕНТА
по направлению подготовки 09.03.03 Прикладная информатика
Направленность (профиль) 09.03.03_YY Наименование направленности (профиля)
образовательной программы

Выполнил
студент гр. 5130903/10301

В.Д. Ярцев

Руководитель
старший преподаватель,
степень, звание¹

Е.В. Комарова

Консультант²
должность, степень

И.О. Фамилия

Консультант
по нормоконтролю³

И.О. Фамилия

Санкт-Петербург
202X

¹ Должность указывают сокращенно, учёную степень и звание — при наличии, а подразделения — аббревиатурами. «СПбПУ» и аббревиатуры институтов не добавляют.

² Оформляется по решению руководителя ОП или подразделения. Только 1 категория: «Консультант». В исключительных случаях можно указать «Научный консультант» (должен иметь степень). Без печати и заверения подписи.

³ Обязателен, из числа ППС по решению руководителя ОП или подразделения. Должность и степень не указываются. Сведения помещаются в последнюю строчку по порядку. Рецензенты не указываются.

СОДЕРЖАНИЕ

Введение	4
Глава 1. Анализ предметной области, технологий стриминга и передачи видеоконтента для реализации серверной части системы.....	6
1.1. Анализ предметной области	6
1.1.1. Понятие видеостриминга.....	6
1.1.2. Роль видеостриминга в различных сферах.....	7
1.1.3. Основные концепции видеостриминга	8
1.2. Анализ функциональности видеоплееров существующих стриминговых платформ.....	9
1.3. Анализ существующих решений в области хранения видеоконтента...	11
1.4. Технологии стриминга видео	13
1.4.1. Протокол RTP	15
1.4.2. Стандарт MPEG-DASH.....	16
1.4.3. Протокол HLS	17
1.4.4. Сравнительный анализ технологий стриминга видео	19
1.5. Протоколы для передачи файлов между узлами сетей.....	21
1.5.1. Протокол HTTP	21
1.5.2. Протокол FTP.....	24
1.5.3. Сравнительный анализ протоколов передачи файлов между узлами сетей	25
1.6. Выводы	26
Глава 2. Проектирование информационной системы	27
2.1. Определение системных требований.....	27
2.2. Декомпозиция системы на компоненты	28
2.3. Техническая проработка системы	31
2.4. Инструменты реализации компонентов системы.....	35
2.4.1. Выбор СУБД	35
2.4.2. Выбор брокера сообщений.....	36
2.4.3. Выбор инструментов для реализации API-Шлюза.....	36
2.4.4. Выбор инструментов для реализации сервиса-загрузчика	37
2.4.5. Выбор инструментов для реализации сервиса-транскодировщика	37
2.4.6. Выбор инструментов для реализации сервиса-хранилища статического видеоконтента	38
2.4.7. Выбор инструментов для реализации клиентских приложений	38
2.4.8. Выбор инструмента балансировки нагрузки	38

2.4.9. Выбор инструмента кэширования	39
2.4.10. Обеспечение безопасности межсервисной сетевой коммуникации..	40
2.5. Выводы	41
Глава 3. Разработка системы	43
3.1. Разработка API-шлюза.....	43
3.1.1. Разработка модуля авторизации	43
3.1.2. Разработка головного модуля	48
3.2. Разработка сервиса-хранилища статического видеоконтента	53
3.3. Разработка сервиса-транскодировщика	55
3.4. Разработка сервиса-загрузчика	61
3.4.1. Разработка потребителя задач обработки сегментов.....	61
3.4.2. Разработка координатора потоков.....	62
3.4.3. Разработка планировщика очистки сессий	63
3.4.4. Разработка менеджера загрузки	64
3.5. Разработка видеоплеера	65
3.5.1. Хранилище	66
3.5.2. Менеджер событий.....	66
3.5.3. Компоненты интерфейса	67
3.6. Разработка редактора синхронизации.....	69
3.6.1. Хранилище	70
3.6.2. Менеджер событий.....	71
Список использованных источников.....	74
Глава 4. Программный код API-Шлюза.....	75

ВВЕДЕНИЕ

В последние годы видеоконтент стал одним из основных источников потребления информации людьми. Технологии видеостриминга используются в различных сферах, включая бизнес, образование, медицину и индустрию развлечений, и играют в них большую роль. Видеоконтент зачастую используется менеджерами по продукции различных компаний для повышения внутренних ключевых показателей эффективности (KPI), включая важные продуктовые метрики: вовлечённость пользователей, удержание пользователей, брендовое восприятие. Всё это говорит о том, что видео является важным инструментом повышения конверсии восприятия людьми контента.

Решаемые при обработке и воспроизведении видео, являются достаточно нетривиальными из-за возникающей высокой сетевой и вычислительной нагрузки. В связи с чем для многих организаций в различных отраслях становится нерентабельным создание и поддержка собственных систем хостинга и стриминга видео и возникает потребность в использовании уже имеющихся решений.

Однако большинство современных сервисов видеостриминга не поддерживает воспроизведение нескольких синхронизированных потоков видео в одном экземпляре видеоплеера, хотя такая потребность возрастает с каждым годом, так как производимый пользователями контент становится технологически сложнее.

Актуальность

На текущий момент проблема синхронизации нескольких видеопотоков решается со стороны бизнеса-потребителя сервисов видеохостинга с помощью средств монтажа. Функциональная возможность переключения между несколькими синхронизированными видеопотоками позволит пользователю самостоятельно выбирать предпочтительные ему фрагменты контента, что положительно скажется на продуктовых и непродуктовых метриках потребления контента в различных сферах: медицина - улучшенный анализ сложных процедур и операций, образование - повышение эффективности обучения благодаря возможности сопоставления видеоматериала и презентации, субтитров, бизнес и сфера развлечений - повышение времени, проведённого пользователем (time spent).

Цель исследования:

Создание приложения с системой адаптивного стриминга видео в нескольких синхронизированных потоках с возможностью навигации между ними, распре-

делённой сегментированной загрузкой контента и редактором синхронизации видеопотоков.

Задачи

1. Разработка архитектуры системы распределённой сегментированной загрузки и раздачи видеоконтента;
2. Разработка головного сервиса для обработки пользовательских запросов, инициации сессий загрузки, распределения асинхронных задач между другими сервисами, формирования пользовательской выдачи и обновления в реальном времени статуса загрузки для клиента;
3. Реализация сервиса для распределённой обработки загрузки видеоконтента по сегментам и преобразования полученного видеоконтента в нужный кодек и в форматы различных протоколов адаптивного стриминга (DASH, HLS) для поддержания кроссплатформенности клиентов;
4. Разработка сервиса для хранения и раздачи видеоконтента;
5. Разработка вспомогательных сервисов для очистки прерванных сессий загрузки и удаления неиспользуемого контента (сchedulers);
6. Разработка архитектуры клиентских приложений для адаптивного стриминга видео и редактора синхронизации видеопотоков;
7. Разработка видеоплеера с поддержкой различных потоковых форматов в разных браузерах и синхронизации между видеопотоками;
8. Разработка браузерного редактора синхронизации видеопотоков;
9. Разработка и настройка среды контейнеризации для автоматизированного развёртывания серверных и клиентских сервисов, СУБД, брокеров сообщений и балансировщиков нагрузки;
10. Проведение нагрузочного и end-to-end тестирования и апробации реализованного приложения;

ГЛАВА 1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ, ТЕХНОЛОГИЙ СТРИМИНГА И ПЕРЕДАЧИ ВИДЕОКОНТЕНТА ДЛЯ РЕАЛИЗАЦИИ СЕРВЕРНОЙ ЧАСТИ СИСТЕМЫ

В данной главе проводится анализ предметной области работы, функциональности видеоплееров существующих стриминговых платформ, а также имеющихся решений в области хранения видеоконтента, и на основании полученных данных рассматриваются основные технологии для стриминга видео и области их применения, а также протоколы для передачи файлов на разном уровне взаимодействия: между клиентом и сервером и межсерверном. Целью данной главы является определение оптимальных технологий для реализации системы с долгосрочным доступ к видеоконтенту и поддержанием возможности его адаптивного воспроизведения на клиентской стороне.

1.1. Анализ предметной области

1.1.1. Понятие видеостриминга

Цифровой мультимедийный контент охватывает широкий спектр цифровых данных, объединяющих текст, звук, изображения и видео, предоставляя пользователю разнообразные способы восприятия информации. С развитием информационных технологий цифровой контент стал важной частью жизни человека, включая интернет-платформы для хранения и распространения медиафайлов.

Одним из ярких примеров является видеоконтент, который благодаря широкому распространению Интернета, мобильных устройств и быстрому развитию технологий передачи данных стал доступен большому числу пользователей по всему миру. С ростом спроса на видеоконтент возникла потребность в эффективных и удобных способах его доставки. Технологии видеостриминга, обеспечивающие бесперебойную передачу видео в реальном времени, стали одним из основных инструментов для решения этой задачи [**apostolopoulosVideoStreaming**]. Эти технологии позволяют устранить необходимость в долгой загрузке файлов перед их воспроизведением, а также значительно облегчают доступ к контенту, позволяя обрабатывать контент на устройствах пользователей оптимально [**korotkovaMultimediaCommunication**].

Видеостриминг делает возможным мгновенный доступ к видеоматериалам, что является важным фактором в эпоху, когда пользователи ожидают максимального удобства и скорости при получении информации.

Видеостриминг — это технология передачи видеоконтента в режиме реального времени или по запросу, позволяющая воспроизводить видеофайлы по мере их загрузки. В отличие от традиционного скачивания, при котором необходимо дождаться полной загрузки файла перед его воспроизведением, видеостриминг обеспечивает более гибкий доступ к контенту. Это достигается путем разбиения видео на сегменты, которые последовательно передаются пользователю для декодирования и воспроизведения. Видеостриминг устраняет необходимость в загрузке файлов перед их просмотром, экономя время пользователей и пространство на их устройствах. Кроме того, видеостриминг решает проблему распределения контента. Он позволяет централизованно хранить видео на серверах и предоставить доступ к нему любому пользователю с подключением к интернету, устраняя географические ограничения [apostolopoulosVideoStreaming].

1.1.2. Роль видеостриминга в различных сферах

Видеостриминг стал неотъемлемой частью различных сфер жизни, включая образование, бизнес, развлечения и социальные взаимодействия. Технологии потокового вещания предоставляют новые возможности для передачи информации в реальном времени и по запросу, что существенно влияет на коммуникацию и восприятие контента.

Роль видеостриминга в образовании

В образовательной сфере видеостриминг позволяет проводить лекции, семинары и тренинги в режиме реального времени, обеспечивая доступ к знаниям независимо от географического положения участников [brameEducationalVideos]. Это особенно актуально для дистанционного обучения и курсов повышения квалификации. Кроме того, видеостриминг дает возможность повысить вовлеченность студентов. Платформы, использующие видеостриминг, могут интегрировать различные элементы взаимодействия и навигации, которые помогают студентам повышать свою вовлеченность в учебный процесс. Более того, сам факт наличия функциональности навигации между видеоконтентом значительно увеличивает среднее время просмотра контента студентами [brameEducationalVideos], что говорит о необходимости их развития.

Роль видеостриминга в коммерческой сфере

В коммерческой сфере видеоконтент, включая видеостриминг, также способствует значительному увеличению вовлеченности аудитории. В отличие от статичных рекламных материалов, видеоконтент является динамичным и визуально привлекательным, что позволяет удерживать внимание зрителей на протяжении более длительного времени [ilyinaVideoMarketing]. Видеоконтент позволяет создавать живые презентации, обзоры, инструкции, которые значительно эффективнее передают информацию, чем текстовые или статичные изображения. Это способствует лучшему пониманию продукта и повышает вероятность совершения покупки. Кроме того, 60% потребителей сначала обращают внимание на видеоконтент на веб-странице и только потом на текст или изображения. Также средний потребитель на 1200% больше делится видеоконтентом, чем ссылками и текстами вместе взятыми [ilyinaVideoMarketing]. На рисунке видно, что видеоконтент удерживает больше всего внимания пользователей по сравнению с другими представлениями информации. Видеостриминг в данном случае играет роль инструмента обеспечения качественного просмотра контента.

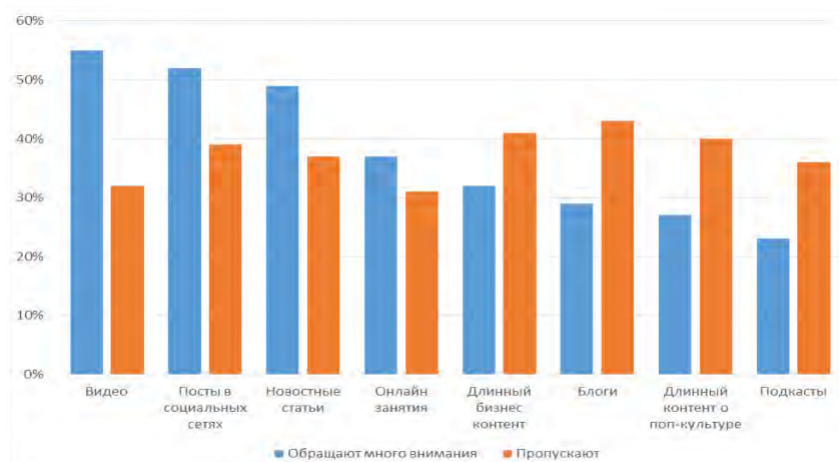


Рис.1.1. Распределение внимания пользователей между различными видами контента [ilyinaVideoMarketing]

1.1.3. Основные концепции видеостриминга

Ключевая особенность видеостриминга — это возможность доставки и воспроизведения контента практически в реальном времени. Традиционно стриминг используется для передачи как предварительно закодированных видеофайлов, так и живых трансляций. Однако существует ряд концепций видеостриминга,

которые необходимо учитывать при разработке для обеспечения качественного воспроизведения контента [**apostolopoulosVideoStreaming**]:

- Сжатие видео (Video Compression): видео требует больших объемов данных для хранения и передачи, что создает сложности при передаче через сети с ограниченной пропускной способностью, особенно для пользователей с медленным интернет-соединением. Сжатие видео позволяет уменьшить размер данных, сохраняя при этом приемлемое качество изображения;
- Адаптивный битрейт (Adaptive Bitrate Streaming): сетевые условия могут меняться в реальном времени, что приводит к снижению качества видео, если пропускная способность сети не соответствует требованиям для передачи высокого качества видеопотока. Эта концепция позволяет автоматически изменять качество видеопотока в зависимости от текущих сетевых условий;
- Протоколы потоковой передачи (Streaming Protocols): чтобы эффективно передавать видеопоток, необходимо синхронизировать передачу данных между сервером и клиентом, а также обеспечить управление потоком и контроль за его состоянием. Протоколы, соответствующие этому требованию, позволяют доставлять видео и аудио с минимальными задержкам при поддержке высокого качества передачи;

Соблюдение данных концепций является необходимым минимумом для реализации системы стриминга видео с высоким качеством.

1.2. Анализ функциональности видеоплееров существующих стриминговых платформ

Стриминговые платформы активно развиваются, в том числе их технический и интерфейсный функционал. Быстрое развитие возможностей видеоплееров стремительно повышает качество взаимодействия с пользователями, что приводит к увеличению вовлеченности зрителей. Эта вовлеченность является ключевым элементом для монетизации контента, через рекламу или другие механизмы, напрямую влияя на прибыль стриминговых сервисов [**xiaoShortVideoMarketing**].

Для анализа существующих стриминговых платформ стоит рассматривать те, которые имеют наибольший срок эксплуатации и стабильный спрос. Это обусловлено тем, что они не только предлагают проверенные временем технологии, но и формируют тренды в области функционала, которые улучшают пользовательский

опыт. Среди таких платформ можно выделить YouTube и Twitch, которые являются лидерами в своей категории и на протяжении многих лет задают стандарты в области потокового видео. Из базовых функций, которые содержат плееры всех перечисленных платформ можно выделить:

1. Управление воспроизведением:

- Кнопки воспроизведения и паузы: стандартные кнопки для начала и приостановки воспроизведения видео;
- Перемотка по времени: возможность перематывать видео по временной шкале плеера. Прокрутка осуществляется с точностью до нескольких секунд, что позволяет быстро перемещаться по видео;
- Управление громкостью: встроенная регулировка громкости с помощью ползунка, а также кнопка для выключения звука.

2. Полноэкранный режим: возможность развернуть видео на весь экран, что улучшает восприятие контента;
3. Регулировка скорости воспроизведения: плеер дает возможность замедлять или ускорять видео, регулируя скорость воспроизведения (например, 0.5x, 1x, 1.5x, 2x);
4. Поддержка доступности и отзывчивого интерфейса: плеер адаптирован под различные устройства, включая мобильные телефоны, планшеты и компьютеры.

Из платформенных особенностей можно выделить наличие у плеера YouTube субтитров на разных языках, созданных предварительно с помощью технологий машинного обучения, а также график интерактивной популярности, характеризующий, какие части видео пользователи чаще всего просматривают, перематывают или возвращаются к ним. Платформа Twitch, помимо перечисленного выше функционала, содержит интерактивные виджеты для проведения опросов разного типа, однако такая функциональность актуальна только для видеостриминга в режиме реального времени, так как выполняет роль коммуникации между автором и пользователями в процессе производства и потребления контента. Интерфейс видеоплееров рассмотренных платформ представлен на рисунках 2-3. Все перечисленные платформы предоставляют пользователям схожие базовые возможности для работы с видеоконтентом. Однако создание контента становится всё более сложным процессом, поскольку авторы действуют в условиях высокой конкуренции и вынуждены разрабатывать новые подходы к его производству. Сегодня попу-

лярным становится использование нескольких камер или видеопотоков, особенно в таких сферах, как образование, медицина или индустрия развлечений и медиа, где требуется передавать пользователю одновременно несколько видеопотоков, например, для отображения полной картины о проведении хирургической операции. Тем не менее, текущие платформы предоставляют возможность воспроизведения контента преимущественно в одном потоке. Управление отображением нескольких элементов видео осуществляется авторами посредством монтажа. Это ограничение накладывает дополнительные требования на процесс создания видео и не всегда позволяет в полной мере отразить потенциал современных методов производства контента. Для удовлетворения потребностей таких авторов и улучшения пользовательского опыта внедрение поддержки воспроизведения видео в нескольких синхронизированных потоках на уровне платформы позволило бы повысить вовлечённость пользователей за счёт большего выбора и гибкости в просмотре контента. В связи с этим в рамках данной работы будет разработано приложение, реализующее эту функциональность. Однако для реализации таких возможностей на клиентской стороне необходимо также, чтобы на стороне сервера поддерживались адаптивность стриминга, технологии сжатия и хранения видео, а также возможность сегментированной загрузки контента для авторов. Эти требования обеспечивают качественное воспроизведение видео для пользователей, даже при низкой пропускной способности сети, и для авторов контента, так как позволяют целостно загружать видео без потери данных и с возможностью восстановления сессий загрузки.

1.3. Анализ существующих решений в области хранения видеоконтента

Хранение больших объемов бинарных данных (blob) при разработке системы с распределённой загрузкой контента является технически нетривиальной задачей, требующей эффективного управления большими объемами данных. Важно учитывать вопросы масштабируемости, скорости доставки и безопасности. Для решения этих проблем необходимо провести анализ современных решений в этой области и заимствовать лучшие практики, чтобы создать оптимальную архитектуру системы.

Amazon Web Services

Amazon web Services (AWS) - один из старейших игроков на рынке облачных технологий, основанный в 2006 году. Он предоставляет широкий спектр компью-

терных услуг, таких как облачные хранилища, службы баз данных, аналитика, сетевые сервисы и многое другое [**muftiCloudReview**].

AWS состоит из компонентов перечисленных ниже:

- Amazon CloudFront: этот сервис предназначен для доставки контента, используемого на веб-сайтах. Он поддерживает как статические, так и динамические данные, а также потоковое видео. CloudFront использует распределенную сеть серверов по всему миру для оптимизации доставки контента пользователям;
- Балансировка нагрузки (Load Balancing): балансировка нагрузки позволяет повысить эффективность работы серверов и приложений. Это важный элемент сетевой инфраструктуры, который улучшает архитектуру классических веб-приложений. В AWS трафик распределяется между экземплярами сервисов, размещенными на разных ресурсах. При этом нагрузка также корректируется в зависимости от добавления или удаления экземпляров сервиса из балансировщика;
- Amazon RDS (Amazon Relational Database Service): сервис Amazon RDS предоставляет возможность работы с реляционными базами данных, аналогичными MySQL или Microsoft SQL, обеспечивая доступ к базе данных в облаке с удобным управлением и настройкой;
- Эластичный кэш (Elastic Cache): система управления кэшированием в облаке, которая обеспечивает эффективную работу за счет сокращения нагрузки на серверы. Она помогает оптимизировать использование памяти и снижать нагрузку на сервисы, улучшая их надежность;
- Простой сервис очередей (Simple Queue Service): облачный сервис для асинхронного обмена сообщениями между компонентами распределенных систем. Обеспечивает надежную доставку, масштабируемость, поддержку стандартных и FIFO очередей, а также интеграцию с другими сервисами AWS. Позволяет снизить связность компонентов, оптимизировать производительность и управлять нагрузкой на системы.

Яндекс.Облако

Яндекс.Облако — это набор взаимосвязанных сервисов и инструментов для работы в облачной платформе [**migratovYandexCloud**]. Это платформа является отечественной и предоставляет для клиентов сервисы и инструменты, решающие те же проблемы, что и составляющие части AWS. Яндекс. Облако состоит из компонентов перечисленных ниже [**yandexDocs**]:

- Объектное хранилище (Object Storage): этот сервис представляет собой масштабируемое хранилище для объектов данных, которое подходит для хранения любых типов файлов;
- Управляемые базы данных (Managed Databases): этот компонент включает поддержку PostgreSQL, MySQL, MongoDB, Redis, ClickHouse, Elasticsearch и других баз данных. Это решение похоже на AWS RDS;
- Load Balancers (Network/Application): Балансировщики нагрузки для распределения трафика между серверами, аналогичные AWS Load Balancing;
- Очереди сообщений (Message Queue): сервис позволяет организовать асинхронный обмен сообщениями между компонентами системы, аналогичен Amazon SQS.

И Yandex.Cloud, и AWS предоставляют мощные и гибкие решения для хранения, обработки и передачи больших бинарных объектов, которые обеспечивают высокую надежность, масштабируемость и интеграцию с другими сервисами. Однако их использование несёт в себе ряд определённых рисков и ограничений при разработке рассматриваемой в ходе работы системы, так как обе платформы работают на основе подписки или оплаты за использование, что делает их применение оправданным только для проектов с достаточным финансовым бюджетом, оба решения являются проприетарными и не предоставляют доступа к исходному коду, что ограничивает гибкость настройки и прозрачность. В связи с этим для решения поставленных в ходе работы задач оптимальным выбором будет реализовать платформу для хранения контента самостоятельно на основе следующих компонентов: балансировщики нагрузки, очереди сообщений, объектное хранилище, управляемые базы данных, управляемые базы данных для кэширования.

1.4. Технологии стриминга видео

Стриминг позволяет передавать видеоконтент в виде пакетов данных, что обеспечивает воспроизведение мультимедиа ещё до завершения загрузки файла. Такой подход снижает задержки перед началом просмотра и уменьшает требования к объему хранилища на стороне пользователя [[apostolopoulosVideoStreaming](#)].

Стриминг решает проблему распределения сетевой и вычислительной нагрузки не только на стороне клиента, но и на стороне сервера. Кроме того, использование технологий стриминга хорошо согласуется с микросервисной архитектурой приложений и позволяет без затруднений горизонтально масштабировать

раздачу видеоконтента - увеличивать количество реплик сервисов, отвечающих за хранение и передачу контента пользователям при соответствующих запросах.

Технологии стриминга можно разделить на следующие категории [apostolopoulosVideoStreaming]:

- Способ кодирования:
 - В реальном времени (Real-time encoding);
 - Предварительное (Pre-encoded video).
- Тип связи:
 - Точка-точка (Point-to-point);
 - Многовещание (Multicast);
 - Широковещание (Broadcast).
- Скорость передачи данных:
 - Постоянная (Constant Bit Rate, CBR);
 - Переменная (Variable Bit Rate, VBR).
- Описание потока:
 - Одиночное описание (Single Description Coding, SD);
 - Множественное описание (Multiple Description Coding, MD).
- Протокол передачи данных:
 - TCP;
 - UDP.
- Тип стриминга:
 - Прямой эфир (Live streaming);
 - Видео по запросу (On-demand streaming).
- Адаптивность:
 - Адаптивный стриминг (Adaptive streaming);
 - Неадаптивный стриминг (Non-adaptive streaming).

Технологии стриминга видео, которые будут использованы в системе, определяются на основе решаемых задач, так как каждая из них соответствует определённым критериям. Иными словами, выбор технологий должен опираться на их соответствии этим критериям для обеспечения наиболее эффективного решения. Основными критериями для реализации рассматриваемой в ходе работы системы являются:

- Способ кодирования: предварительное - система должна поддерживать долгосрочный доступ к видеоконтенту, а не доступ к контенту в реальном времени;

- Тип связи: точка-точка - в системе связь осуществляется между двумя конечными точками — сервером и клиентом;
- Скорость передачи данных: переменная - каждый видеопоток должен быть адаптивен и менять битрейт в зависимости от динамичности контента;
- Описание потока: одиночное - видеопоток может быть представлен в виде одного файла со списком сегментов, каждый из которых может быть получен клиентом по запросу;
- Протокол передачи данных: ТСП - система предоставляет долгосрочный доступ к контенту, все сегменты видео хранятся на сервере и являются целостными, поэтому должны доставляться пользователю без потерь;
- Тип стриминга: видео по запросу;
- Адаптивность: адаптивный стриминг - система должна иметь возможность регулировать качество воспроизведения видео на стороне пользователя в зависимости от индивидуальной скорости передачи данных.

1.4.1. Протокол RTP

RTP предоставляет функции транспортировки данных в реальном времени от источника до получателя, подходящие для приложений, передающих данные в реальном времени, такие как аудио, видео или данные моделирования, через мультикаст или юникаст-сети [rfcRtp].

Мультикаст-сети (Multicast) используются для отправки одного потока данных нескольким получателям одновременно.

Юникаст-сети (Unicast) предполагает индивидуальную передачу данных от отправителя к каждому получателю. Для каждого клиента создаётся уникальный поток.

Каждый RTP-пакет содержит фиксированный заголовок, за которым следует полезная нагрузка переменной длины. Заголовок включает информацию, необходимую для синхронизации и демультиплексирования медиапотоков [rfcRtp]. Формат заголовка представлен на рисунке [rfcRtp].

RTP использует переменную скорость передачи данных в зависимости от характеристик медиа и сети. Этот протокол также обычно использует одно описание потока для мультимедийных данных, где каждый поток определяется отдельно для синхронизации. В качестве протокола для передачи данных используется UDP, так как он предназначен для приложений с низкими задержками и реального

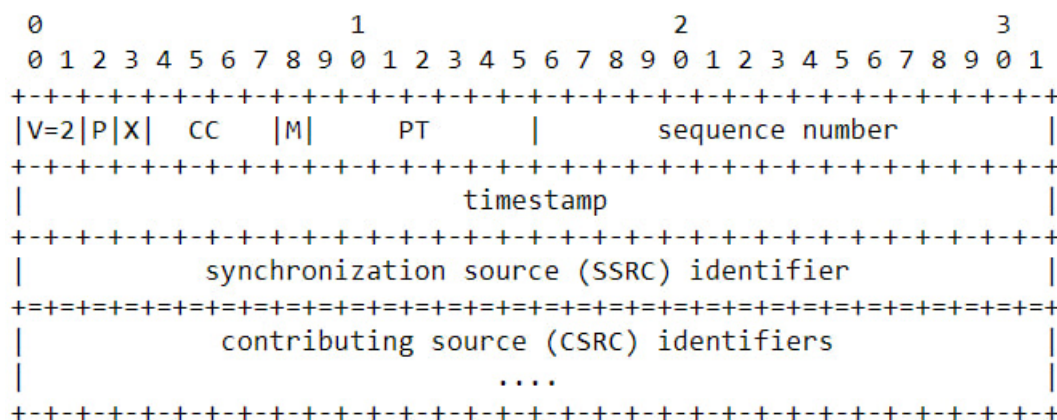


Рис.1.2. Заголовок RTP-пакета [rfcRtp]

времени - целостность данных не играет ключевой роли, допустимы небольшие потери пакетов. RTP поддерживает адаптивный стриминг с использованием RTCP, который предоставляет отчеты о состоянии сети. Это позволяет адаптировать видеопоток [rfcRtp].

1.4.2. Стандарт MPEG-DASH

MPEG-DASH (Dynamic Adaptive Streaming over HTTP) — это стандарт для эффективной потоковой передачи мультимедийных данных с использованием существующей инфраструктуры HTTP. Он поддерживает как потоковую передачу по запросу (on-demand), так и в реальном времени (live streaming) [mpegDashSite].

Этот стандарт разделяет видео и аудио контент на небольшие сегменты, каждый из которых может быть закодирован с разным битрейтом. Также он поддерживает возможность определения длины сегмента по времени, что крайне удобно для выбора оптимального значения при запуске, например, A/B экспериментов для приложений, использующих эту технологию. Без использования средств нагрузочного тестирования - только на основе различий в продуктовых и технических метриках между двумя версиями приложения можно определить оптимальное значение длины сегмента.

Воспроизведение начинается с загрузки первого сегмента, после чего плеер динамически выбирает сегменты с оптимальным качеством в зависимости от текущей скорости сети и возможностей устройства. MPEG-DASH не зависит от кодеков, что позволяет использовать различные форматы сжатия: H.264, H.265, VP8 и другие. Стандарт использует в качестве протокола передачи данных на прикладном уровне HTTP, на транспортном - TCP [sodagarMpegDash].

Важно отметить, что использование протоколов HTTP и TCP в основе стандарта позволяет поддерживать целостность передаваемых пользователю сегментов, что влечёт за собой определённые накладные расходы при передаче данных, но лучше всего подходит для стриминга видео по запросу, так как улучшает пользовательский опыт в системах с долгосрочным доступом к контенту.

Стандарт поддерживает как потоковое вещание по запросу, так и прямые трансляции (live streaming), включая функции перемотки назад и паузы в прямом эфире. Представление видеопотока описывается в файле формата MPD. Файл MPD (Media Presentation Description) содержит структуру мультимедийного контента, включая информацию о сегментах, их длительности, битрейте, разрешении и других параметрах. MPD может быть статическим или динамическим, в зависимости от типа контента [sodagarMpegDash]. Структура формата MPD представлена на рисунке 1.3.

1.4.3. Протокол HLS

HTTP Live Streaming (HLS) — это протокол адаптивной потоковой передачи мультимедийного контента, разработанный компанией Apple, который используется для доставки мультимедийного контента по сети Интернет [rfcHls].

HLS поддерживает потоковую передачу в реальном времени, где данные кодируются и сегментируются для прямых трансляций сразу при получении данных и передаёт их пользователю.

Этот протокол также поддерживает передачу заранее закодированных видео по запросу пользователя. Предварительно созданные и закодированные сегменты упрощают доставку пользователю и минимизируют задержки. Основным режимом работы HLS — это точка-точка, где клиент напрямую запрашивает сегменты через HTTP у сервера. Протокол изначально не поддерживает многовещание или широкое вещание [rfcHls]. Для работы с такими режимами требуется дополнительная инфраструктура (например, использование CDN).

HLS позволяет использовать переменный битрейт. Клиент адаптируется к доступной пропускной способности, выбирая поток с соответствующим качеством. Это позволяет оптимизировать использование сети. Протокол также теоретически поддерживает постоянный битрейт, однако особое внимание в протоколе уделяется именно переменному битрейту. HLS работает исключительно через HTTP, а значит, поверх TCP. Это обеспечивает целостность данных при передаче [rfcHls].

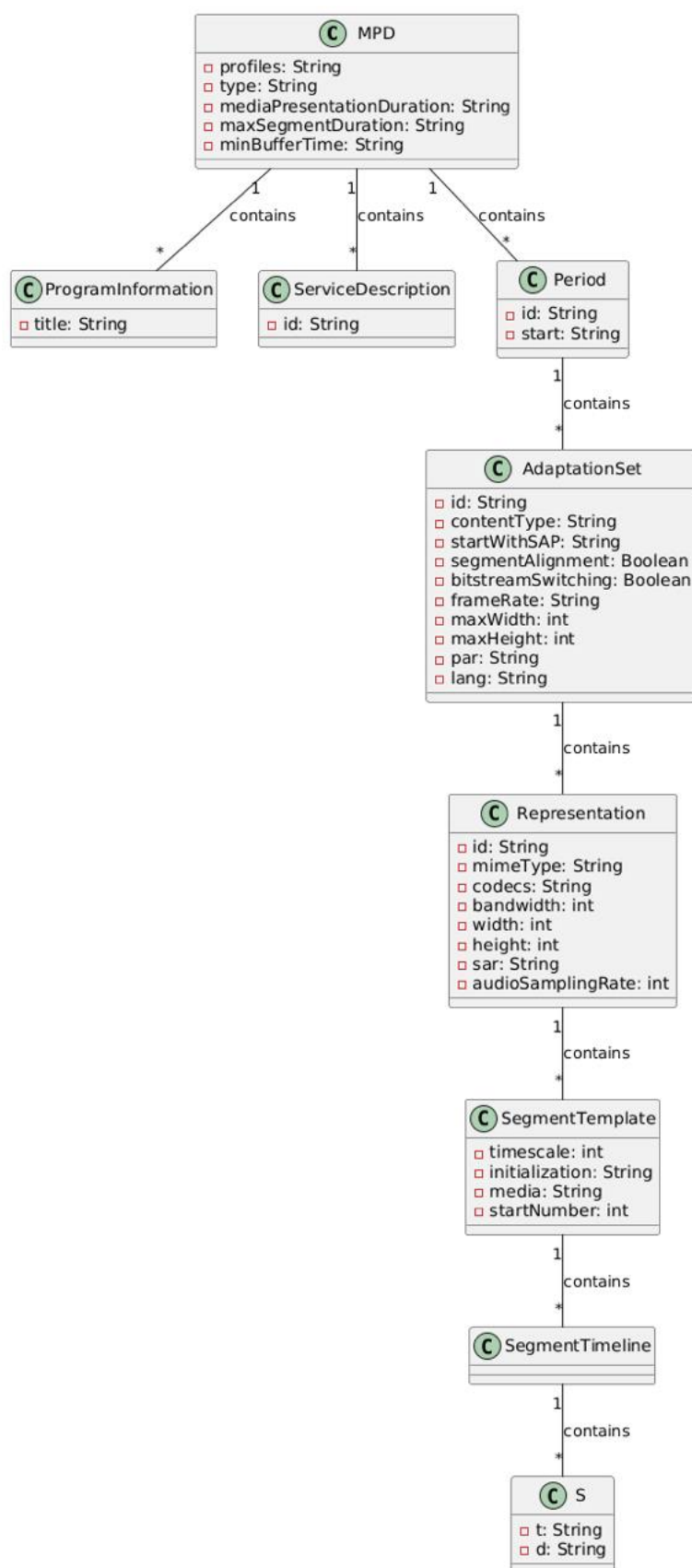


Рис.1.3. Структура формата MPD [rfcRtp]

HLS использует единую последовательность сегментов, которые представляют

поток на определённом уровне качества [fecheyHlsReview]. Его можно отнести к одиночному типу описания потока.

Так как протокол был создан компанией Apple, он занимает доминирующее положение в её продуктах, в частности, в мобильной и веб версиях браузера Safari. HLS поддерживается по умолчанию в этом браузере и не требует установки дополнительной инфраструктуры для использования, что выделяет её среди остальных технологий, когда речь идёт о продуктах Apple.

Концептуально HLS состоит из трёх частей: серверного компонента, компонента раздачи и клиентского программного обеспечения [fecheyHlsReview]. Архитектура протокола представлена на рисунке 1.4.

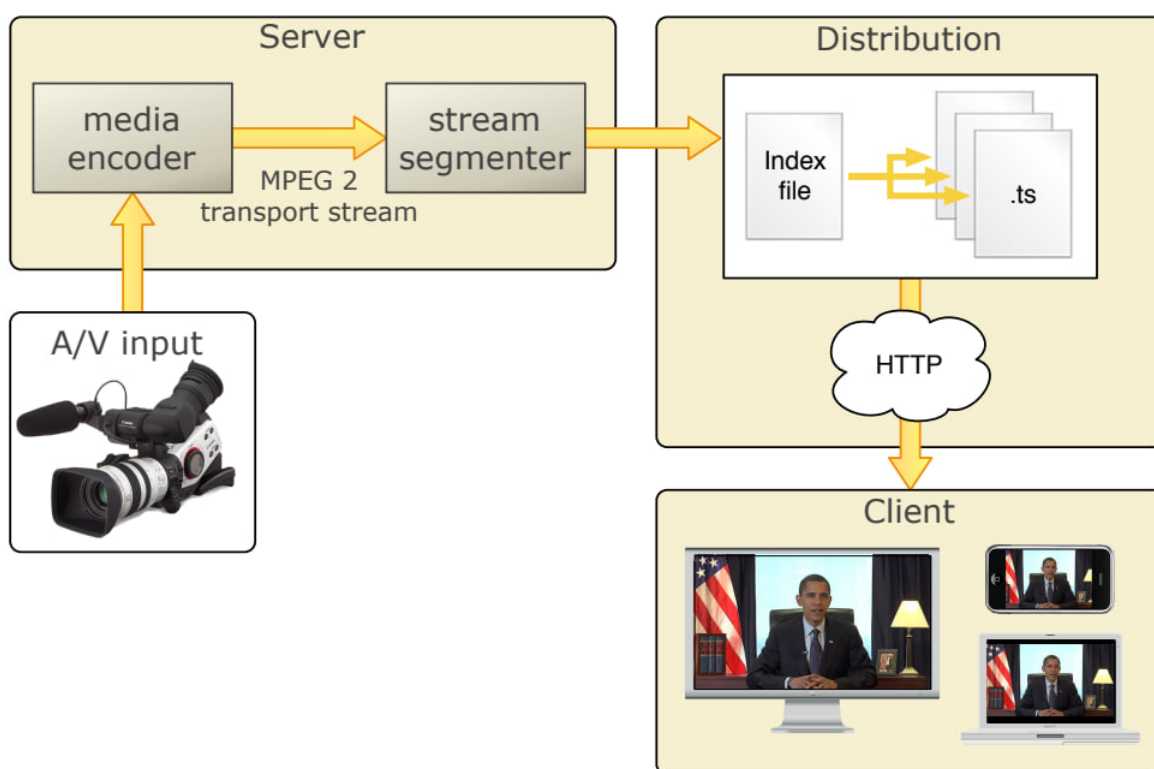


Рис.1.4. Архитектура протокола HTTP Live Streaming [fecheyHlsReview]

1.4.4. Сравнительный анализ технологий стриминга видео

Для выбора оптимального решения стриминга видео с долгосрочным доступом к контенту необходимо сравнить определённые ранее технологии по выделенным в разделе 1.2 критериям. Рассмотрим таблицу 1.1.

Таблица выше показывает, что для долгосрочного доступа к видеоконтенту наиболее подходящими решениями являются HLS и MPEG-DASH. Обе технологии поддерживают переменный битрейт, благодаря чему возможен адаптивный

Таблица 1.1

Сравнение технологий стриминга видео

Критерий сравнения	RTP	MPEG-DASH	HLS
Способ кодирования	Реальное время (Real-time)	Предварительное (Pre-encoded) и Реальное время (Real-time)	Предварительное (Pre-encoded)
Тип связи	Точка-точка (Point-to-point), Многовещание (Multicast)	Точка-точка (Point-to-point)	Точка-точка (Point-to-point)
Скорость передачи данных	Переменная (VBR)	Переменная (VBR), Постоянная (CBR)	Переменная (VBR), Постоянная (CBR)
Описание потока	Множественное (MD)	Одиночное (SD)	Одиночное (SD)
Протокол передачи данных	UDP	TCP	TCP
Тип стриминга	Прямой эфир	Видео по запросу, Прямой эфир	Видео по запросу, Прямой эфир
Адаптивность	Неадаптивный стриминг	Адаптивный стриминг	Адаптивный стриминг

стриминг. Они используют протокол TCP, что гарантирует целостность данных при передаче. Эти протоколы позволяют получить видео по запросу и обеспечивают качественный пользовательский опыт на разных устройствах.

В качестве основной технологии для стриминга видео явно стоит выделить MPEG-DASH. Помимо остальных преимуществ, эта технология позволяет более качественно минимизировать задержки при передаче данных, благодаря более атомарному разбиению видео на сегменты [bouzakariaLowLatencyDash]. Однако для разных версий браузера Safari стоит отдать предпочтение протоколу HLS. Инфраструктура этого браузера оптимизирована именно для работы с HLS. Таким образом, для поддержки кроссплатформенности рассматриваемой в ходе работы системы наилучшим решением будет использовать и MPEG-DASH, и HLS, но для разных браузерных окружений. Загрузку инфраструктуры MPEG-DASH необходимо производить только при отсутствующей поддержке HLS.

1.5. Протоколы для передачи файлов между узлами сетей

При выборе протокола в системах с загрузкой и передачей файлов между узлами сети важно учитывать особенности видеоконтента, который отличается большим объемом и требует продуманного подхода. Различные протоколы предлагают решения в зависимости от задач: одни оптимизированы для передачи небольших сегментов, другие обеспечивают целостность и надежность при отправке крупных файлов. Также важно учитывать безопасность системы и кроссплатформенность технологии на клиентской стороне.

1.5.1. Протокол HTTP

HTTP (HyperText Transfer Protocol) — это протокол передачи гипертекстовых документов, который является основным способом обмена данными в интернете [rfcHttp10]. HTTP был разработан для обмена текстовыми документами и ресурсами между веб-серверами и клиентами, однако со временем он стал основным механизмом для передачи данных любых типов, включая файлы или бинарные данные.

Основные принципы работы HTTP:

- HTTP представляет собой протокол прикладного уровня модели OSI (Open Systems Interconnection). Он работает в модели запрос-ответ, где клиент (обычно это веб-браузер) отправляет запрос к серверу, а сервер в ответ отправляет данные;
- Каждый HTTP запрос состоит из:
 - Метод запроса (GET, POST, PUT, DELETE и др.);
 - URL (Uniform Resource Locator) — адрес ресурса;
 - HTTP-версия (например, HTTP/1.1);
 - Заголовки — дополнительные метаданные, которые могут содержать информацию о клиенте [gourleyHttpGuide]. Заголовки можно использовать для передачи данных о текущей пользовательской сессии в зашифрованном виде, а также для разметки сегментов файлов при их загрузке в виде бинарных данных;
 - Тело запроса — данные, передаваемые с запросом.

Пример взаимодействия между сервером и клиентом на основе протокола HTTP представлена на рисунке 1.5.

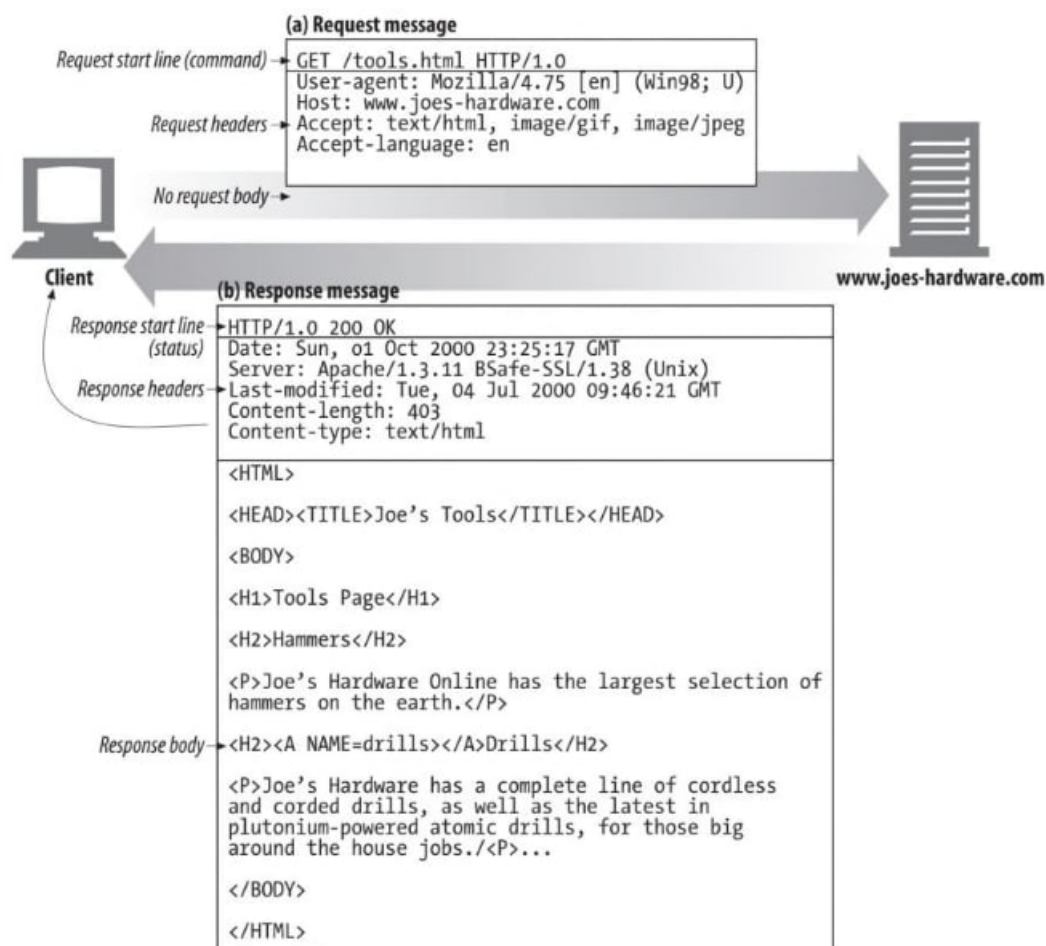


Рис.1.5. Пример взаимодействия между сервером и клиентом на основе протокола HTTP
[gourleyHttpGuide]

Рассмотрим различные способы передачи файл на сервер с помощью HTTP:

- Одним из самых распространенных способов отправки файлов через HTTP является использование заголовка "Content-Type" со значением "multipart/form-data" [rfcHttp10]. Он используется в основном при отправке данных через HTML-формы, когда помимо файлов могут быть отправлены другие типы данных, такие как текстовые поля;
- Файлы можно передавать в виде бинарных данных. Как и в предыдущем способе, для передачи используется заголовок "Content-Type" со значением "application/octet-stream". Этот подход обычно используется для передачи больших файлов или когда необходимо передать двоичные данные, такие как изображения или видео. Преимущество данного способа неочевидно, однако он позволяет разбивать файл на клиентской стороне на несколько сегментов и отправлять его в виде нескольких частей. Разметку для этих сегментов можно передавать в заголовках запроса;

- Файлы могут быть закодированы в строку Base64 и отправлены в виде строки. Например, в теле запроса или в качестве части данных JSON. Такой способ не подходит для передачи больших файлов, так как требует дополнительных неэффективных вычислений на клиентской и серверной сторонах.

Преимущества использования протокола при загрузке файлов:

- Кроссплатформенность и поддержка всех браузеров позволяет передавать файлы с различных устройств и браузерных сред;
- HTTP позволяет отправлять файлы различными способами и предоставляет большую гибкость в формировании запросов к серверу благодаря наличию заголовков. Файлы можно предварительно разбить на сегменты на клиентской стороне и последовательно или параллельно отправлять на сервер;
- HTTP в совокупности с протоколом SSL/TLS позволяют обеспечить защищённое шифрованное соединение между клиентом сервером. Совокупность этих протоколов обобщённо имеет название HTTPS. HTTPS поддерживается по умолчанию всеми современными браузерами и не требует дополнительных настроек в отличие от других протоколов.

Недостатки использования протокола при загрузке файлов:

- Спецификация HTTP не ограничивает размер запроса, однако серверы и браузеры могут накладывать ограничения на размер передаваемых данных [rfcHttp10]. В таком случае файл необходимо предварительно обработать на стороне клиента и отправлять данные несколькими запросами в виде, например, сегментов;
- При отправке больших файлов HTTP-запросы могут сильно загружать сервер, так как загрузка файлов без предварительного преобразования на клиентской стороне требует их синхронной обработки на стороне сервера;
- Если передача файла занимает много времени, могут возникать проблемы с тайм-аутами на стороне клиента или сервера. Тайм-ауты могут возникать, если сервер или клиент не получают данных в течение установленного времени.

1.5.2. Протокол FTP

FTP - это протокол, разработанный для надежной передачи файлов между клиентами и серверами в сетях с использованием TCP [**rfcFtp**].

Основные цели и задачи протокола FTP:

- Перенос файлов: обеспечение механизма для передачи файлов между хостами, независимо от их операционных систем или файловых структур [**rfcFtp**];
- Абстракция файловой структуры: протокол предоставляет общий интерфейс для работы с файлами [**rfcFtp**], скрывая различия в файловых системах между клиентом и сервером;
- Поддержка взаимодействия человека с системой: FTP предлагает удобный способ работы для клиента, предоставляя команды для взаимодействия с сервером через текстовые запросы;

Особенности протокола:

- FTP использует два независимых соединения [**rfcFtp**]:
 - Управляющее соединение (Control Connection): работает на 21 TCP-порту и используется для передачи команд и ответов между клиентом и сервером [**rfcFtp**];
 - Соединение данных (Data Connection): работает на 20 TCP-порту и используется непосредственно для передачи файлов каталогов.
- Режимы передачи данных:
 - Активный режим (Active Mode): сервер открывает соединение для передачи данных на указанный клиентом адрес [**rfcFtp**];
 - Пассивный режим (Passive Mode): сервер сообщает клиенту о доступном порте, и клиент инициирует соединение [**rfcFtp**].

Преимущества FTP:

- Протокол предоставляет команды для работы с файлами и каталогами: создание, удаление, переименование и перемещение;
- TCP, который использует FTP, гарантирует доставку всех пакетов [**rfcFtp**];
- FTP поддерживает возобновление передачи файлов [**rfcFtp**];
- Протокол обеспечивает универсальный подход для передачи как текстовых, так и двоичных данных.

Ограничения и недостатки FTP:

- Логины, пароли и данные передаются в незашифрованном виде [**rfcFtp**]. Необходимо самостоятельно поддерживать шифрование на серверной и клиентской частях приложения;
- Протокол не поддерживает современные механизмы авторизации, такие как токены;
- Не поддерживается в современных браузерах.

1.5.3. Сравнительный анализ протоколов передачи файлов между узлами сетей

Ключевым требованием к протоколу для передачи файлов между клиентом и сервером является его кроссплатформенность. Из рассмотренных в данной главе протоколов этому требованию соответствует только протокол HTTP. Кроме того, вместе с этим протоколом во всех современных браузерах по умолчанию поддерживается также SSL/TLS [**rfcHttp10**], что обеспечивает безопасность передаваемых пользователем данных. Поэтому в качестве протокола для передачи данных между сервером и клиентом оптимальным выбором является HTTP.

В системах хранения и обработки видео зачастую используются разные серверы для выполнения этих функций в отдельности. Для передачи файлов между сервером обработки и сервером хранения также необходимо сделать выбор в пользу того или иного протокола. В данном случае нет необходимости беспокоиться о его поддержке, так как клиенты протокола можно установить на соответствующие сервера. Кроме того, сеть передачи между ними можно виртуализировать и скрыть от внешнего пользователя, на сегодняшний день с этим отлично справляются средства контейнеризации Docker, containerd, и другие. Это позволяет обеспечить безопасность между серверами, поэтому средства авторизации и аутентификации самого протокола играют посредственную роль. Таким образом, для межсерверного обмена файлами подходит не только протокол HTTP, но и FTP. Благодаря своему двухканальному режиму, более развёрнутому списку команд для работы с файлами и директориями, а также возобновлению сессий загрузки FTP является наиболее эффективным решением для межсерверного обмена файлами.

1.6. Выводы

В данной главе были рассмотрены технологии стриминга и передачи видеоконтента для реализации серверной части системы. Проведён сравнительный анализ производительности, адаптивности и совместимости с современными браузерами соответствующих технологий.

По итогам сравнительного анализа были сделаны следующие выводы:

- Для стриминга видео целесообразно использовать протоколы MPEG-DASH и HLS, которые обеспечивают адаптивный стриминг, минимальные задержки и целостность данных при передаче. На серверной части системы необходимо поддерживать оба протокола, а на клиенте - в зависимости от браузерной среды: MPEG-DASH подходит для большинства браузеров, кроме Safari - для него более предпочтителен HLS;
- В качестве кодека для сжатия видео выбран H.264, так как он демонстрирует лучшее качество сжатия при меньших битрейтах, высокую производительность, поддержку аппаратного декодирования и совместимость с современными устройствами и протоколами;
- Для передачи данных между клиентом и сервером выбран HTTP, благодаря его кроссплатформенности, поддержке шифрования (с использованием SSL/TLS) и гибкости работы с сегментированной передачей данных. Для межсерверного обмена контентом был выбран FTP, который обеспечивает надёжную передачу файлов с возможностью возобновления загрузок.

ГЛАВА 2. ПРОЕКТИРОВАНИЕ ИНФОРМАЦИОННОЙ СИСТЕМЫ

В данной главе рассмотрено проектирование системы распределённой сегментированной загрузки и раздачи видеоконтента. В ней содержится определение системных требований приложения и техническая проработка системы на их основе.

2.1. Определение системных требований

Важно уточнить, что системные требования приложения являются отправной точкой для всего процесса разработки, поскольку именно они определяют, каким образом система будет решать различные задачи. От них зависит выбор архитектурных паттернов, поэтому не стоит пренебрегать их явным определением.

В текущей работе одним из ключевых требований к системе является её способность адаптироваться к увеличению нагрузки. Обработка видео достаточно ресурсоёмкий процесс, однако он не должен влиять на работу пользователей с приложением. Пользователи, как минимум, должны в любой момент времени беспрепятственно получать доступ к интерфейсу приложения, а это уже говорит о необходимости разделения бизнес-логики приложения на клиентскую и серверную.

Кроме того, так как загрузка видеоконтента является продолжительным процессом, система должна быть терпима к изменению пропускной способности сети пользователя.

Пользователи в системе фактически разделяются на две категории: загрузчики (владельцы) и потребители контента. Точкой входа для загрузчиков является страница для редактирования и загрузки видеоконтента синхронизированных потоков, а точкой входа для потребителей - страница с видеоплеером, который воспроизводит эти потоки синхронно. Это означает, что права различных категорий пользователей различаются, что формирует новое функциональное требование в виде безопасности приложения. Это требование справедливо в целом для любой информационной системы - пользователям для взаимодействия с ней могут быть доступны только определённые разрешённые узлы системы.

Система должна быть терпима к выходу из строя отдельных узлов. Приложения, которые работают с большим количеством данных, а в этом проекте речь идет об обработке видеоконтента, умеют к сбоям адаптироваться к сбоям.

Подытожив все перечисленное выше, можно определить следующие основные системные требования к приложению:

- Масштабируемость;
- Безопасность;
- Доступность;
- Отказоустойчивость.

2.2. Декомпозиция системы на компоненты

При декомпозиции приложения на структурные составляющие необходимо отталкиваться от точек входа в него. В этом проекте определены две точки входа: видеоплеер и редактор синхронизации видеопотоков. Причём, видеоплеер должен иметь возможность быть интегрированным без дополнительных действий со стороны бизнеса на другие сайты, а на странице редактора он должен отображаться как полученное в результате загрузки видео превью итогового результата. Наиболее подходящим выбором в данном случае будет сепарировать логику редактора от видеоплеера и сделать видеоплеер независимой страницей. Старые и современные браузеры имеют хорошую поддержку Iframe API, которое позволяет встраивать одни html страницы как независимые части других. Причём по умолчанию такие встраиваемые страницы имеют существенные ограничения по использованию браузерного API относительно родительского документа, а доступ выдаётся точно с помощью атрибутов тега iframe. Это также говорит о безопасности такого решения. Для того, чтобы разделить две данных страницы, они должны иметь общий источник данных. Это зона ответственности серверной части приложения (backend).

Редактор синхронизации для своей работы требует от сервера контракты взаимодействия для аутентификации, создания модели видео, потоков, сессий загрузки, определения состояния и прогресса загрузки видео, загрузки потоков видео и их консистентности. Видеоплееру также требуется контракт взаимодействия с сервером для отслеживания состояния видео, а также дополнительно ссылки на статику видео в виде манифестов MPEG и HLS и соответствующих чанках, с помощью которой он сможет воспроизводить контент. Схема приложения на текущем уровне декомпозиции представлена на рисунке 2.1.

Важно понимать, что высоконагруженные приложения не могут себе позволить использовать сервер одновременно и для выполнения какой-либо бизнес

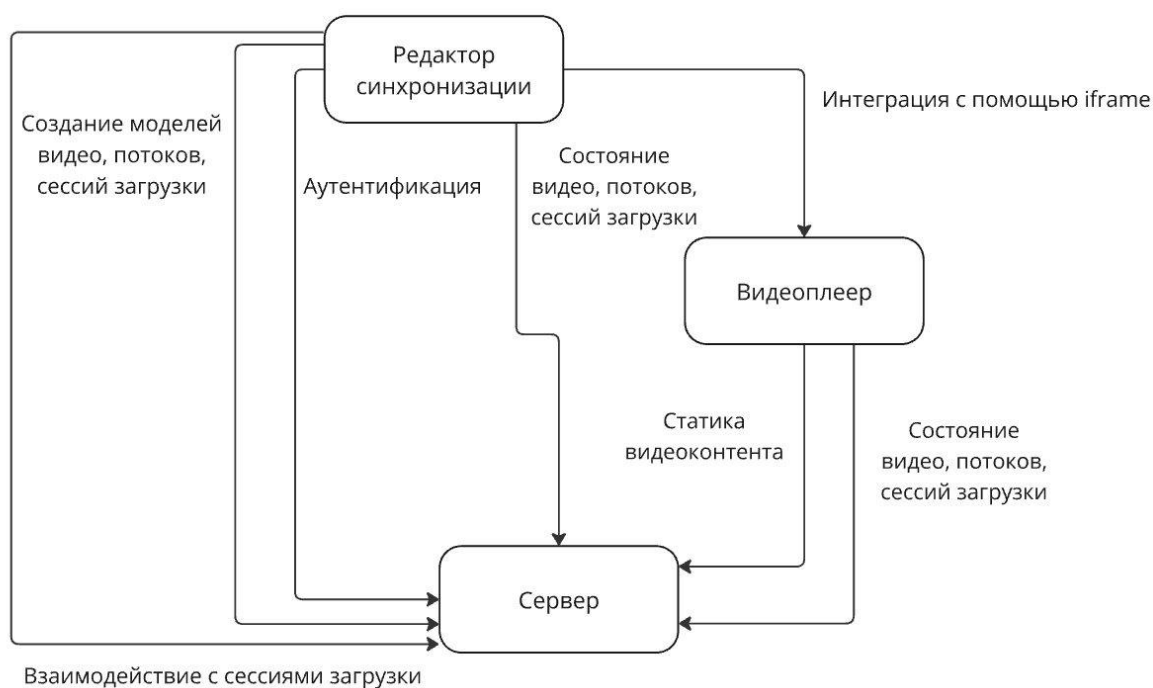


Рис.2.1. Схема системы первого уровня декомпозиции

логики, и для раздачи статического контента. Это обусловлено тем, что часть своего процессорного времени сервер будет тратить также на передачу статике пользователю. В случае с видеоконтентом раздача его статического контента будет достаточно ресурсоемкой операцией. Необходимо какое-то разделение ответственности. Кроме того, в рассматриваемом проекте количество чтений статического контента будет сильно превышать количество записей, как в аналогичных системах. Эти доводы говорят о необходимости сепарирования логики загрузки и раздачи статического контента видео от головной части сервера. Теперь серверную часть можно представить в виде головного сервиса - API-Шлюза (API-Gateway), и сервиса загрузки и раздачи видео статике. Обновленная схема системы представлена на рисунке 2.2.

Среди перечисленных ранее требований к системе содержится доступность и отказоустойчивость. Обеспечение бесперебойной загрузки видеоконтента при нестабильной пропускной способности пользователя является нетривиальной задачей. При загрузке видео единым потоком трудно гарантировать его своевременную доставку и обработку, более того, длительность HTTP запросов ограничивают современные браузеры, принудительно их прерывая. Помимо прочего, для таких ресурсоемких обработок тяжело рассчитать допустимую нагрузку. Поэтому при проектировании архитектуры приложения оперировать абстрактными сущностями нет возможности, необходимо учитывать специфику. Одним из решений проблемы отказоустойчивости и доступности системы, которая должна обрабатывать



Рис.2.2. Схема системы второго уровня декомпозиции

большое количество бинарных данных, является сегментация контента, то есть разбиение его на сегменты или чанки. Они могут быть произвольной длины, однако для того, чтобы равномерно распределить нагрузку между клиентами, необходимо определить одинаковый размер для всех сегментов.

Такое решение позволило рассматривать процесс загрузки видеоконтента не как "чёрный ящик основную работу которого выполняет браузер и разработанные части серверных фреймворков, а как достаточно предсказуемую и контролируруемую процедуру. Клиент сможет обрабатывать прерванные запросы и продолжать работу с сессиями загрузки, даже через несколько минут после восстановления сети или сервера.

Можно заметить, что между сервисом-хранилищем статического видеоконтента и API-шлюзом есть незаполненное функциональное пространство, так как в API-шлюзе мы получаем необработанные бинарные данные в видео сегментов, а на сервисе-хранилище статика видео хранятся данные в виде закодированных чанков и объединяющих их манифестах HLS и MPEG-DASH. Это пространство должен занять транскодировщик, в зоне ответственности которого будет принять бинарные данные от API-Шлюза, преобразовать их в нужные форматы, аудио и видео кодеки, и загрузить на сервис со статикой.

Однако располагать транскодировщик внутри головного сервиса является дорогостоящим и трудно масштабируемым решением, так как основная задача

API-шлюза состоит в быстрой обработке пользовательских запросов, а транскодирование это сложная вычислительная задача. Как минимум структурно, даже не затрагивая возможности горизонтального масштабирования, транскодировщик должен располагаться отдельно. Обновленная схема системы с учетом транскодировщика представлена на рисунке 2.3.

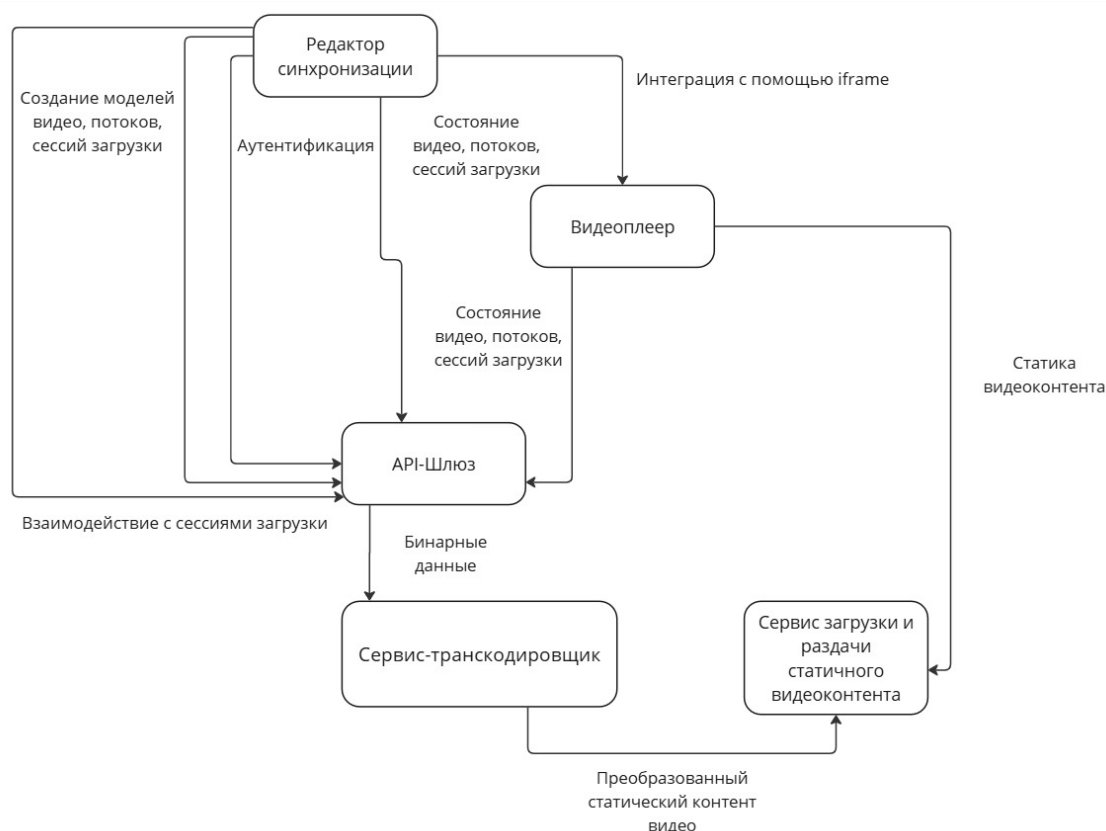


Рис.2.3. Схема системы третьего уровня декомпозиции

2.3. Техническая проработка системы

В этой части построения архитектуры не только сами компоненты системы, но и то, как именно будет реализовано взаимодействие между компонентами, какие технологии и инструменты будут использоваться, а также какие механизмы обеспечат соответствие системным требованиям программного продукта.

Из рассмотренной ранее схемы приложения (рис. 2.3) можно сделать вывод, что система точно содержит несколько состояний. Это говорит о необходимости использования базы данных для их хранения. На основе ранее описанных функциональных и системных требований можно выделить четыре сущности:

- Пользователь;

- Видео;
- Поток;
- Сессия загрузки.

Пользователь (User) необходим для того, чтобы по загруженному в систему контенту можно было идентифицировать его создателя. Кроме того, пользователи должны иметь разграничение доступа к контенту - только создатели видео могут взаимодействовать с сессиями загрузки. Экземпляр пользователя должен содержать идентификатор, email - для возможного продуктового масштабирования в будущем (отправка писем на почту, двухфакторная аутентификация), хеш пароля - для аутентификации, дату регистрации.

Видео (Video) представляет собой абстрактную сущность, которая содержит идентификатор, заголовок, описание, дату создания, дату полной загрузки и ссылку на пользователя-создателя, а также текущий статус видео. Статус включает следующие состояния:

- Создано (CREATED);
- Загружается (UPLOADING);
- Раздаётся (DISTRIBUTED);
- Заблокировано (BLOCKED).

Каждое видео может содержать множество связанных с ним потоков, которые будут отображаться в плеере.

Поток (Flow), как сущность, похож на видео, однако содержит специфичную информацию о конкретном потоке, который будет воспроизводиться в плеере. Он содержит идентификатор, текущий статус, дату создания, дату загрузки, ссылку на привязанное к потоку видео. Статус включает следующие состояния:

- Создано (CREATED);
- Загружается (UPLOADING);
- Раздаётся (DISTRIBUTED);
- Скрыт (HIDDEN).

Каждый поток может содержать сессию загрузки, но притом только одну. К потоку может быть привязана сессия загрузки тогда и только тогда, когда его статус равен "Загружается".

Благодаря сессии загрузки пользователи могут отслеживать процесс загрузки каждого потока в отдельности. Кроме того, сессия позволяет поддерживать консистентность загрузки и обработки видеоконтента, чтобы система была устойчива к "промахам" при загрузке видео со стороны пользователей. Она должна

содержать идентификатор, что полезно также для логирования и отслеживания промежуточных состояний загрузки для стороны разработки, общее количество байт потока, количество уже загруженных и обработанных байтов видео, ссылку на связанный поток.

Данные имеют чёткие структуры и связи между собой, что говорит о необходимости использования реляционной базы данных для их хранения.

Для хранения данных системы были созданы соответствующие таблицы с необходимыми столбцами и типами в рамках СУБД PostgreSQL. Схема таблиц представлена на рисунке 2.4.

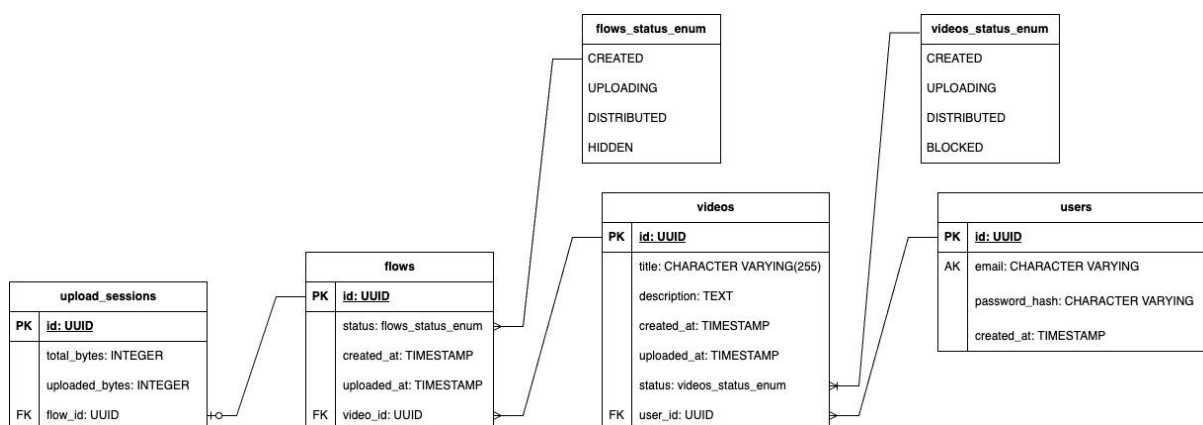


Рис.2.4. Схема таблиц базы данных системы

Код DDL (Data Definition Language) базы данных находится в приложении 1.

При загрузке видео, разделённого на сегменты, последовательно, без задержек существенно возрастет нагрузка на сервис-транскодировщик. Помимо того, что сама операция транскодирования является ресурсоёмкой, количество одновременных сессий загрузки может быть большим. API-шлюз должен отвечать на запросы пользователей как можно скорее и позволить ему дожидаться результата выполнений транскодирования, проводить валидацию отправленных сегментов нет возможности, так как это ухудшит пользовательский опыт.

Чтобы решить эту проблему, головному сервису необходимо возвращать ответ пользователю синхронно, а саму обработку осуществлять асинхронно. Для решения задач подобного рода используются брокеры сообщений или очереди, которые позволяют упорядочить задачи для выполнения в единую структуру типа FIFO.

Однако при такой реализации API-Шлюзу необходимо предусмотреть возможность отслеживания состояния загрузки видео для клиента. Эта проблема может быть решена различными способами: поллингом состояния или оповещени-

ем клиента о его изменении с помощью технологии WebSocket. Но главное, что при таком способе решения проблемы единственным источником истины является состояние в базе данных, а способ обработки данных не имеет значения.

Обновленная схема системы с учетом новых сущностей представлена на рисунке 5.

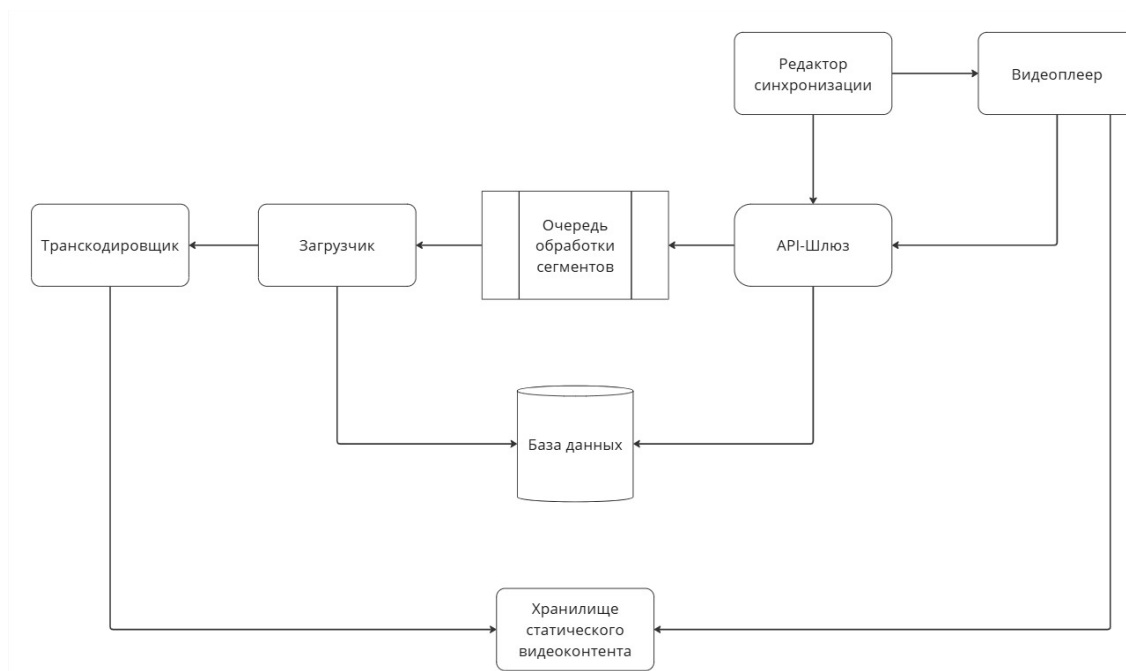


Рис.2.5. Схема системы с учётом базы данных и очередью обработки видеосегментов и сервисом-загрузчиком

Важно уточнить, что в системе также должен быть предусмотрен и механизм очищения созданных и неиспользуемых сессий загрузки и видео, так как, помимо лишней информации в базе данных, создаются дополнительные ресурсы в сервисе-транскодировщике, и при их неконтролируемом росте система теряет всякую возможность масштабирования. Для решения проблем такого рода используются планировщики задач (англ. job scheduler). Планировщик с определённой периодичностью выполняет часть бизнес логики приложения, которая требует периодичности, а затем прекращает свою работу. В текущем проекте планировщику необходимо периодически уничтожать неиспользуемые сессии как ресурс в транскодировщике, так и информацию о них и связанных сущностях в базе данных. Соответственно, такая функциональность структурно должна быть расположена где-то между сервисом-загрузчиком и сервисом-транскодировщиком, если исходить из схемы системы. Запуск планировщик как отдельный процесс операционной системы является дорогим решением, так как требует дополнительных ресурсов на среду выполнения и ещё одно подключение к базе данных. Поэтому

оптимальным решением будет расположить логику работы планировщика внутри сервиса-загрузчика, так как он будет содержать пул соединений к базе данных, а также иметь программную реализацию коммуникации с сервисом-транскодировщиком. В таком случае при выборе технологии для реализации сервиса-загрузчика необходимо отталкиваться от возможности планирования задач.

2.4. Инструменты реализации компонентов системы

Теперь, когда структура системы определена и включены все основные компоненты, необходимо перейти к выбору технологий для реализации сервисов. В рамках данного этапа будут определены инструменты и программные средства для реализации сервисов системы, их масштабирования и безопасности, а также средства коммуникации между ними. Для это рассмотрим особенности каждого из сервисов и определим соответствующие технологии.

2.4.1. Выбор СУБД

Для реализации серверной части системы необходима система управления базами данных, предназначенная для хранения пользовательских данных и информации о состоянии сущностей: видео, потоков и сессий загрузки. СУБД должна быть реляционной, так как для данных, которые планируется хранить, важны структурированность, чёткая схема и возможность эффективного выполнения сложных запросов и выборок с использованием соединений. Также обязательным требованием является поддержка многопоточной работы и транзакционности, так как доступ к данным будет осуществляться в нескольких потоках.

Из решений с открытым исходным кодом таким требованиям в наибольшей степени соответствует PostgreSQL. Эта СУБД обладает развитой поддержкой транзакций, гарантирует целостность и изолированность данных даже при высокой конкуренции потоков и процессов. Кроме того, PostgreSQL содержит в себе возможность шардирования (техника горизонтального разделения больших баз данных на более мелкие) базы данных, что позволяет масштабировать систему в будущем.

2.4.2. Выбор брокера сообщений

Для эффективной реализации асинхронной обработки сегментов необходимо использовать брокер сообщений. Основными требованиями к брокеру являются: надёжная доставка сообщений, подтверждение очереди о принятии сообщений, а также гарантия согласованности данных в случае отказов или временной недоступности отдельных компонентов. Важность перечисленных требований объясняется тем, что сегменты видео содержат определённый порядок, и даже потеря одного из них приведёт к повреждённому видео.

Из решений с открытым исходным кодом указанным наиболее полно соответствует требованиям брокер RabbitMQ. RabbitMQ использует стандартный протокол AMQP и предоставляет встроенные механизмы подтверждения принятия сообщений и повторной отправки в случае сбоев, что позволяет гарантировать доставку и сохранять согласованность данных во всех асинхронных сценариях системы.

2.4.3. Выбор инструментов для реализации API-Шлюза

API-шлюз – это основная точка входа в систему, которая обеспечивает маршрутизацию запросов, аутентификацию пользователей, управление сессиями и доступ к состоянию видео, потоков, сессий. Ключевая задача API-шлюза – возвращать результат пользователю как можно быстрее, минимизируя задержки при обработке запросов, чтобы не блокировать пользовательские сценарии. Для этого он не ждет завершения длительных операций, таких как транскодирование, а выполняет обработку асинхронно, передавая задачи в очередь RabbitMQ, и возвращает пользователю подтверждение о приеме данных, а также формирует ссылки на статические файлы видеоконтента. Взаимодействие с сервисом происходит посредством HTTP. Характеристика сервиса выше говорит о том, что Node.js будет оптимальным выбором для API-шлюза, поскольку его неблокирующий ввод/вывод позволяет эффективно обрабатывать большое количество одновременных запросов без создания дополнительных потоков, что значительно снижает нагрузку на сервер и позволяет без проблем масштабироваться горизонтально, также учитывая, что многие модули Node.JS написаны на низкоуровневых языках программирования.

2.4.4. Выбор инструментов для реализации сервиса-загрузчика

Сервис-загрузчик отвечает за прием поступающих сегментов видео, их валидацию, отправку на транскодирование, а также управление сессиями загрузки, включая их удаление после истечения срока хранения, и обновление состояния видео, потоков, сессий, включая отслеживание их консистентности. Он должен поддерживать обработку бинарных данных, интегрироваться с брокером сообщений для принятия задач, а также иметь высокую производительность, так как является самым нагруженным по части бизнес-логики. Также сервис включает планировщик, который автоматически удаляет неиспользуемые или просроченные сессии загрузки, освобождая ресурсы и поддерживая актуальность данных. Java является оптимальным выбором, так как поддерживает многопоточность и оптимальный баланс между уровнем абстракции и производительностью языка. В качестве фреймворка лучше всего использовать Spring Boot, который предоставляет удобные механизмы для интеграции с RabbitMQ. Этот фреймворк также содержит удобный механизм для периодического выполнения задач - Spring Task Scheduler, который позволяет эффективно управлять автоматическими процессами без нагрузки на основные бизнес-операции.

2.4.5. Выбор инструментов для реализации сервиса-транскодировщика

Сервис-транскодировщик отвечает за обработку, транскодирование загруженных сегментов видео, объединение их в полноценное видео, которое состоит из статического контента: манифестов для потокового вещания в форматах HLS и MPEG-DASH и соответствующих чанков, а также отправку полученных файлов на сервис-хранилище через FTP. Сервис и принадлежащие ему процессы управляемы извне посредством HTTP. В основе сервиса должен быть непосредственно транскодировщик. Оптимальным выбором в данной связи будет FFmpeg – мощное средство кодирования и транскодирования видео, которое является стандартом и фактически монополистом в области мультимедийной обработки, предоставляя поддержку всех популярных кодеков, контейнеров и форматов трансляции. Благодаря неблокирующему вводу/выводу (Non-blocking I/O) в Node.js, сервис может одновременно обрабатывать и передавать по FTP большое количество файлов, не блокируя выполнение других задач. Это позволяет масштабировать транскодирование.

2.4.6. Выбор инструментов для реализации сервиса-хранилища статического видеоконтента

Сервис-хранилище статического видеоконтента отвечает за прием транскодированных видеофайлов через FTP и их раздачу пользователям по HTTP. Он также должен обеспечивать быструю обработку нескольких запросов одновременно. Как в случае с API-Шлюзом, неблокирующий ввод/вывод, поддерживаемый по умолчанию в Node.js, выделяет его как оптимальное средство для соблюдения поставленных требований. А в совокупности с Express.js позволяет поддержать раздачу статического контента без дополнительных настроек.

2.4.7. Выбор инструментов для реализации клиентских приложений

Клиентские приложения могут быть разработаны только на JavaScript, поскольку это единственный язык программирования, поддерживаемый всеми современными и устаревшими браузерами. Видеоплеер и редактор синхронизации характеризуются большим количеством динамических состояний и асинхронных операций, что повышает риск возникновения ошибок. Для снижения вероятности ошибок приложение должно основываться на иммутабельности данных и использовать эффективный менеджер асинхронных операций и событий. Таким требованиям соответствует набор инструментов React, Redux и Redux-Saga, которые позволяют выстраивать масштабируемую древовидную структуру: React – компонентов, Redux – состояний, Redux-Saga – событий. React обеспечивает компонентную архитектуру и декларативный подход, Redux управляет состоянием на основе иммутабельности, а Redux-Saga эффективно управляет асинхронными операциями на основе генераторов и собственного менеджера.

Обновленная схема системы с учётом всех выбранных технологий и протоколов коммуникации представлена на рисунке 2.6.

2.4.8. Выбор инструмента балансировки нагрузки

Несмотря на все описанные выше оптимизации и разграничения ответственности, риск выхода из строя всей системы всё равно есть. Если один из сервисов не справится с нагрузкой и резко прекратит работу, система, как минимум, перестанет работать ожидаемо. Даже при обработке всех возможных ошибок в сервисах такая вероятность никогда не равняется нулю. Чтобы минимизировать эти риски, необхо-

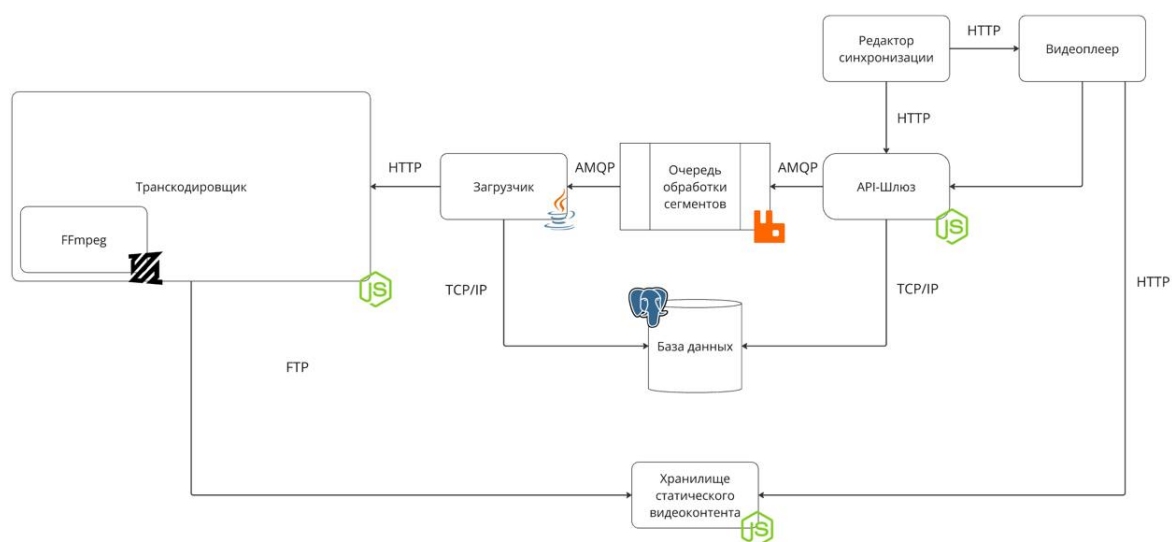


Рис.2.6. Схема системы с учётом с определёнными технологиями

можно реплицировать существующие сервисы. В таком случае при выходе из строя одного из них остальные будут работать, как и прежде, а его нагрузку распределят между собой остальные сервисы. Однако при таком подходе необходим механизм, который бы принимал все запросы от клиента сервиса и распределял трафик между такими репликациями. Для решения этой задачи необходимо использовать балансировщик нагрузки. Лидером среди них является NGINX, который также является и решением с открытым кодом. В текущем проекте балансировщик может быть использован для каждого сервиса, который будет содержать открытый порт для коммуникации с ним.

Под это описание подходят все сервисы, кроме загрузчика. Его можно реплицировать без добавления балансировщика нагрузки, так как каждая реплика будет читать сообщения из очереди независимо. Однако важно учитывать, что в таком случае, каждая реплика этого сервиса не должна содержать планировщик, который очищает ресурсы просроченной сессии. Планировщика достаточно расположить на одной станда-лон-реплике (англ. standalone). Чтобы это сделать, необходимо реализовать инициализацию планировщика на основе значения переменной окружения. В таком случае мы сможем регулировать планировщик и реплику его запуска.

2.4.9. Выбор инструмента кэширования

Ещё одна проблема может возникнуть с реплицированием сервиса-транскодировщика, так как он порождает процессы FFmpeg, которые привязаны к

конкретной реплике. Это значит, что после запроса на создание процесса транскодирования для какой-то сессии загрузки, все последующие запросы должны маршрутизироваться на конкретную реплику транскодировщика. Кроме того, реплики сервиса-загрузчика равномерно между собой будут распределять задачи из очереди. Это значит, что разные реплики загрузчика могут работать с одинаковыми сессиями и необходимо синхронизировать между ними маршрутизацию до определенной реплики транскодировщика. Для решения этой проблемы необходимо использовать сервис для кэширования типа “Ключ=Значение”, чтобы реплики могли делиться между собой информацией о маршруте до нужной реплики. Лидером среди таких решений, является СУБД Redis. Помимо прочего, такая маршрутизация требует и дополнительной настройки со стороны NGINX, которую тоже необходимо предусмотреть.

Схема системы с учётом необходимости реплицирования и балансировки нагрузки представлена на рисунке 2.7.

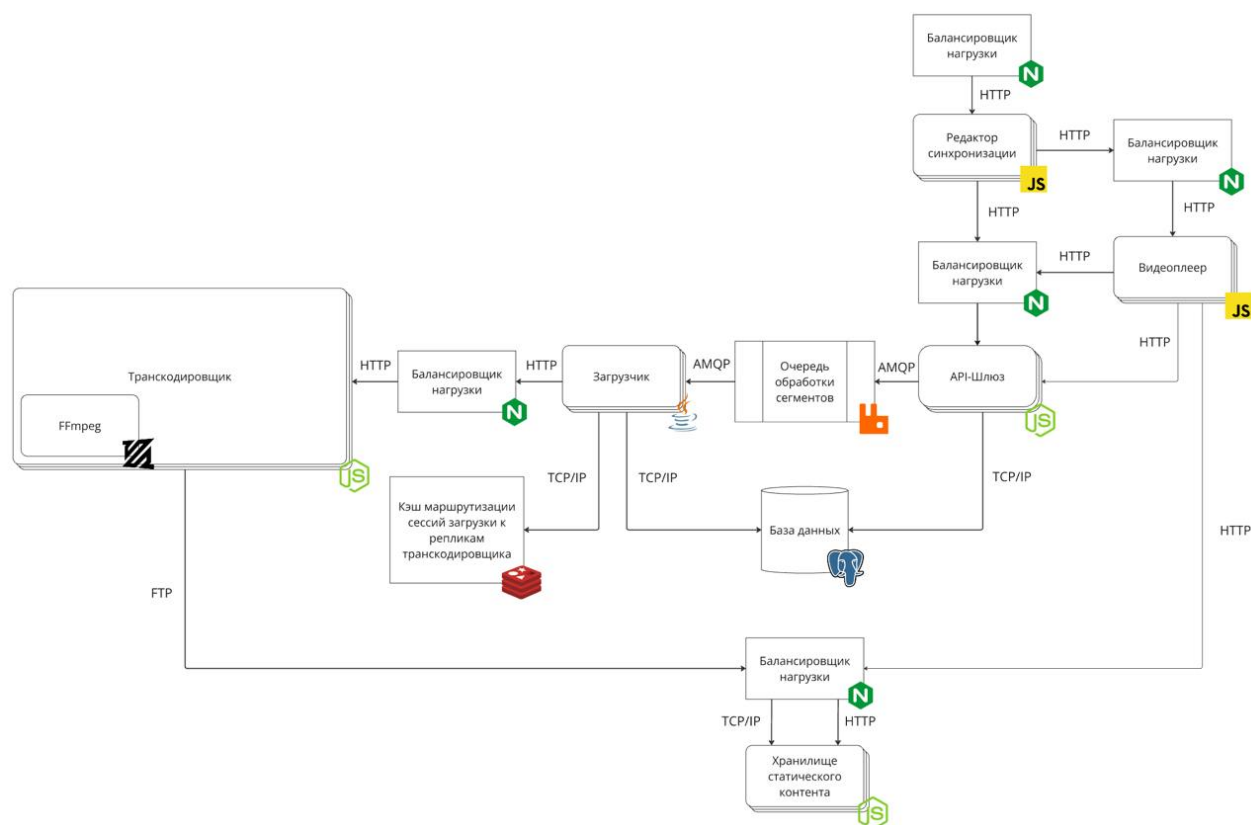


Рис.2.7. Схема системы с учётом с определёнными технологиями

2.4.10. Обеспечение безопасности межсервисной сетевой коммуникации

Важно учитывать, что при такой реализации без дополнительного конфигурирования и разработки будет огромное количество открытых TCP/IP портов,

что не является безопасным. Для того, чтобы решить эту проблему, необходимо создать приватную сеть для межсервисного взаимодействия и оставить открытыми только те порты, на которые клиент может безопасно отправлять запросы. Разделим на схеме стрелки, определяющие межсервисную коммуникацию, по признаку приватности/публичности на основе цвета. Обновлённая схема представлена на рисунке 2.8.

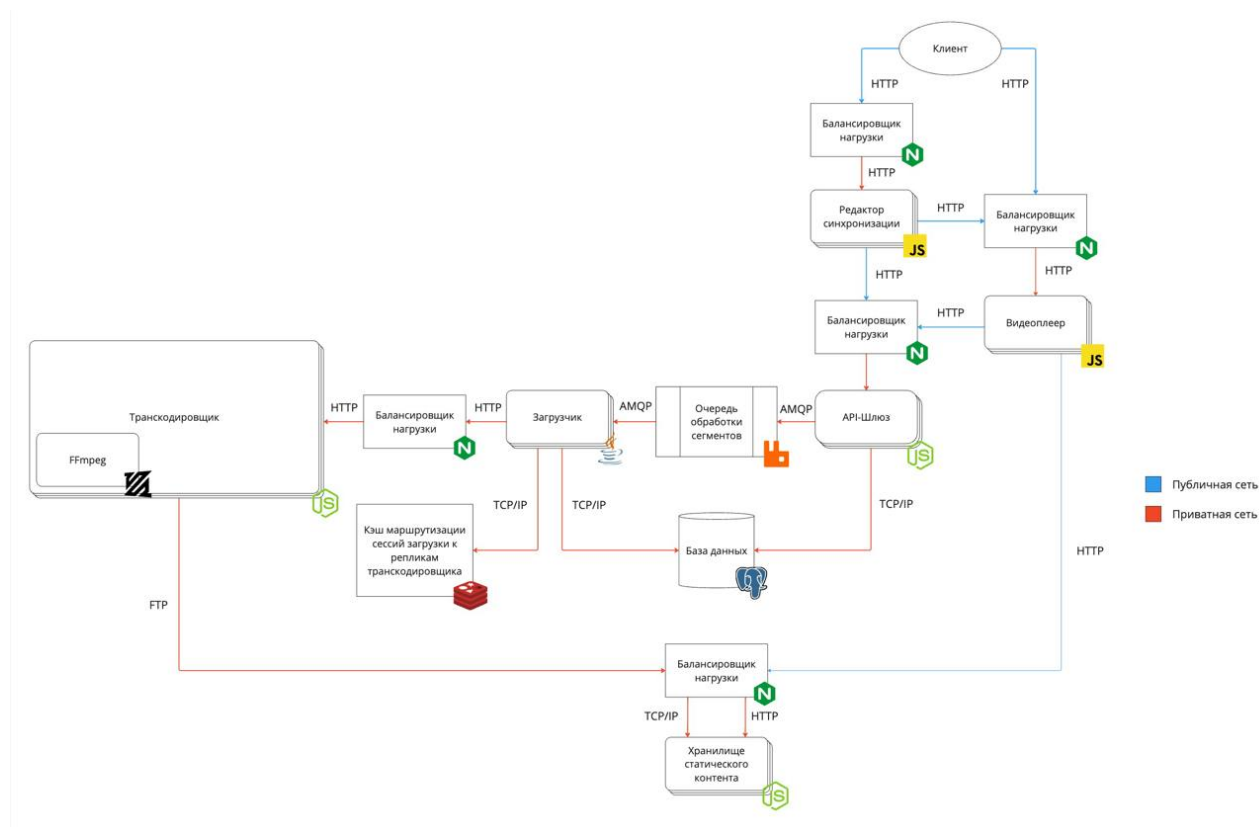


Рис.2.8. Схема системы с учётом с определёнными технологиями

2.5. Выводы

В данной главе разработана архитектура системы распределённой сегментированной загрузки и раздачи видеоконтента. Формирование архитектурных решений было основано на анализе требований к масштабируемости, отказоустойчивости, адаптивности системы к нагрузке и обеспечению бесперебойного взаимодействия пользователей с приложением. В ходе проектирования были выделены основные функциональные компоненты системы, определены методы взаимодействия между компонентами, механизмы асинхронной обработки данных, способы передачи видеоконтента и оптимальные технологии для реализации каждого сервиса, а также определены способы и технологии для балансирования

нагрузки между ними и поддержания безопасности при межсервисном взаимодействии.

ГЛАВА 3. РАЗРАБОТКА СИСТЕМЫ

3.1. Разработка API-шлюза

3.1.1. Разработка модуля авторизации

Модуль аутентификации является неотъемлемой частью API-шлюза и выполняет важные функции по обеспечению безопасности, идентификации и контроля доступа пользователей к ресурсам системы. Его основная задача – предоставлять механизм регистрации, аутентификации, управления сессиями и обновления токенов, обеспечивая безопасное взаимодействие между клиентскими приложениями и серверными компонентами. Для того, чтобы мы могли идентифицировать пользователя на серверной стороне, проводить его аутентификацию и валидировать права на модификацию информации о видео, загрузку сегментов видео в соответствующую сессию, то есть проводить авторизацию.

Кроме того, для того, чтобы API-Шлюз имел хорошую поддержку в будущем и позволял дополнять его различной логикой, необходимо унифицировать процесс авторизации и получения информации.

В качестве основного фреймворка для разработки серверных приложений в среде исполнения Node.js оптимальным решением будет использовать NestJS, так как он базируется на низкоуровневом Express и архитектуре клиентского фреймворка Angular, что делает его достаточно удобным и эффективным инструментом.

Одним из наиболее безопасных и масштабируемых способов проводить идентификацию и авторизацию является использование протокола OAuth 2.0. Структурно он включает в себя следующие основные компоненты:

- Ресурсный владелец (Resource Owner) — пользователь, владеющий доступом к защищённому ресурсу;
- Клиент (Client) — приложение, запрашивающее доступ к ресурсу от имени пользователя;
- Сервис авторизации (Authorization Server) — компонент, выполняющий аутентификацию пользователя и выдающий токены доступа;
- Сервис ресурсов (Resource Server) — сервер, на котором хранятся защищённые данные и который предоставляет доступ на основании валидного токена.

Работа протокола заключается в следующем: клиент перенаправляет пользователя на сервис авторизации, где тот подтверждает согласие на предоставление доступа. После этого сервис авторизации выдает клиенту токен доступа (access token) — временный цифровой ключ, позволяющий обращаться к защищённым ресурсам на сервере ресурсов. Поскольку access token имеет ограниченное время действия, вместе с ним может быть выдан и токен обновления (refresh token) — специальный долгоживущий токен, с помощью которого клиент может получить новый access token без повторного участия пользователя. Таким образом, OAuth 2.0 позволяет безопасно и удобно предоставлять ограниченный по объёму и времени доступ к данным, избегая необходимости передачи паролей сторонним приложениям.

Основное преимущество OAuth 2.0 заключается в масштабировании. В текущем проекте в качестве сервера ресурсов выступает рассмотренный API-Шлюз. С использованием этого протокола появляется возможность задействовать разные серверы авторизации для него. В рамках текущего проекта остановимся на одном сервере авторизации, который будет реализован ниже - это вполне соответствует функциональным требованиям системы. Однако при реализации данного протокола необязательно располагать сервис авторизации и сервис ресурсов независимо. Они могут быть частью общего сервиса - API-Шлюза. Это позволит унифицировать процесс авторизации и получения информации об идентифицированном пользователе.

Для реализации модуля необходимо предоставить пользователю возможность проводить регистрацию, аутентификацию и обновление токенов. Так как модуль не представляет какой-то конкретной бизнес логики проекта, его необходимо расположить на отдельном контроллере AuthController - это позволит вынести логику на отдельный слой API.

Для эндпоинтов, которые требуют аутентификации, необходимо реализовать механизм, который бы автоматически её проводил и предоставлял информацию о пользователе из базы данных. Эту логику можно изолировать в декоратор AuthGuard. Логику, связанную непосредственно с генерацией и валидацией JWT токенов, необходимо вынести в отдельный сервис JwtService для соблюдения принципа единственности ответственности. Общую бизнес-логику AuthController и AuthGuard необходимо вынести в общий сервис AuthService для избежания повторения кода.

Представим взаимодействие между описанными выше классами в виде UML-диаграммы для наглядности. Схема UML модуля авторизации представлена на рисунке 3.1.

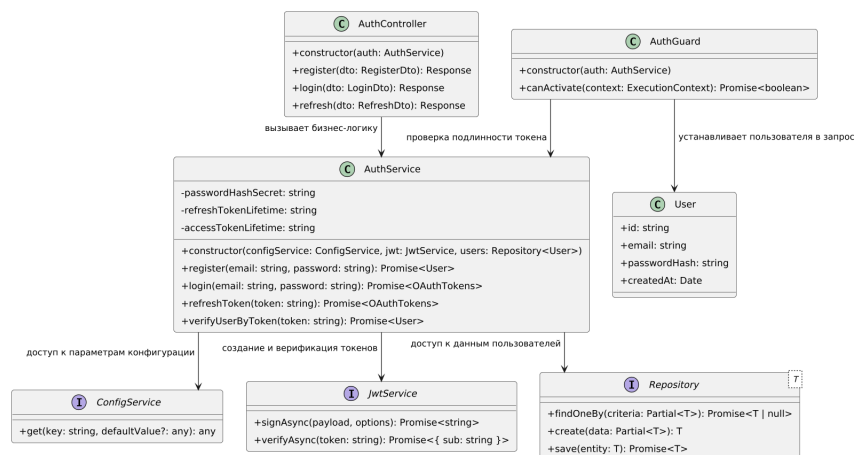


Рис.3.1. UML-диаграмма модуля авторизации

Из диаграммы видно, что мы можем построить граф зависимостей между классами, благодаря описанию их конструкторов. Более того, не содержат дополнительных аргументов для их конфигурации в конструкторах. Вся необходимая конфигурация инкапсулирована в `ConfigService`. Это значит, что представленные классы имеют чистые зависимости между друг другом. Такая структура позволяет использовать механизм инъекции зависимостей (англ. *dependency injection*). Такой механизм поддерживается по умолчанию в NestJS, нам лишь необходимо расставить над классами соответствующие декораторы, выполняющие функционал разметки для определения роли класса или метода в NestJS приложении.

На диаграмме также изображены дополнительные классы `ConfigService` и `Repository`. Первый из них представляет собой сервис для централизованного доступа к конфигурационным параметрам. Этот класс представлен в модуле `"@nestjs/common"`. С помощью специальных конфигурационных объектов из этой же библиотеки можно преобразовывать переменные окружения в удобный формат для конкретного приложения, например, преобразовать литерал из цифр в числовой тип данных. Кроме того, NestJS позволяет не задавать переменные окружения непосредственно в операционной системе, а перечислить их в файле `.env` проекта. Переменные окружения позволяют отделить конфигурационные параметры от исходного кода, что делает приложение более гибким и безопасным. Это упрощает адаптацию приложения к различным средам без изменения кода, что даст нам возможность управлять конфигурацией проекта в дальнейшем при развёртывании системы. А `Repository` в контексте NestJS и библиотеки TypeORM представляет

собой абстракцию для работы с базой данных. Он предоставляет набор методов для выполнения операций с сущностями, таких как создание, поиск, обновление и удаление данных, без необходимости напрямую писать SQL-запросы.

При регистрации пользователь отправляет email и пароль через POST-запрос на `"/auth/register"`. Контроллер `AuthController` обрабатывает этот запрос и вызывает метод `register` сервиса `AuthService`, который сначала проверяет, существует ли уже пользователь с таким почтовым адресом в базе данных. Если пользователь найден, выбрасывается исключение. В противном случае создается новый пользователь с хэшированным паролем (с использованием HMAC SHA-256 и секретного ключа), и сохраняется в базе данных. Результат регистрации пользователя возвращается пользователю. Все эти процессы представлены на диаграмме последовательности регистрации на рисунке 3.2.

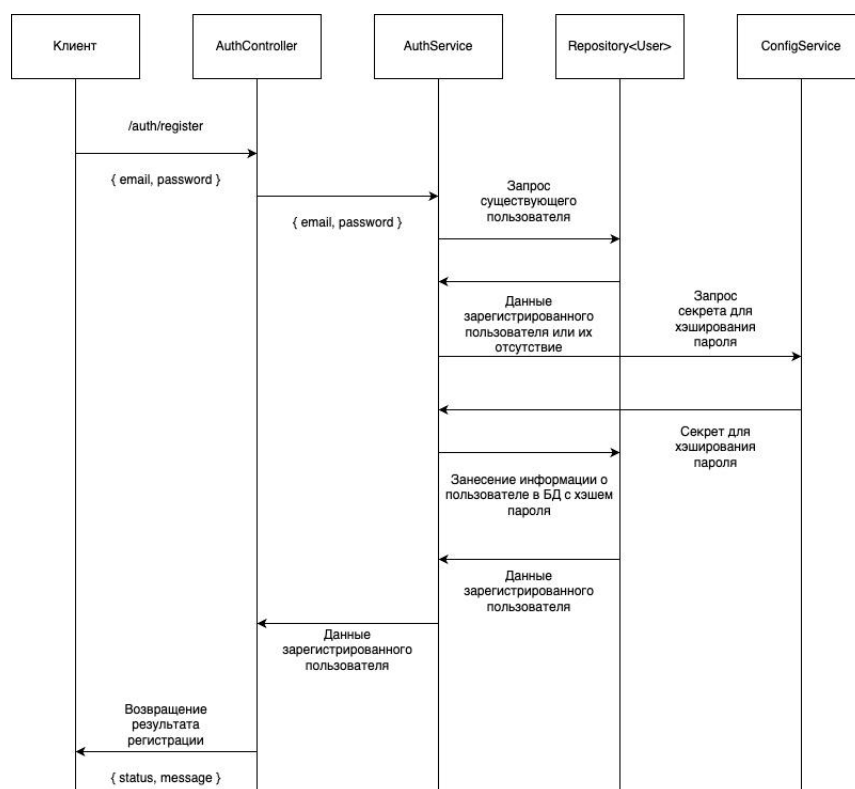


Рис.3.2. Диаграмма последовательности процессов регистрации пользователя

Процесс аутентификации пользователя также начинается с запроса с email и паролем на эндпоинт `"/auth/login"` где контроллер передает данные в сервис аутентификации. Сервис проверяет наличие пользователя в базе данных и сравнивает хэш пароля с сохраненным значением. Перед тем, как генерировать хэш, сервис обращается к `ConfigService` для получения секрета. Если проверка успешна, генерируются два токена с помощью `JwtService`: `accessToken` для доступа к защищенным ресурсам и `refreshToken` для обновления `accessToken` по истечении его

срока действия. Эти токены возвращаются клиенту, а в случае ошибки отправляется соответствующее сообщение об отказе в доступе. Все описанные процессы выше представлены на диаграмме последовательности аутентификации на рисунке 3.3.

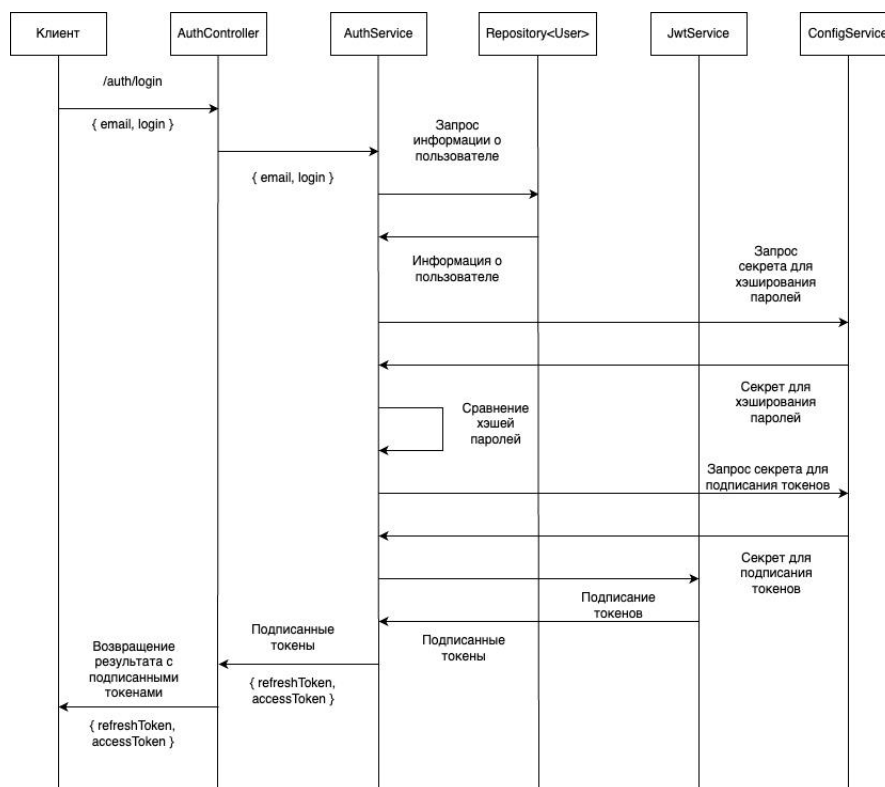


Рис.3.3. Диаграмма последовательности процессов аутентификации пользователя

Внешним входом для использования модуля авторизации в других частях API-Шлюза является класс AuthGuard. Другие контроллеры могут его использовать как часть промежуточного слоя обработки HTTP запроса определённых эндпоинтов. Он наследуется от базового механизма защиты NestJS - PassportAuthGuard со стратегией "local". В нём переопределяется метод canActivate и дополняется специфичной логикой проекта. При каждом запросе AuthGuard извлекает заголовок авторизации (Authorization) из входящего HTTP-запроса и проверяет его наличие. Затем с помощью метода verifyUserByToken из AuthService происходит верификация токена, и если пользователь успешно идентифицирован, он добавляется в объект запроса (request.user), что позволяет другим обработчикам эндпоинтов API-Шлюза обращаться к аутентифицированному пользователю. Если токен отсутствует, некорректен или верификация не удалась, выбрасывается исключение UnauthorizedException, которое приводит к возврату ошибки со статусом 401 (Unauthorized).

3.1.2. Разработка головного модуля

Головной модуль приложения отвечает за обработку запросов пользователей, связанных с их данными и видеоконтентом. Точкой входа в этот модуль является класс `AppController`, который содержит набор HTTP-эндпоинтов. В частности, модуль должен содержать логику, которая отвечает за возвращение информации об аутентифицированном пользователе, создание новой записи видео, потоков и инициализации загрузочных сессий для последующей загрузки сегментов видеоконтента, а также в нём должна содержаться логика обработки загрузки сегментов видеоконтента, осуществляя проверку прав доступа и корректности данных, передаваемых пользователем. Кроме того, модуль должен предоставлять возможность для пользователя отслеживать прогресс и статус загрузки потоков видео. Перечисленную выше функциональность можно разделить на следующие эндпоинты:

- `me` - получение информации о текущем аутентифицированном пользователе по токену доступа (`accessToken`);
- `upload` - загрузка сегмента видео для определённой сессии загрузки;
- `state` - получение текущего состояния видео и связанных потоков, сессий загрузки;
- `create-video` - создание видео и связанных потоков, сессий загрузки.

Все перечисленные выше эндпоинты являются REST и могут использовать JSON формат для взаимодействия между сервером и клиентом, кроме `"upload"`. Этот эндпоинт принимает в качестве тела запроса данные в виде последовательности байт. Он должен быть настроен таким образом, чтобы принимать HTTP-запросы с MIME-type равным `"application/octet-stream"`. Сама разметка сегмента видео должна быть определена в заголовках запроса, что предусматривает их семантика.

Основная работа головного модуля API-Шлюза заключается в обработке пользовательских запросов, первичной валидации данных полученных от пользователей, а также возврате в безопасном формате данных клиентам. При обработке эндпоинта `"me"` головной сервис должен вернуть безопасное представление аутентифицированного пользователя или вернуть результат с ошибкой при отсутствии такового. Необходимость в отдельном эндпоинте `"me"` объясняется тем, что клиентскому приложению в самом начале своего жизненного цикла необходимо определить текущий статус аутентификации пользователя. Это связано со сроком жизни токенов аутентификации - он ограничен. На уровне клиента узнать, про-

срочен ли тот или иной токен, нет возможности, поэтому необходимо отправить предварительный запрос на сервер.

Вся перечисленная логика выше не может находиться в контроллере `AppController`, потому что он должен отвечать только за работу с HTTP запросами. Отделение бизнес-логики от контроллера и её вынесение в отдельный класс `AppService` необходимо для того, чтобы разделить интерфейс взаимодействия с клиентами и непосредственную обработку данных. В `AppService` следует вынести работу с базой данных с помощью ранее рассмотренных репозиторий `TypeORM`, логику создания видео, управления потоками и загрузочными сессиями, а также обработку входящих сегментов видеоконтента.

Асинхронную передачу данных, связанную с загрузкой видеофрагментов, логично вынести в отдельный класс `ChunkExchangeService`, который будет использовать брокер сообщений `RabbitMQ` для передачи задач на выполнение сервису-загрузчику.

Важно также учитывать, что возвращать информацию, полученную из базы данных, в исходном виде нельзя, так как это не соответствует требованиям безопасности системы. Перед отправлением ответа пользователю данные необходимо предварительно преобразовать. За это будет отвечать `VideoConverterService`.

UML-диаграмма модуля с учётом разделения бизнес логики на соответствующие классы представлена на рисунке 3.4.

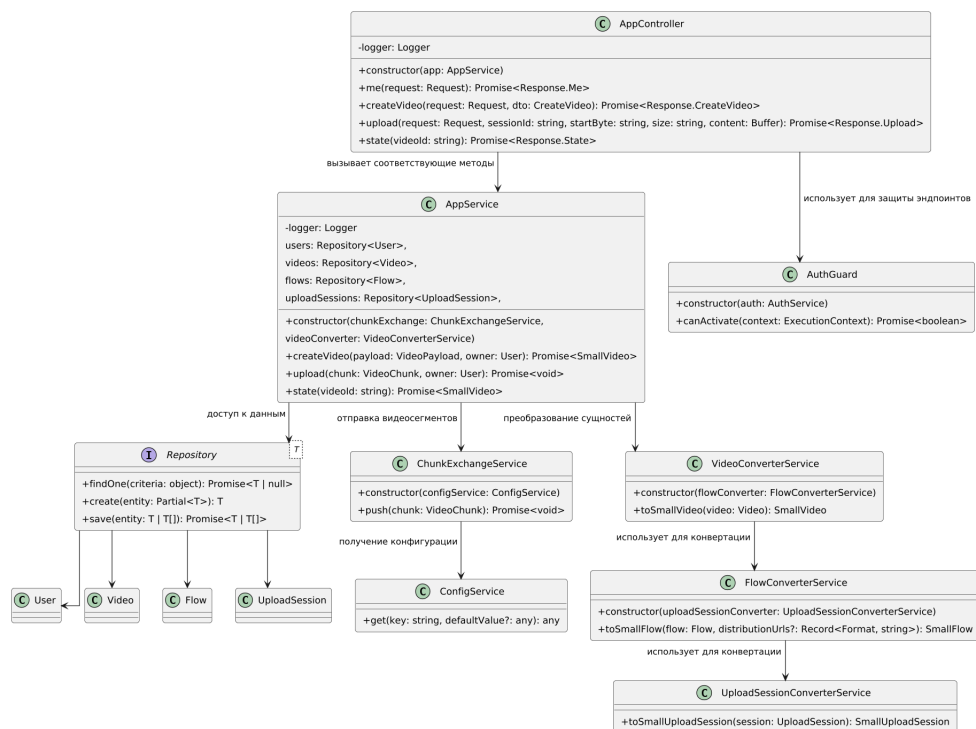


Рис.3.4. UML-диаграмма головного модуля API-Шлюза

Рассмотрим каждый из эндпоинтов в отдельности ниже.

Эндпоинт "me" предназначен для получения информации о текущем аутентифицированном пользователе. Он использует защитный механизм AuthGuard, описанный ранее в данной главе, который отвечает за извлечение и верификацию авторизационного токена из запроса. После успешной проверки токена AuthGuard получает экземпляр пользователя из базы данных и прикрепляет его к объекту запроса. Далее в самом эндпоинте происходит конвертация данных пользователя в специальный объект, который содержит только безопасные поля, исключая конфиденциальные данные - хэш пароля, создавая тем самым DTO (Data Transfer Object). Использование AppService здесь не требуется, так как вся бизнес-логика этого эндпоинта уже инкапсулирована в AuthGuard.

Эндпоинт "create-video" обрабатывает запросы от клиента с телом, содержащим название видео (title), описание (description) и массив размеров потоков видеоконтента (totalBytesList). Он также определяет и количество итоговых потоков, которые будут загружены для этого экземпляра видео. Этот эндпоинт также использует защитный механизм AuthGuard, соответственно, требует от пользователя токен доступа. После успешной аутентификации экземпляр пользователя извлекается из объекта запроса и передаётся в метод createVideo класса AppService. В этом методе создаётся новая сущность видео, которая сохраняется в базе данных. Затем для каждого элемента списка totalBytesList создаются соответствующие потоки (Flow) и сессии загрузки (UploadSession). Эти сущности также сохраняются в базу данных и связываются с созданным видео. В конце выполнения метода происходит конвертация созданных сущностей в компактный объект (SmallVideo) при помощи VideoConverterService для отправки обратно клиенту. Этот сервис, в свою очередь, инъектирует как зависимость себе FlowConverterService и переводит потоки тоже в безопасные и компактные объекты (SmallFlow). А он, в свою очередь, инъектирует UploadSessionConverterService, который делает то же самое для объекта сессии загрузки. В ответе клиенту возвращается статус выполнения запроса и полученный компактный объект с информацией о созданном видео и его загрузочных потоках в соответствии с их порядком в запросе. Все процессы, описанные выше, представлены на диаграмме последовательности на рисунке 3.5.

Эндпоинт "upload" ожидает от клиента POST-запрос, содержащий идентификатор сессии загрузки (sessionId в URL), номер начального байта (x-start-byte) и размер сегмента (x-size) в заголовках запроса, которые являются разметкой загружаемого сегмента, а также бинарные данные сегмента в теле запроса. Эндпоинт

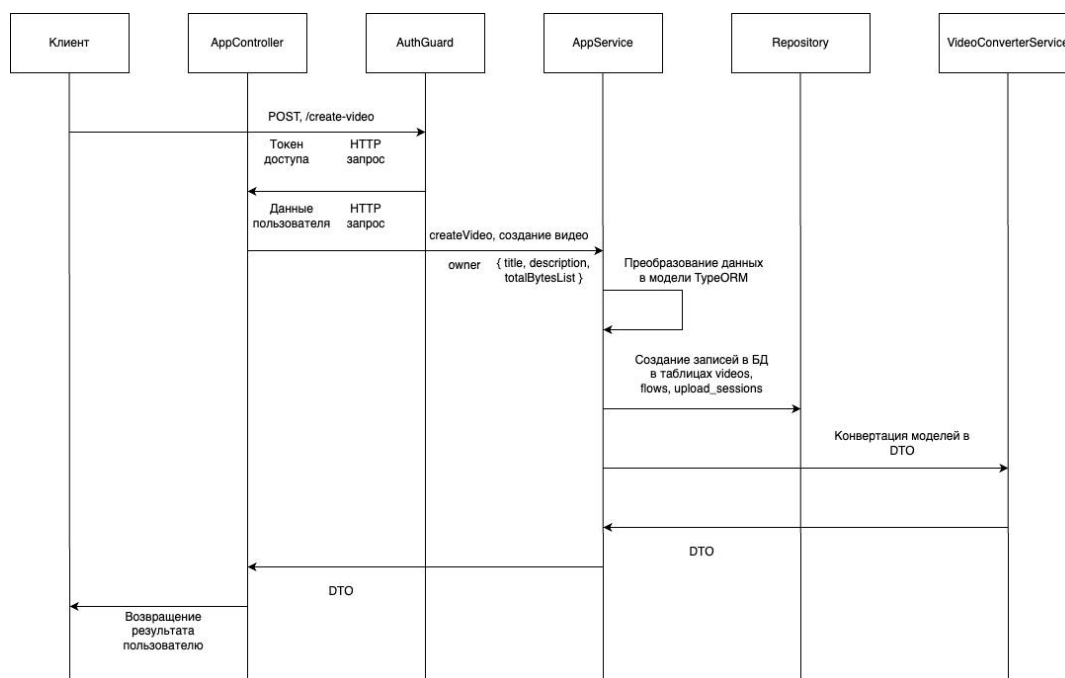


Рис.3.5. UML-диаграмма головного модуля API-Шлюза

также защищён с помощью механизма авторизации AuthGuard. После проверки авторизации контроллер преобразует заголовки с номером начального байта и размером в числовой формат и проверяет их корректность. Далее данные передаются в метод `upload` сервиса `AppService`, который извлекает из базы данных соответствующую сессию загрузки вместе с её связанными объектами видео, потока и пользователя-владельца. Запрос осуществляется с помощью `INNER JOIN` оператора `PgSQL`. Затем сервис проверяет права доступа, удостоверяясь, что сессия загрузки принадлежит пользователю, отправившему запрос. В случае успешной проверки чанки отправляются через компонент `ChunkExchangeService` в брокер сообщений `RabbitMQ` в соответствующий обменник. По завершении этих действий клиенту возвращается ответ с информацией об успешной загрузке сегмента. Все процессы, описанные выше, представлены на диаграмме последовательности на рисунке 3.6.

Процесс передачи сегмента в очередь в `ChunkExchangeService` тоже можно декомпозировать более детально. Метод `push` этого класса предназначен для асинхронной передачи сегментов в брокер сообщений `RabbitMQ` с помощью канала подключения протокола `AMQP`. Этот метод принимает сформированный объект сегмента видео. В начале метода формируются специальные заголовки сообщения, содержащие необходимую информацию для корректной обработки сегмента на принимающей стороне. После этого сообщение публикуется в соответствующий обменник, параметры которого извлекаются из `ConfigService`, то есть из соот-

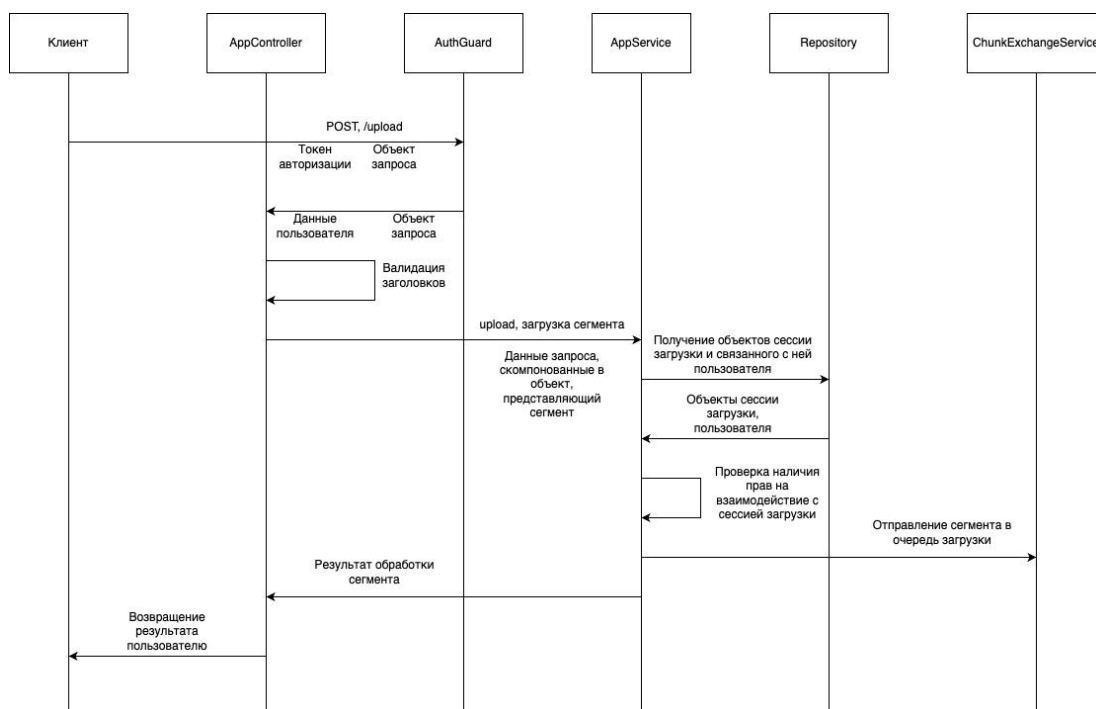


Рис.3.6. Диаграмма последовательности процессов загрузки сегмента видео

ветствующих переменных окружения, что позволяет легко изменять параметры взаимодействия с брокером без модификации кода. Далее метод ожидает подтверждения получения или отказа (ack/nack) от обработчика, после чего разрешает или отклоняет соответствующий асинхронный запрос. Диаграмма последовательности отправки сегмента в очередь сообщений представлена на рисунке 3.7.

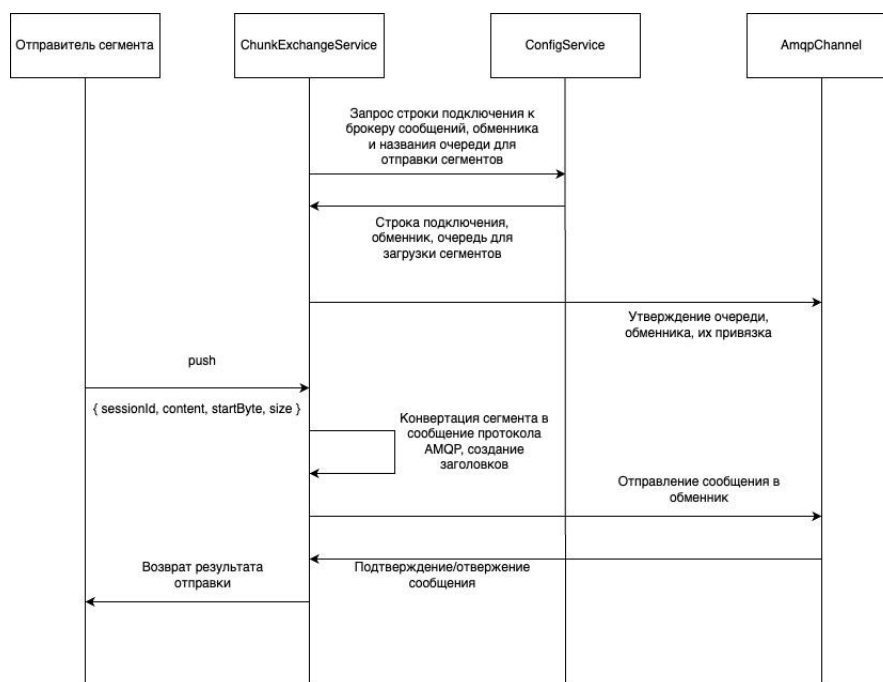


Рис.3.7. Диаграмма последовательности отправки сегмента видео в очередь сообщений

Эндпоинт "state" ожидает от клиента GET-запрос, содержащий идентификатор видеозаписи (videoId) в URL. Внутри обработчика этого запроса контроллер вызывает метод state класса AppService. Сервис выполняет поиск записи видео в базе данных по указанному идентификатору и дополнительно извлекает связанные с ним потоки и сессии загрузки. Если видеозапись не найдена, сервис генерирует ошибку NotFoundError, которая перехватывается контроллером и преобразуется в HTTP-ответ с соответствующим статусом ошибки. Если же запись видео успешно найдена, она конвертируется в компактный формат (SmallVideo) с помощью сервиса VideoConverterService и отправляется обратно клиенту.

3.2. Разработка сервиса-хранилища статического видеоконтента

Видео для адаптивного стриминга с использованием протоколов HLS и DASH представлено в виде соответствующих манифестов .m3u8 и .mpd и связанных с ними файлов-чанков формата MPEG-TS и M4S, описывающих последовательность сегментов и параметры поиска. Такая структура позволяет сохранять и передавать неизменяемые файлы непосредственно через специализированный сервер. Сохранение данных в файловой системе обеспечит высокую скорость доступа к манифестам и чанкам видеоконтента, а HTTP-сервер, отвечающий за их раздачу, позволит браузерам кэшировать получаемые ресурсы.

Таким образом, этот сервис разделяется на два независимых модуля: UploadModule - модуль загрузки, DistributionModule - модуль раздачи. Оба модуля имеют общий ресурс в виде файловой системы, где статические данные будут храниться. Это означает, что путь к директории в файловой системе, где будут храниться данные, необходимо вынести в отдельную переменную окружения - ROOT_PATH, которую будут использовать оба модуля. Несмотря на это, у каждого модуля будет использоваться свой экземпляр ConfigService, так как настройки каждого из них специфичны.

Как ранее отмечалось, для межсерверной передачи файлов оптимальным способом является использование FTP протокола. Соответственно, модуль загрузки должен поддерживать FTP сервер, который готов обрабатывать операции взаимодействия с файлами. Node.js библиотека "ftp-srv" с корневым классом FtpSrv поддерживает эту возможность. Этот протокол по умолчанию раздаёт порт 20 для установления управляющего соединения. Этот порт достаточно легко вынести в переменную окружения для конфигурации. Однако протокол также динамически

может использовать порты для свободных подключений в диапазоне от 1024 до 65535. Этот диапазон необходимо также сконфигурировать и ограничить, чтобы сделать работу сервиса более предсказуемым, конфигурируемым и надёжным.

Для модуля с раздачей статики необходимо учитывать только путь до корневой директории, где хранятся файлы, а также порт HTTP-сервера. Для запуска сервера с раздачей статического контента оптимальным решением будет использовать модуль `ServeStaticModule`, который поставляется из фреймворка `NestJS`, для беспрепятственной интеграции со средой. При определении `DistributionModule` описанный выше модуль необходимо передать в массив значений `imports` в объект декоратора `Module` для его инициализации. Важно также учесть и политики безопасности сервиса. Он должен возвращать HTTP ответ только разрешённым источникам, в текущем проекте - только плееру. Для этого также необходимо определить переменную окружения - `PLAYER_ORIGIN`. Кроме того, также необходимо правильно настроить процесс загрузки двух модулей в одном проекте, так как модуль загрузки не должен раздавать никаких дополнительных портов, кроме системных для FTP. Для этого при инициализации модуля с помощью `NestFactory` будем использовать `createApplicationContext` вместо метода `create`, как для обычных HTTP приложений. Программный код старта приложения с учетом требований представлен ниже.

Листинг 3.1

Код старта сервиса-хранилища

```

5  async function bootstrap() {
    const [distribution] = await Promise.all([
      NestFactory.create(DistributionModule),
      NestFactory.createApplicationContext(UploadModule),
    ]);

    distribution.enableCors({
      origin: [
10  PLAYER_ORIGIN,
      ],
      methods: [
        'GET',
15  ],
    });

    await distribution.listen(process.env.DISTRIBUTION_PORT
      ?? 4050);
  }

```

20

`bootstrap();`

UML-диаграмма сервиса с учётом требований выше представлена на рисунке 3.8.

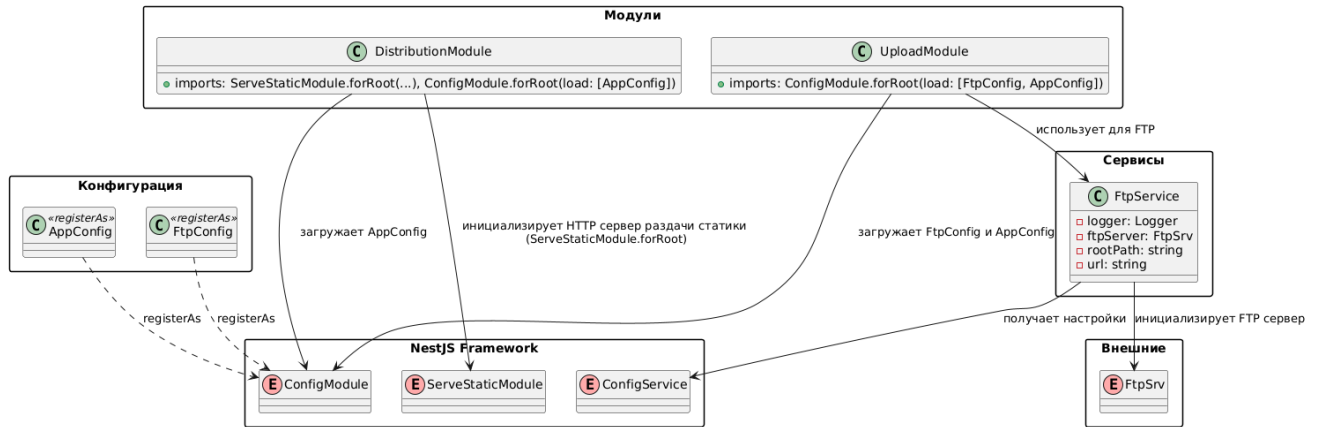


Рис.3.8. UML-диаграмма сервиса-хранилища статического видеоконтента

3.3. Разработка сервиса-транскодировщика

Перед тем, как загрузить статический видеоконтент на сервис-хранилище, его необходимо транскодировать в необходимые видео и аудио кодеки и преобразовать в форматы DASH и HLS. Это является основной зоной ответственности этого сервиса. Важно также учитывать, что сервис, являясь частью системы, должен предоставлять интерфейсы для взаимодействия с другими сервисами, а также поддерживать протоколы других связанных компонентов. Это фактически означает, что сервис должен предоставлять HTTP-эндпоинты для сервиса-загрузчика, а также поддерживать FTP-клиент для передачи сформированного контента из файлов на сервис-хранилище. Последнее требование говорит о том, что сформированные файлы должны сохраняться либо с помощью абстракции, которая умеет представлять псевдо-файловую систему в оперативной памяти, либо непосредственно в файловой системе.

Как было отмечено ранее, в этом сервисе будет использоваться продукт с открытым исходным кодом - FFmpeg. Он представлен в виде набора библиотек, в том числе и программы-кодировщики, и консольных утилит. Использование этого программного осуществляется с помощью создания процесса "ffmpeg" с соответствующим набором параметров, который будет рассмотрен далее. Это означает,

что каждый такой процесс будет привязан к процессу-родителю и необходимо предусмотреть механизм маршрутизации к конкретной реплике сервиса.

Все конфигурируемые параметры необходимо вынести в переменные окружения для гибкости настройки этого сервиса. Точкой входа в сервис будет являться контроллер `AppController`. Он принимает HTTP-запросы, передаёт на выполнение главному сервису `AppService` и возвращает результат пользователю в определённом формате. Контроллер инyectирует в конструктор два сервиса: `AppService` - содержит методы, реализующие бизнес-логику работы с потоками FFmpeg, файловой системой и FTP клиентом, `ConfigService` - инкапсулирует работу с конфигурацией сервиса. Из конфигурации контроллер извлекает с помощью метода `get` извлекаются следующие параметры: `globalPrefix` - глобальный URL-префикс для API для текущей реплики, а также `host` - хост, на котором развернут текущий сервис для внешнего клиента (сервиса-загрузчика).

Рассмотрим реализацию каждого эндпоинта в отдельности ниже:

- `createFlow`: принимает запрос, извлекает из его параметров `uploadSessionId`, логирует идентификатор новой сессии для отладки работы сервиса при развертывании, вызывает метод `createFlow` экземпляра `AppService` с полученным идентификатором сессии загрузки, использует конфигурационные параметры `globalPrefix` и `host`, чтобы динамически сформировать часть URL маршрутизации к текущей реплике, и возвращает его пользователю. Пример сформированного URL: `"http://fflow:4050/r1"`;
- `deleteFlow`: принимает запрос, извлекает из его параметров `uploadSessionId`, вызывает одноимённый метод экземпляра `AppService` и передаёт в него идентификатор сервиса загрузки. Обрабатывает ошибку `FlowNotFoundError` и возвращает 404 код состояния пользователю, если поток не найден;
- `pushToFlow`: принимает в качестве тела запроса бинарные данные видео, логирует идентификатор сессии и размер полученных данных для отладки в будущем, вызывает одноименный метод объекта `AppService` для передачи данных по входящий поток FFmpeg. Как и эндпоинт выше, возвращает 404 код состояния в случае отсутствия созданного потока;
- `finishFlow`: принимает запрос на завершение текущего процесса, извлекает параметр `uploadSessionId`, вызывает одноимённый метод экземпляра `AppService`. В случае, если поток не найден, возвращает пользователю 404 код состояния.

Однако фактическое добавление `globalPrefix` для текущей реплики надо дополнительно настроить. NestJS содержит реализованный промежуточный слой, который устанавливает всему приложению глобальный префикс - `setGlobalPrefix`. Запросы на создание потока в реплику будут приходить без этого префикса, так как он будет распределяться равномерно между созданными репликами с помощью балансировщика нагрузки. Чтобы разрешить эту проблему конфигурации, необходимо написать промежуточный слой обработки запрашиваемого URL. Если URL запроса не будет начинаться с глобального префикса, он будет добавлен принудительно, чтобы мы могли обработать запросы без конкретной маршрутизации. Кроме того, код инициализации также должен обрабатывать запросы с телом в виде бинарных данных. Для этого при инициализации необходимо задать промежуточный слой, который бы обрабатывал запросы с заголовком "Content-Type": "application/octet-stream". Код инициализации приложения с учётом требований выше представлен ниже:

Листинг 3.2

Код старта сервиса-транскодировщика

```

function globalPrefixFiller(globalPrefix: string = '') {
  return (req, res, next) => {
    if (!req.url.startsWith(globalPrefix)) {
      req.url = globalPrefix + req.url;
    }
    next();
  };
}

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  app.use(bodyParser.raw({ type: 'application/octet-stream', limit: '10mb' }));
  app.setGlobalPrefix(process.env.GLOBAL_PREFIX);
  app.use(globalPrefixFiller(process.env.GLOBAL_PREFIX));

  await app.listen(process.env.PORT ?? 5050);
}

bootstrap();

```

Перед описанием главного сервиса `AppService` рассмотрим компоненты, от которых он зависит.

Для того, чтобы файлы, созданные с помощью FFmpeg, отправлялись на сервис-хранилище, необходимо реализовать FTP клиент. Основные операции и подключение к FTP серверу реализуются в библиотеке “basic-ftp” - она является единственным поддерживаемым FTP клиентом на Node.js с открытым исходным кодом на текущий момент. Для передачи файлов с помощью этого протокола необходимо каждый раз создавать новую FTP сессию, так как каждая из них содержит свой контекст. Эту логику необходимо выделить в отдельный класс FtpSession. Он содержит два поля: client - экземпляр FTP клиента из библиотеки “basic-ftp”, options - настройки подключения к FTP. А также методы: асинхронный launch - выполняет подключение к серверу, createDir - создаёт директорию на сервере по указанному пути, uploadFromDir - загружает содержимое локальной директории на FTP сервер, destroy - закрывает FTP соединение и освобождает ресурсы. Однако, чтобы напрямую не создавать FTP сессии каждый раз при использовании, так как параметры подключения будут достаточно типовыми и конфигурируемыми, логику их инициализации можно вынести в отдельный класс FtpSessionsOrchestratorService. Этот класс внедряет сервис конфигурации ConfigService и получает из него параметры подключения к FTP-серверу: хост и порт (с возможностью задать значения по умолчанию). Он имеет единственный метод create, который создаёт и запускает новую FTP-сессию, возвращая её экземпляр.

Работу с потоками FFmpeg сервисам вынесем в отдельный сервис - FFlowService. В конструкторе происходит инициализация конфигурационных параметров из ConfigService. Конструктор FFlowService инициализирует следующие параметры из сервиса конфигурации:

- videoCodec — кодек для видео (по умолчанию 'libx264');
- audioCodec — кодек для аудио (по умолчанию 'aac');
- videoBitrate — битрейт видео (по умолчанию '5000k');
- audioBitrate — битрейт аудио (по умолчанию '192k');
- dashSegmentDuration — длительность сегмента для DASH (в секундах, по умолчанию 10);
- dashSegmentFormat — формат сегмента для DASH (по умолчанию 'mp4');
- hlsTime — длительность сегмента для HLS (в секундах, по умолчанию 4);
- hlsPlaylistType — тип плейлиста HLS (по умолчанию 'vod');
- logLevel — уровень логирования FFmpeg (по умолчанию 'debug').

В качестве кодеков будут использоваться H.264 и AAC, как было определено в первой главе. Также в конструкторе инициализируется in-мемогу хранилище `flowBySessionId`, представляющее собой словарь, в котором ключом является идентификатор сессии (`uploadSessionId`), а значением — экземпляр дочернего процесса (`ChildProcess`), связанный с конкретным процессом `FFmpeg`.

Класс выше также содержит следующие методы:

- `createFlow`: формирует аргументы командной строки для `FFmpeg` с использованием билдера `ffmpegArgsBuilder`, включая входной поток (`pipe:0`) и два выходных — для DASH и HLS. Благодаря выделенному входному потоку процессы `FFmpeg` могут существовать без ограничений по времени и принимать бинарные данные асинхронно. Если для указанного `uploadSessionId` уже существует активный процесс, выбрасывается исключение `FlowAlreadyExistsError`. После успешного создания дочернего процесса `FFmpeg`, он сохраняется в `flowBySessionId`;
- `deleteFlow`: завершает активный процесс `FFmpeg` с помощью метода `killFlow`. После завершения процесс удаляется из словаря. Метод предполагает дальнейшее
- расширение с очисткой временных ресурсов;
- `pushToFlow`: записывает переданный бинарный буфер в `stdin` процесса `FFmpeg`. Если поток не существует, выбрасывается `FlowNotFoundError`;
- `finishFlow`: завершает поток корректно, закрывая `stdin` процесса и ожидая завершения выполнения через `listenDestroy`. После завершения процесса он удаляется из словаря `flowBySessionId`;
- `killFlow`: является низкоуровневым методом для принудительного завершения процесса с помощью `SIGKILL`. Аналогично `finishFlow`, он дожидается завершения и очищает записи из внутреннего хранилища;
- `listenDestroy`: подписывается на события `exit` и `close` дочернего процесса.

Главный сервис `AppService` в конструкторе из `ConfigService` извлекает имя выходного манифеста, на основе которого формируются конечные имена DASH и HLS файлов (`manifest.mpd`, `manifest.m3u8`). Этот сервис инъецирует в конструкторе следующие зависимости: ранее рассмотренные `FFlowService` и `FtpSessionsOrchestratorService`. `AppService` содержит следующие методы:

- `createFlow`: подготавливает пути директорий вывода (для DASH и HLS) и создает параметры для процесс `FFmpeg`, указав соответствующие пути

- вывода, кодеки и остальные перечисленные ранее настройки. После этого FFlowService запускает процесс FFmpeg с заданными параметрами вывода;
- **deleteFlow**: завершает активный процесс FFmpeg, связанный с сессией, и удаляет все временные файлы и директории, связанные с этой сессией из файловой системы;
 - **finishFlow**: завершает поток транскодирования через FFlowService, после чего инициирует FTP-сессию через FtpSessionsOrchestratorService, создает директорию на удалённом сервере и загружает туда все сгенерированные файлы из локальной директории сессии. После успешной отправки файлов происходит очистка локального каталога и завершение FTP-сессии;
 - **pushToFlow**: передаёт бинарные данные в stdin активного FFmpeg-процесса, делегируя эту задачу FFlowService.

Для визуализации структуры сервиса была построена UML-диаграмма классов, представленная на рисунке 3.9.

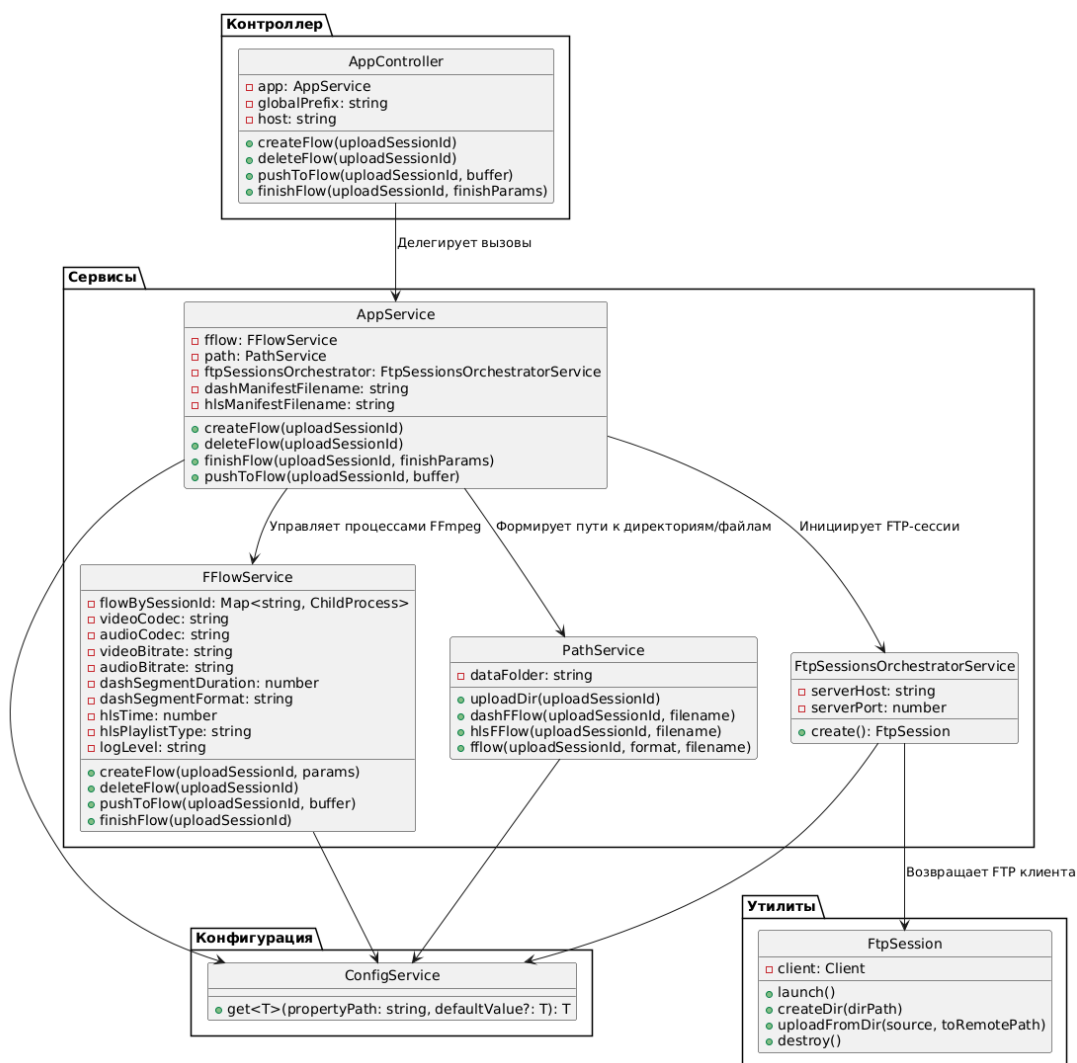


Рис.3.9. UML-диаграмма сервиса-транскодировщика

3.4. Разработка сервиса-загрузчика

Между сервисом-транскодировщиком и API-Шлюзом располагается промежуточный сервис, который занимается выполнением задач по валидации и обработке видеосегментов, актуализацией состояния видео, потоков и сессий загрузки, а также удалением просроченных видео и их ресурсов. Вся перечисленная выше логика входит в зону ответственности сервиса-загрузчика.

3.4.1. Разработка потребителя задач обработки сегментов

Как было указано ранее, API-Шлюз взаимодействует с сервисом-загрузчиком с помощью очереди RabbitMQ, передавая в неё задачи для обработки видеосегментов, а фактически - сегменты с разметкой в виде заголовков и меткой сообщения.

Для того чтобы считывать данные из очереди RabbitMQ, необходимо выделить логику, связанную с обработкой событий очереди, в отдельный класс — `ChunkConsumerService`. Этот класс ответственен за асинхронное получение сообщений из очереди, их валидацию и передачу на дальнейшую обработку.

В качестве фреймворка для реализации сервиса загрузчика был выбран Spring Boot на язык программирования Java. Он также, как и NestJS, поддерживает механизм инъекции зависимостей, поэтому мы можем строить зависимости между компонентами сервиса, проставляя соответствующие аннотации.

Для того чтобы считывать данные из очереди RabbitMQ, необходимо выделить логику, связанную с обработкой событий очереди, в отдельный класс - `ChunkConsumerService`. За инициализацию подключения к очереди отвечает `ChunkQueueConfiguration`, который содержит создание фабрики экземпляров `Connection` и `Channel`. В классе `ChunkConsumerService` реализован метод `init`, помеченный аннотацией `@PostConstruct`, который выполняется после инициализации объекта. В этом методе осуществляется пассивная декларация очередей (`uploadQueue` и `replyQueue`), так как за их создание отвечает головной сервис API-Шлюз. После этого инициализируется обработчик сообщений через метод `basicConsume` класса, который принимает два обработчика сообщений — `handleDelivery` и `handleCancelDelivery`. Метод `handleDelivery` вызывается при получении нового сообщения из очереди. Он отвечает за извлечение, валидацию и приведению к нужному типу заголовков сообщения, таких как идентификаторы

сессии (sessionId), корреляции (correlationId), а также информацию о позиции и размере видеосегмента (startByte, size). В случае успешной проверки сообщение подтверждается (basicAck), идентификатор корреляции добавляется в список ожидающих обработки сегментов (hangingChunkCorrelationIds), создается объект VideoChunk и генерируется событие ChunkReceivedEvent для дальнейшей обработки. В случае некорректных заголовков отправляется отказ (basicNack) без повторной доставки. Этот класс также является и слушателем событий RejectChunkEvent и AcceptChunkEvent в соответствующих методах acceptChunk и rejectChunk для возвращения обработки результата сообщения в очередь и логирования. Коммуникация между компонентами внутри сервиса реализована именно таким образом для того, чтобы ослабить связь между ними. Это позволяет в будущем беспрепятственно перейти на другую технологию очереди сообщений.

Метод reply проверяет, ожидает ли сегмент подтверждения по correlationId, и формирует сообщение с соответствующими заголовками статуса и ошибки. Сообщение публикуется обратно в очередь ответов через basicPublish, после чего correlationId удаляется из списка ожидающих сегментов.

3.4.2. Разработка координатора потоков

Для того, чтобы напрямую не коммуницировать с сервисом-транскодировщиком и с кэшем маршрутизации сессий загрузки к URL его реплик при обработке сегментов, описанную логику необходимо вынести в отдельный класс - FFlowCoordinatorService. Для непосредственного взаимодействия с кэшем также необходимо создать отдельный сервис - FFlowCacheService. Так как система не должна зависеть напрямую от каких-то технологий - работа с ними должна быть инкапсулирована. Он будет содержать следующие методы:

- addFFlowUrl - сохраняет URL конкретной реплики транскодировщика (fflowUrl) в Redis-кэше по идентификатору сессии загрузки (uploadSessionId);
- getFFlowUrl - получает URL реплики транскодировщика из Redis по идентификатору сессии загрузки;
- deleteFFlowUrl - удаляет URL реплики транскодировщика из кэша по идентификатору сессии загрузки.

Вернёмся к основному сервису - FFlowCoordinatorService. Класс инyectирует в свой конструктор ApiService и FFlowCacheService и ApiService. ApiService - это

утилитарный класс, который содержит методы для отправки запросов сервису-транскодировщику. Внутри себя он использует HTTP клиент WebClient, который поставляется фреймворком. Этот класс также использует следующие определённые в проекте DTO: FFlowBaseResponse, FFlowCreateResponse, FFlowFinishParams. Класс FFlowCoordinatorService содержит следующие методы:

- **ensureFlow**: проверяет наличие URL потока транскодировщика в Redis-кэше по идентификатору сессии загрузки (uploadSessionId). Если URL отсутствует, метод обращается к внешнему API для создания потока и сохраняет полученный URL в кэше;
- **pushToFlow** - отправляет массив байт с видеоданными в поток транскодировщика. Метод извлекает URL потока из кэша и осуществляет отправку данных через внешний API;
- **deleteFlow** - удаляет поток транскодировщика, отправляя соответствующий запрос на внешний API, после чего удаляет URL потока из кэша;
- **finishFlow** - завершает поток транскодировщика, передавая необходимые параметры через внешний API и удаляя информацию о потоке из кэша.

Таким образом, FFlowCoordinatorService координирует работу всех реплик системы, обеспечивая эффективное и согласованное управление потоками транскодировщика и маршрутизацию запросов. UML-диаграмма классов, описывающих FFlowCoordinatorService и взаимодействие с другими компонентами представлена на рисунке 3.10.

Рис.3.10. UML-диаграмма классов, описывающая FFlowCoordinatorService и взаимодействие с другими компонентами

3.4.3. Разработка планировщика очистки сессий

Ранее был описан процесс создания видео, однако не был описан процесс удаления видео. Этой бизнес-логикой занимается планировщик в сервисе-загрузчике, который удаляет просроченные видео - CleanUploadResourcesSchedulerService. Время, которое считается допустимым для существования сессии регулируется с помощью переменной окружения. Этот класс помечен аннотацией ConditionalOnProperty, которая ссылается на переменную окружения, определяющую необходимость включения планировщика. Время жизни видео (TTL) и время между активными сессиями планировщика задаются также с помощью переменных окружения. Класс содержит единственный метод scheduleClean, который отправляет собы-

тие `CleanUploadResourcesEvent` с временем жизни видео (`videoTtl`). Этот метод помечен аннотацией `Scheduled`, что определяет его выполнение в отдельном потоке. Это означает, что в обработчике этого события необходимо предусмотреть конкурентные запись и чтение данных, координатора потоков.

3.4.4. Разработка менеджера загрузки

Ранее мы рассматривали события, которые публикуются различными компонентами сервиса. Однако обеспечить их обработку, а также безопасное конкурентное взаимодействие с другими компонентами несколькими потоками одновременно, должен отдельный класс - `UploadManager`. Для своей работы использует несколько внешних компонентов, инъецируемых в него через конструктор: `UploadSessionRepository`, `FlowRepository`, `VideoRepository` - классы для доступа к сущностям базы данных, которые создаются с помощью фреймворка `Hibernate` в составе `Spring Boot`, а также `FFlowCoordinatorService` и `ApplicationEventPublisher` - для публикации ответных событий. Класс содержит поле `lock` типа `ReentrantLock`, которое обеспечивает потокобезопасность, гарантируя, что только один поток может одновременно изменять общее состояние и ресурсы при обработке загружаемых сегментов. `UploadManager` содержит два метода обработки событий:

- `handleVideoChunk`: активируется каждый раз, когда приходит событие о получении нового фрагмента видео (`ChunkReceivedEvent`). Сначала метод логирует информацию об обрабатываемом сегменте, затем блокирует доступ к общей логике с помощью поля `lock`, чтобы обеспечить потокобезопасность операций. После блокировки происходит проверка существования сессии загрузки (`UploadSession`) по полученному идентификатору. Если сессия не найдена, событие с отклонением сегмента отправляется обратно. Далее проверяется корректность позиции полученного сегмента: если он не соответствует ожидаемым байтам или превышает общий размер файла, загрузка также отклоняется. После успешной проверки данные сохраняются, а информация о загруженных байтах обновляется. При завершении загрузки потока его статус меняется на `DISTRIBUTED`, аналогично обновляется статус видео, если все потоки завершены. После этого сессия удаляется, и отправляется событие подтверждения сегмента.
- `handleCleanEvent`: активируется по событию `CleanUploadResourcesEvent` и выполняет очистку устаревших видео. В начале метод блокирует доступ

с помощью поля lock для потокобезопасности. Затем вычисляет момент времени (expiredThreshold), после которого видео считается устаревшим. Далее запрашивает из VideoRepository все видео, созданные до этой даты и имеющие статус CREATED или UPLOADING. Для всех найденных видео извлекаются связанные идентификаторы сессий загрузок. Затем метод удаляет соответствующие ресурсы через FFlowCoordinatorService и очищает записи о видео и сессиях загрузок из базы данных. UML-диаграмма классов сервиса представлена на рисунке 3.11.

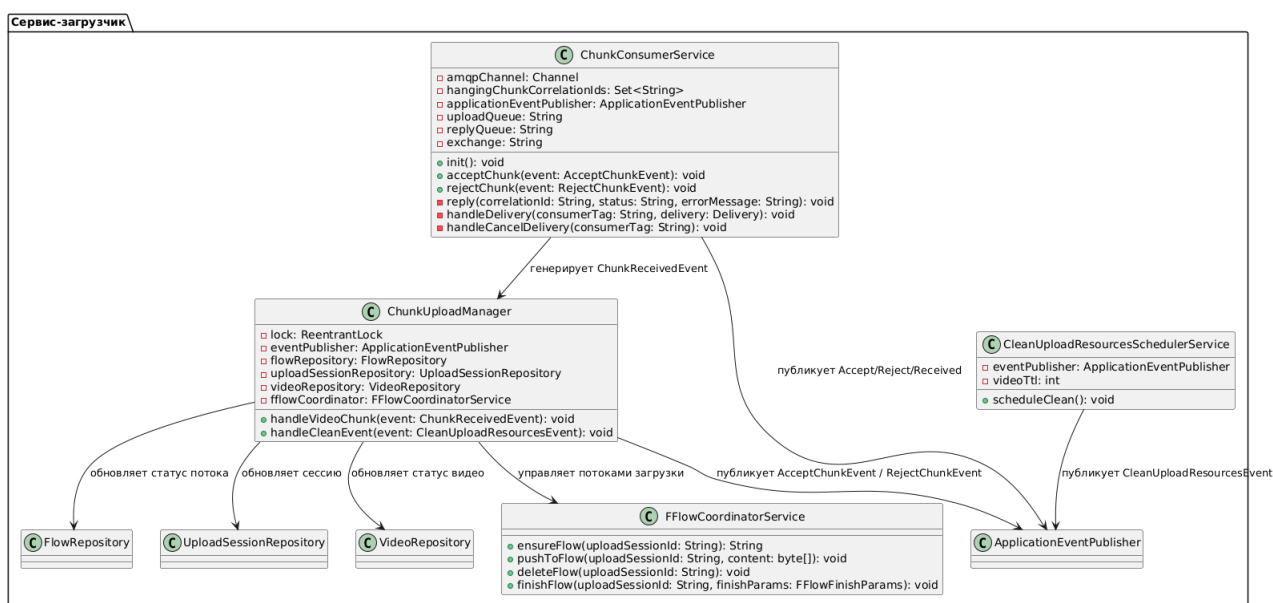


Рис.3.11. UML-диаграмма сервиса-загрузчика

3.5. Разработка видеоплеера

Видеоплеер необходим для воспроизведения адаптивного потокового видеоконтента, представленного в нескольких синхронизированных потоках. Основная задача видеоплеера - обеспечить пользователям возможность удобного переключения между потоками и предоставить качественный просмотр видеоматериалов без задержек и ошибок вне зависимости от окружения. Для этого он и должен поддерживать работу с протоколами DASH и HLS. Как было отмечено ранее, клиентское приложение разделяется на три составляющих: компоненты интерфейса, хранилище данных и менеджеры событий. Рассмотрим их в отдельности.

3.5.1. Хранилище

Redux-хранилище является единственным источником истины для всех компонентов приложения. Это означает, что все необходимые данные компоненты интерфейса берут из хранилища с помощью хука `useSelector`. Поэтому все данные нельзя хранить в общей линейной структуре, они должны быть разделены на зоны ответственности. Видеоплеер содержит следующие основные состояния:

- `player` – отвечает за воспроизведение и содержит следующие подсостояния:
 - `status` – текущий статус проигрывания (`PAUSED`, `PLAYING`, `FINISHED`);
 - `currentTime` и `duration` – текущее положение и длительность видео;
 - `isFullScreenOpened` – полноэкранный режим;
 - `volume` – уровень громкости;
 - `isMuted` – включён ли режим "без звука".

Эти состояния распространяются на все проигрываемые потоки.

- `video` – управляет метаданными видео и потоками:
 - `id`, `title`, `description`, `uploadedAt` – общая информация о видео;
 - `flowById`, `flowIds`, `mainFlowId` – список потоков и идентификатор текущего основного потока.
- `config` – содержит настройки интерфейса, которые определяются с помощью переменных окружения:
 - `isControlsHideable` – может ли интерфейс управления автоматически скрываться.
- `error` – отслеживает ошибки:
 - `type` – тип возникшей ошибки или `null`, если ошибка отсутствует.

Хранилище предоставляет набор действий (`action`) для управления изменениями состояния. Эти события будут использоваться менеджером событий далее. Например, экшен `setTime` используется для синхронного обновления времени и длительности видео, а `setStatus` – для управления статусом воспроизведения.

3.5.2. Менеджер событий

Для обработки событий используется архитектурный паттерн саг, реализуемый `Redux-Saga`. Это означает, что у саг (обработчиков событий) есть древовидная структура. Корневая сага делегирует обработку дочерним. Саги используют класс `VideoManager`, который представляет динамический словарь, способный оповещать

интерфейс об изменениях в его содержании. В качестве значений он содержит экземпляры браузерных плееров потоков. Рассмотрим каждую сагу в порядке очереди от корня к дочерним узлам ниже:

- `fillContent` – при инициализации получает ID видео из адресной строки, запрашивает данные о видео с сервера и проверяет статус. Если видео раздаётся (DISTRIBUTED), то заполняет хранилище метаданными (id, title, description, uploadedAt), списком потоков (flows) и устанавливает основной поток (mainFlowId). Если статус видео иной или потоки отсутствуют, выставляется соответствующая ошибка в хранилище;
- `handleUpdateFlows` – отслеживает событие `updateFlows` и создаёт экземпляры видеоплееров с помощью поддерживаемой в окружении технологии (DASH или HLS) для новых потоков, добавляя их в менеджер потоков;
- `playSaga` – запускает воспроизведение всех видеопотоков при срабатывании события `play` и устанавливает статус в `PLAYING`;
- `pauseSaga` – останавливает воспроизведение всех видеопотоков при срабатывании события `pause` и устанавливает статус в `PAUSED`;
- `reloadSaga` – перематывает видеопотоки на начало и запускает воспроизведение при экшене `reload`;
- `seekSaga` – перематывает видеопотоки на указанный временной промежуток при действии `seek`;
- `togglePlaySaga` – переключает воспроизведение в зависимости от текущего состояния (`PLAYING`, `PAUSED`, `FINISHED`) при действии `togglePlay`;
- `toggleSoundSaga` – включает или отключает звук при действии `toggleSound`;
- `updateVolumeSaga` – применяет новую громкость к главному потоку при действии `setVolume`;
- `updateIsMutedSaga` – устанавливает заглушение звука основного потока при действии `setIsMuted`;
- `uploadMainFlowWatcher` – отслеживает изменения основного потока (`setMainFlowId`), обновляет обработчики событий и состояние звука, отписываясь от старых и подписываясь на новые события потока.

3.5.3. Компоненты интерфейса

Компоненты интерфейса содержат древовидную структуру с переиспользуемыми узлами для того, чтобы избежать повторения кода, а также выстраивать

большое масштабируемое DOM дерево браузерной страницы. Рассмотрим каждый из компонентов в порядке очереди от корня к узлам ниже:

- **App:** корневой компонент, в котором собирается вся структура плеера. Он получает данные из хранилища (потoki, статус плеера, настройки) и отображает основной и второстепенные видеопотоки с помощью компонентов Slot. Компонент следит за движениями мыши, чтобы скрывать/показывать панель управления: используется состояние `lastWasTouched`, которое обновляется при событиях `onClick` и `onMouseMove` на контейнере `main`. Также реализована подписка на событие `fullscreenchange` через `document.addEventListener`, чтобы отслеживать вход и выход из полноэкранного режима. Обработывается завершение видео (`PlayerStatus.FINISHED`) и обновление текущего времени (`setTime`);
- **Slot:** отвечает за отображение одного видеопотока. Получает `HTMLVideoElement` из `video-manager` по `flowId` и встраивает его в DOM. При размонтировании удаляет видеоэлемент. Реализует переключение потока по клику с помощью передачи событий вверх по дереву;
- **Controls:** панель управления, отображаемая в нижней части плеера. В зависимости от статуса воспроизведения отображает разные состояния (`play`, `pause`, `reload`) и вызывает соответствующие экшны (`play`, `pause`, `reload`). Также содержит компоненты управления временем (`TimeRange`, `TimeIndicator`) и громкостью (`VolumeControl`). Обработывает нажатие кнопки полноэкранного режима;
- **TimeRange:** временная шкала. Рассчитывает прогресс воспроизведения и отрисовывает его с помощью `Range`. Обработывает движение мыши, клик и перемещение по шкале. Для этого подписывается на события `mousemove` и `mouseup` глобально через `window.addEventListener`, что позволяет точно отслеживать позицию мыши даже за пределами компонента. При отпуске мыши производится вызов экшена `seek` на рассчитанное время;
- **TimeIndicator** – отображает текущее время воспроизведения и общее время видео. Получает значения из хранилища (`currentTime`, `duration`) и преобразует их в форматированную строку, понятную пользователю;
- **VolumeControl:** отображает иконку звука в зависимости от уровня громкости или статуса `isMuted`. Использует `Range` для изменения громкости. Обработывает клики для включения/выключения звука и изменение гром-

кости с помощью мыши. Использует события mousedown, mousemove, mouseup, а также mouseenter и mouseleave для управления состоянием регулировки звука. Вызывает экшен setVolume или toggleSound;

- Range: универсальный компонент шкалы прогресса. Используется в TimeRange и VolumeControl. Обрабатывает события мыши (enter, leave, down, move, up), рассчитывает прогресс по положению курсора и передаёт результат в родительский компонент через колбэки. Внутри компонента используется useLayoutEffect для подписки на глобальные события mousemove и mouseup с последующим удалением подписки при размонтировании.

UML-диаграмма компонентов интерфейса видеоплеера представлена на рисунке 3.12.

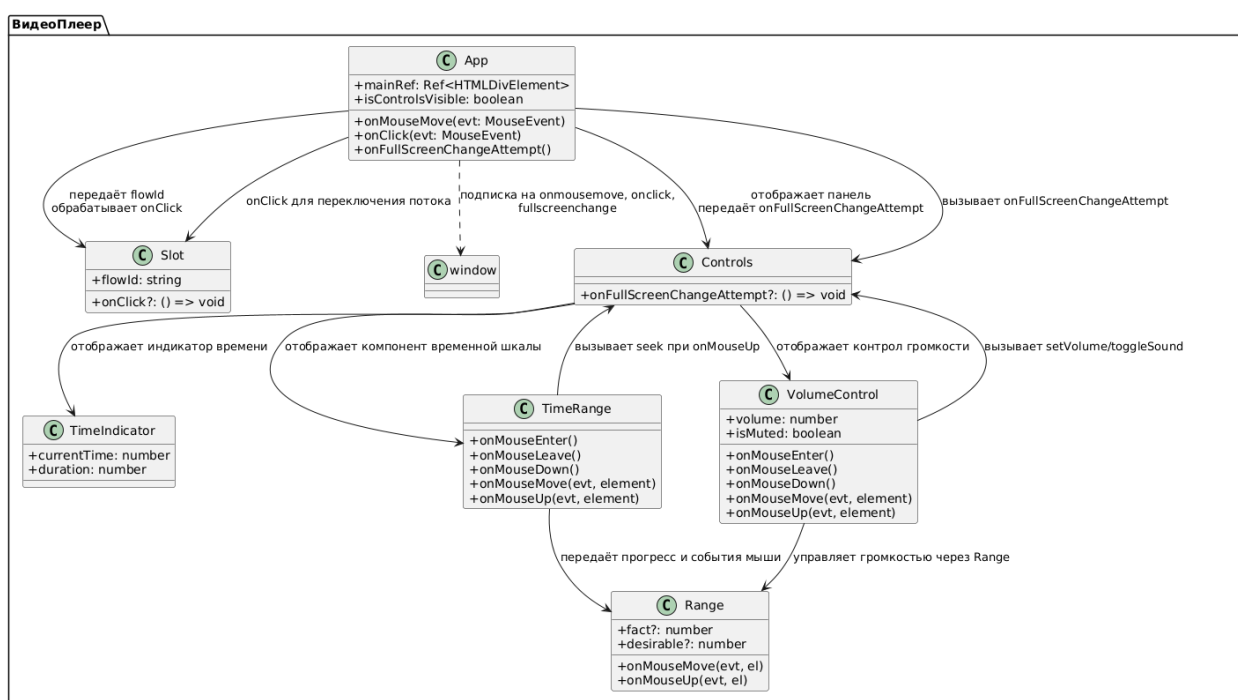


Рис.3.12. UML-диаграмма компонентов интерфейса видеоплеера

Интерфейс плеера представлен на рисунке 3.13.

3.6. Разработка редактора синхронизации

Редактор синхронизации предназначен для локальной работы в браузере пользователя с потоками видео. Он должен содержать функциональность создания кандидатов на загрузку потоков видео, их удаления, изменения порядка, непосредственно загрузку, управления метаданными видео. Это клиентское приложение,

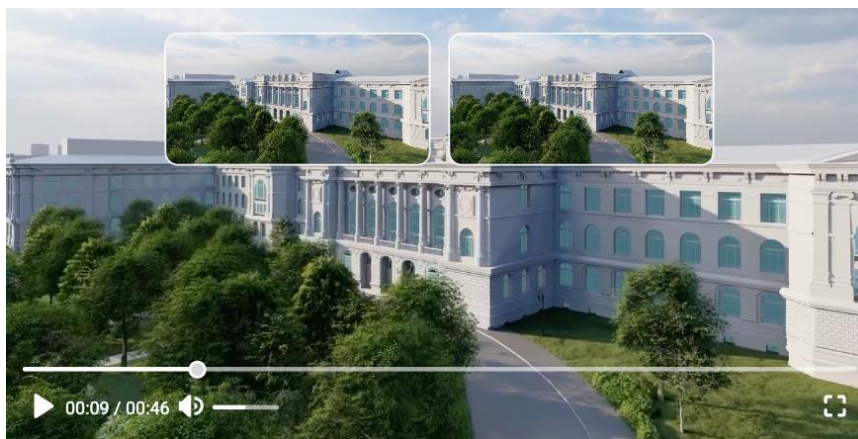


Рис.3.13. Интерфейс видеоплеера

как и видеоплеер, содержит большое количество событий и состояний, поэтому разделяется таким же образом на три составляющих: хранилище, менеджеры событий и компоненты интерфейса. Рассмотрим их ниже.

3.6.1. Хранилище

Хранилище представляет собой единый источник истины для всего приложения и структурировано по нескольким основным областям состояния, которые представлены ниже:

- **auth** — отвечает за аутентификацию пользователя и включает:
 - *Состояния:*
 - accessToken — токен доступа пользователя;
 - refreshToken — токен обновления пользователя.
 - *События:*
 - login — осуществление аутентификации;
 - logout — осуществление деаутентификации;
 - register — регистрация пользователя;
 - updateTokens — обновление токенов;
 - reset — удаление токенов из хранилища.
- **stage** — текущее состояние экрана приложения:
 - SYNCHRONIZING — форма добавления потоков;
 - LOADING — ожидание загрузки;
 - UPLOADING — активная загрузка файлов;
 - DISTRIBUTION — вывод итоговой ссылки.
- **flowCandidates** — список временных идентификаторов потоков:
 - addCandidate — добавление кандидата;

- deleteCandidateById — удаление кандидата по идентификатору;
 - addCandidateVideoAction — добавление кандидата с активным видео;
 - deleteCandidateByIdAction — удаление кандидата с активным видео.
- **upload** — информация о загружаемых потоках:
 - *Состояния*:
 - flowIds — список идентификаторов потоков;
 - flowById — словарь с метаданными и прогрессом;
 - commonStatus — общее состояние процесса загрузки.
 - *События*:
 - updateFlows — обновление текущих потоков;
 - updateCommonStatus — обновление статуса загрузки.
- **distribution** — результат успешной загрузки:
 - playerId — URL плеера, сгенерированный после дистрибуции.

3.6.2. Менеджер событий

Для обработки асинхронных действий, как и в видеоплеере, используется архитектурный паттерн саг. И для того, чтобы контролировать жизненный цикл кандидатов потоков видео, также будет реализован класс VideoManager вместе с react-хуком useVideoManager, который будет обновлять состояние приложения при изменении потоков видео.

Рассмотрим функциональность каждой из саг ниже:

- loginSaga — выполняет POST-запрос на /auth/login, получает токены и сохраняет их в localStorage. При успешном запросе обновляет состояние auth;
- logoutSaga — удаляет токены из localStorage и сбрасывает состояние auth;
- registerSaga — выполняет POST-запрос на регистрацию пользователя;
- addCandidateVideoSaga — добавляет Blob-видео в VideoManager по candidateId после выбора пользователем;
- deleteCandidateByIdSaga — удаляет поток из хранилища и удаляет соответствующее видео из VideoManager по идентификатору;
- startUploadSaga — главная сага загрузки, которая реализует следующую логику:

- Переключает интерфейс в режим ожидания (LOADING);
 - Получает список кандидатов и соответствующие видеофайлы из VideoManager;
 - Отправляет метаданные на сервер (заголовок, описание, размеры файлов);
 - Получает список потоков и обновляет хранилище;
 - Копирует видеофайлы из кандидатов в основные потоки в VideoManager;
 - Загружает видео по частям (вызов uploadFlowSaga);
 - Параллельно запускает pollVideoState, отслеживающую статус загрузки;
 - После окончания загрузки и распределения устанавливает URL плеера и переключает интерфейс на DISTRIBUTION. При ошибке возвращает SYNCHRONIZING.
- uploadFlowSaga — вспомогательная сага, разбивающая видео на чанки и отправляющая их на сервер;
 - pollVideoState — опрашивает состояние видео на сервере с интервалом STATE_POLLING_DELAY. При получении статуса BLOCKED или DISTRIBUTED завершает выполнение.

Общая UML-диаграмма компонентов хранилища и саг приложения представлена на рисунке 3.14.

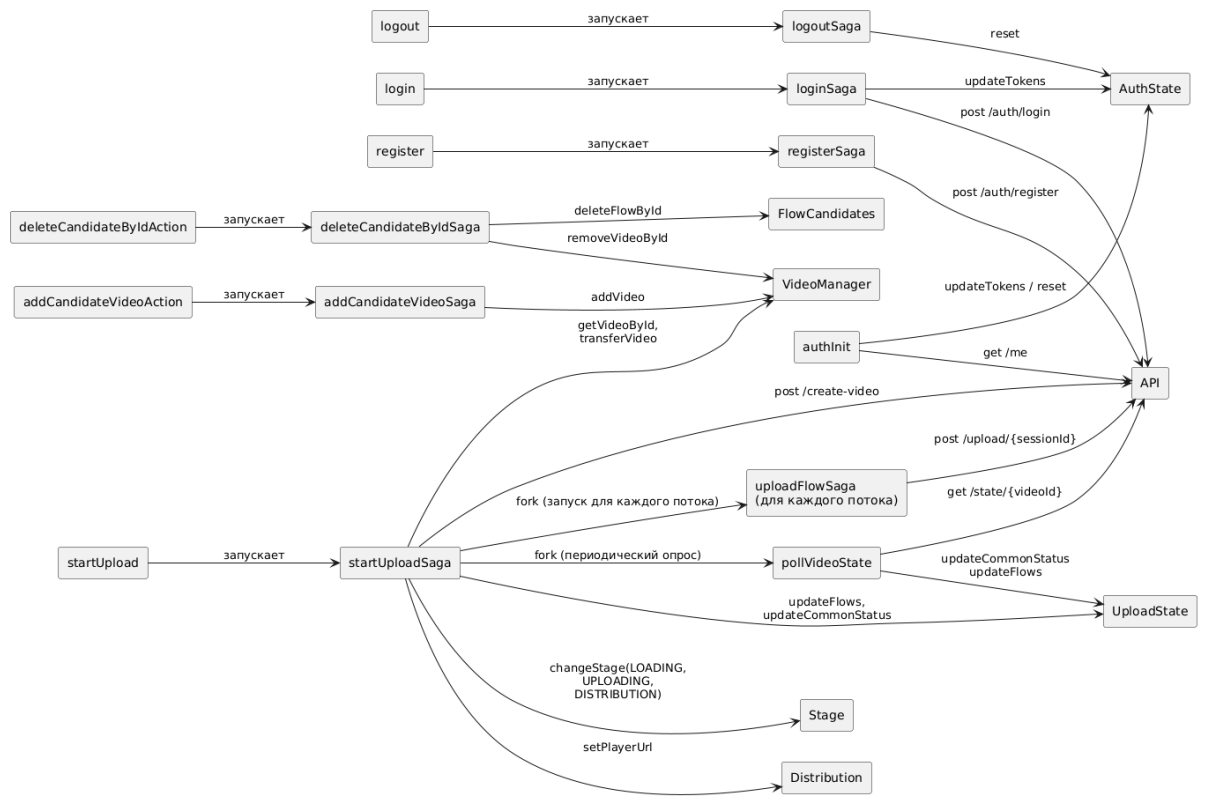


Рис.3.14. UML-диаграмма компонентов хранилища и саг редактора синхронизации

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

ГЛАВА 4. ПРОГРАММНЫЙ КОД API-ПЛЮЗА

Листинг 4.1

```

                                config.ts

import { registerAs } from '@nestjs/config';

export const AmqpConfig = registerAs('amqp', () => ({
5   connectionString: process.env.AMQP_CONNECTION_STRING,
   chunkUploadExchange: process.env.
       CHUNK_UPLOAD_AMQP_EXCHANGE,
   chunkUploadQueue: process.env.CHUNK_UPLOAD_AMQP_QUEUE,
   chunkReplyQueue: process.env.CHUNK_REPLY_AMQP_QUEUE,
}));

export const AuthConfig = registerAs('auth', () => ({
10   passwordHashSecret: process.env.PASSWORD_HASH_SECRET,
   jwtSecret: process.env.JWT_SECRET,
   refreshTokenLifetime: process.env.REFRESH_TOKEN_LIFETIME
   ,
15   accessTokenLifetime: process.env.ACCESS_TOKEN_LIFETIME,
}));

export const HolderConfig = registerAs('holder', () => ({
20   'baseUrl': process.env.HOLDER_BASE_URL,
   'manifestName': process.env.manifestName,
}));

export const RedisConfig = registerAs('redis', () => ({
25   connectionString: process.env.REDIS_CONNECTION_STRING,
   videoCacheDB: parseInt(process.env.VIDEO_CACHE_DB, 10)
}));

```

Листинг 4.2

```

                                auth.controller.ts

import { Body, Controller, HttpException, Post } from '
    @nestjs/common';

@Controller('auth')
5 export class AuthController {
   constructor(private readonly auth: AuthService) {
   }

   @Post('register')

```

```

10  async register(@Body() { email, password }: Gateway.
    Request.Register): Promise<Gateway.Response.Register>
    {
    try {
        await this.auth.register(email, password);
        return {
            status: 'success',
15         message: 'Successfully registered user',
        };
    } catch (err) {
        throw new HttpException({
            status: 'error',
20         message: 'User has already registered',
        }, 400);
    }
    }

25  @Post('login')
    async login(@Body() { email, password }: Gateway.Request
        .Login): Promise<Gateway.Response.Login> {
    try {
        const tokens = await this.auth.login(email, password
        );
        return {
30         status: 'success',
        message: 'Successfully logged in',
        payload: tokens,
        };
    } catch (err) {
35         throw new HttpException({
            status: 'error',
            message: 'Credentials are invalid',
        }, 400);
    }
40  }

    @Post('refresh')
    async refresh(@Body() { refreshToken }: Gateway.Request.
        Refresh) {
    try {
45         const tokens = await this.auth.refreshToken(
            refreshToken);
        return {
            status: 'success',
            message: 'Successfully refreshed',

```

```

        payload: tokens,
    };
    } catch (err) {
        throw new HttpException({
            status: 'error',
            message: 'Token is invalid',
        }, 400);
    }
}
}
}

```

Листинг 4.3

auth.service.ts

```

import { Repository } from 'typeorm';
import { JwtService } from '@nestjs/jwt';
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { ConfigService } from '@nestjs/config';
import * as crypto from 'crypto';

@Injectable()
export class AuthService {
    private readonly passwordHashSecret: string;
    private readonly refreshTokenLifetime: string;
    private readonly accessTokenLifetime: string;

    constructor(configService: ConfigService,
        private readonly jwt: JwtService,
        @InjectRepository(User) private readonly users:
            Repository<User>) {
        this.passwordHashSecret = configService.get('auth.
            passwordHashSecret', 'password-hash-secret');
        this.refreshTokenLifetime = configService.get('auth.
            refreshTokenLifetime', '1d');
        this.accessTokenLifetime = configService.get('auth.
            accessTokenLifetime', '30m');
    }

    async register(email: string, password: string): Promise
        <User> {
        const existingUser = await this.users.findOneBy({
            email,
        });
        if (existingUser) {

```

```

        throw new Error('User with email ${email} already
            exists');
    }
30    const newUser = this.users.create({
        email,
        passwordHash: this.hash(password),
    });
    await this.users.save(newUser);
35    return newUser;
}

async login(email: string, password: string): Promise<
    Gateway.OAuthTokens> {
    const user = await this.verifyUser(email, password);
40    return this.generateTokens(user.id);
}

async refreshToken(token: string): Promise<Gateway.
    OAuthTokens> {
    try {
45        const { sub: id } = await this.jwt.verifyAsync(token
            );
        return this.generateTokens(id);
    } catch (err) {
        throw new Error('Invalid refresh token ${token}');
    }
50 }

async verifyUserByToken(token: string) {
    const { sub: id } = await this.jwt.verifyAsync(token);
    const user = await this.users.findOneBy({
55        id,
    });
    if (!user) {
        throw new Error('Token is not verified');
    }
    return user;
60 }

private async verifyUser(email: string, password: string
    ) {
    const user = await this.users.findOneBy({
65        email,
    });

```



```

    if (!user || user.passwordHash !== this.hash(password)
        ) {
        throw new Error('User is not verified');
    }
    return user;
}

private async generateTokens(userId: string): Promise<
    Gateway.0AuthTokens> {
    const payload = { sub: userId };
    const [refreshToken, accessToken] = await Promise.all
    ([
    this.jwt.signAsync(payload, { expiresIn: this.
        refreshTokenLifetime }),
    this.jwt.signAsync(payload, { expiresIn: this.
        accessTokenLifetime }),
    ]);

    return {
        refreshToken,
        accessToken,
    };
}

private hash(str: string) {
    return crypto.createHmac('sha256', this.
        passwordHashSecret)
        .update(str)
        .digest('hex');
}
}

```

Листинг 4.4

auth.guard.ts

```

import { ExecutionContext, Injectable,
    UnauthorizedException } from '@nestjs/common';
import { AuthGuard as PassportAuthGuard } from '@nestjs/
    passport';
import { AuthService } from '../auth.service';
import { Request } from 'express';

@Injectable()
export class AuthGuard extends PassportAuthGuard('local')
{

```

```

10     constructor(private readonly auth: AuthService) {
        super();
    }

    async canActivate(context: ExecutionContext): Promise<
        boolean> {
        const request = context.switchToHttp().getRequest<
            Request>();
15         const authHeader = request.headers['authorization'];
        if (!authHeader || !authHeader.startsWith('Bearer '))
            {
                throw new UnauthorizedException('Authorization token
                    required');
            }
        const [, token] = authHeader.split(' ');
20         try {
            const user = await this.auth.verifyUserByToken(token
                );
            request.user = user;
            return Boolean(user);
        } catch (err) {
25         throw new UnauthorizedException('Token is not valid
            ');
        }
    }
}

```

Листинг 4.5

app.controller.ts

```

import { Body, Controller, Param, Get, Post, Req,
    UseGuards, Headers, Logger, HttpException, HttpStatus }
    from '@nestjs/common';
import { Request } from 'express';

5  @Controller()
    export class AppController {
        private readonly logger = new Logger(AppController.name)
            ;

        constructor(private readonly app: AppService) {
10         }

        @UseGuards(AuthGuard)
        @Get('me')
    }

```

```

15  async me(@Req() request: Request): Promise<Gateway.
    Response.Me> {
    try {
        const { user } = request;
        const { id, email, createdAt } = user;
        return {
            status: 'success',
20         payload: { id, email, createdAt },
        };
    } catch (error) {
        throw new HttpException(
            {
25         status: 'error',
            message: error.message,
        },
        HttpStatus.BAD_GATEWAY,
        );
30     }
}

@UseGuards(AuthGuard)
@Post('create-video')
35  async createVideo(@Req() request: Request,
    @Body() { title, description, totalBytesList }: Gateway.
        Request.CreateVideo): Promise<Gateway.Response.
            CreateVideo> {
    try {
        const { user } = request;
        const video = await this.app.createVideo({
40         title,
            description,
            totalBytesList,
        }, user);
        return {
45         status: 'success',
            payload: video,
        };
    } catch (error) {
        throw new HttpException(
50         {
            status: 'error',
            message: error.message,
        },
        HttpStatus.BAD_GATEWAY,
55         );
    }
}

```

```

    }
}

@UseGuards(AuthGuard)
60 @Post('upload/:sessionId')
    async upload(
        @Req() request: Request,
        @Param('sessionId') sessionId: string,
        @Headers('x-start-byte') startByteHeader: string,
65 @Headers('x-size') sizeHeader: string,
        @Body() content: Buffer,
    ): Promise<Gateway.Response.Upload> {
        try {
            const startByte = Number(startByteHeader);
70 if (isNaN(startByte)) {
                throw new Error('Start byte is not number');
            }
            const size = Number(sizeHeader);
            if (isNaN(size)) {
75 throw new Error('Size is not number');
            }
            const { user } = request;
            await this.app.upload({
                sessionId,
80 startByte,
                size,
                content,
            }, user);
            return {
85 status: 'success',
                message: 'Chunk is successfully uploaded',
            };
        } catch (error) {
            const status = (() => {
90 if (error instanceof NotFoundError) {
                return HttpStatus.NOT_FOUND;
            }

            if (error instanceof NoRightsError) {
95 return HttpStatus.UNAUTHORIZED;
            }

            return HttpStatus.BAD_REQUEST;
        })();
100 throw new HttpException(

```

```

    {
      status: 'error',
      message: error.message,
    },
    status,
  );
}
}

110 @Get('state/:videoId')
    async state(@Param('videoId') videoId: string): Promise<
      Gateway.Response.State> {
    try {
      const state = await this.app.state(videoId);
      return {
115       status: 'success',
        payload: state,
      };
    } catch (error) {
      const status = (() => {
120       if (error instanceof NotFoundError) {
         return HttpStatus.NOT_FOUND;
       }

       return HttpStatus.BAD_REQUEST;
125     })();
      throw new HttpException(
        {
          status: 'error',
          message: error.message,
130        },
        status,
      );
    }
  }
135 }

```

Листинг 4.6

app.service.ts

```

5 import { Repository } from 'typeorm';
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';

@Injectable()

```

```

export class AppService {
  constructor(private readonly chunkExchange:
    ChunkExchangeService,
    @InjectRepository(User) private readonly users:
      Repository<User>,
10    @InjectRepository(Video) private readonly videos:
      Repository<Video>,
    @InjectRepository(Flow) private readonly flows:
      Repository<Flow>,
    @InjectRepository(UploadSession) private readonly
      uploadSessions: Repository<UploadSession>,
    private readonly videoConverter: VideoConverterService)
  {
  }
15

  async createVideo({ title, description, totalBytesList
    }: Gateway.VideoPayload, owner: User): Promise<
    Gateway.SmallVideo> {
    const video = this.videos.create({
      title,
      description,
20      user: owner,
    });
    await this.videos.save(video);
    const flows = totalBytesList.map(() => this.flows.
      create({
        video,
25      }));
    await this.flows.save(flows);
    const uploadSessions = totalBytesList.map((totalBytes,
      idx) => {
      const flow = flows[idx];
      const uploadSession = this.uploadSessions.create({
30        totalBytes,
        flow,
      });
      return uploadSession;
    });
    await this.uploadSessions.save(uploadSessions);
35

    flows.forEach((flow, idx) => {
      flow.uploadSession = uploadSessions[idx];
    });
    video.flows = flows;
40    return this.videoConverter.toSmallVideo(video);
  }
}

```

```

    }

    async upload(chunk: IVideoChunk, owner: User) {
45      const { sessionId } = chunk;
      const uploadSession = await this.uploadSessions.
        findOne({
          where: {
            id: sessionId,
          },
50      relations: {
        flow: {
          video: {
            user: true,
          },
55      },
        },
      });

      if (!uploadSession) {
60        throw new NotFoundError('Session ${sessionId} not
          found');
      }

      const { flow } = uploadSession;
      const { video } = flow;
      const { user } = video;
65      if (user.id !== owner.id) {
        throw new NoRightsError('Session owner mismatches');
      }

      await this.chunkExchange.push(chunk);
70    }

    async state(videoId: string): Promise<Gateway.SmallVideo
      > {
      const video = await this.videos.findOne({
        where: {
75        id: videoId,
        },
        relations: {
          flows: {
            uploadSession: true,
80        },
        },
      });
    });

```



```

85         if (!video) {
            throw new NotFoundError('Video ${videoId} is not
                found');
        }

        return this.videoConverter.toSmallVideo(video);
    }
90 }

```

Листинг 4.7

chunk-exchange.service.ts

```

import { randomUUID } from 'crypto';
import { Injectable, OnModuleInit } from '@nestjs/common';
import { ConfigService } from '@nestjs/config';
5 import { IVideoChunk, retry, Task } from '@stombie/retube-
    core';
import { connect as amqpConnect, ConfirmChannel,
    Connection, Message } from 'amqplib';

@Injectable()
export class ChunkExchangeService implements OnModuleInit
    {
10     private readonly amqpConnectionString: string;
        private readonly chunkUploadExchange: string;
        private readonly chunkUploadQueue: string;
        private readonly chunkReplyQueue: string;
        private readonly taskByCorrelationId: Record<string,
            Task>;

15     private initPromise: Promise<void> | null = null;
        private connection: Maybe<Connection>;
        private channel: Maybe<ConfirmChannel>;

20     constructor(configService: ConfigService) {
        this.amqpConnectionString = configService.get<string>
            >('amqp.connectionString', 'amqp://localhost');
        this.chunkUploadExchange = configService.get<string>('
            amqp.chunkUploadExchange', 'chunk-exchange');
        this.chunkUploadQueue = configService.get<string>('
            amqp.chunkUploadQueue', 'chunk-upload');
        this.chunkReplyQueue = configService.get<string>('amqp
            .chunkReplyQueue', 'chunk-reply');
25     this.taskByCorrelationId = {};
    }

```

```

onModuleInit() {
  if (!this.initPromise) {
30     this.initPromise = this.init();
  }

  return this.initPromise;
}

35
push({ sessionId, content, startByte, size }:
  IVideoChunk): Promise<void> {
  const chunkId = randomUUID();
  const headers = {
    'x-session-id': sessionId,
40    'x-start-byte': startByte,
    'x-size': size,
    'x-correlation-id': chunkId,
  };

45  const routingKey = `chunk.${chunkId}`;

  return new Promise((resolve, reject) => {
    this.channel.publish(
      this.chunkUploadExchange,
50      routingKey,
      content,
      { headers },
      (err) => {
        if (err) {
55            reject(err);
            return;
        }
        this.taskByCorrelationId[chunkId] = { resolve,
          reject };
        }
60      );
    });
  });
}

private async init() {
65  this.connection = await retry(() => amqpConnect(this.
    amqpConnectionString));
  this.channel = await this.connection.
    createConfirmChannel();
  await Promise.all([

```

```

    this.channel.assertExchange(this.chunkUploadExchange,
        'topic', { durable: false }),
    this.channel.assertQueue(this.chunkUploadQueue, {
        durable: false }),
70  this.channel.assertQueue(this.chunkReplyQueue, {
        durable: false }),
    ]);
    await this.channel.bindQueue(this.chunkUploadQueue,
        this.chunkUploadExchange, 'chunk.*');
    this.channel.consume(this.chunkReplyQueue, this.
        handleReply);
    }

75
private handleReply = (message: Message) => {
    const { headers } = message.properties;
    if (!headers ||
        typeof headers['x-correlation-id'] !== 'string' ||
80  typeof headers['x-status'] !== 'string' ||
        (headers['x-status'] !== 'ack' &&
        headers['x-status'] !== 'nack')) {
        this.channel.nack(message);
        return;
85  }
    const correlationId = headers['x-correlation-id'];
    if (!this.taskByCorrelationId[correlationId]) {
        this.channel.nack(message);
        return;
90  }
    this.channel.ack(message);
    const task = this.taskByCorrelationId[correlationId];
    delete this.taskByCorrelationId[correlationId];
    if (headers['x-status'] === 'ack') {
95  task.resolve();
        return;
    }
    const error = headers['x-error'];
    if (typeof error === 'string') {
100  task.reject(new Error(error));
        return;
    }
    task.reject(new Error('Chunk ${correlationId} rejected
        with no reason'));
    }
105 }

```

app.module.ts

```

import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { ConfigModule, ConfigService } from '@nestjs/
  config';
5 import { JwtModule } from '@nestjs/jwt';

const DB_ENTITIES = [User, Video, Flow, UploadSession];

@Module({
10   imports: [
     TypeOrmModule.forRoot({
       type: 'postgres',
       url: process.env.DB_CONNECTION_STRING ?? 'postgresql
         ://postgres:postgres@localhost:5432/retube',
       synchronize: true,
15       entities: DB_ENTITIES,
     }),
     TypeOrmModule.forFeature(DB_ENTITIES),
     ConfigModule.forRoot({
       isGlobal: true,
20       load: [AmqpConfig, RedisConfig, AuthConfig,
         HolderConfig],
     }),
     JwtModule.registerAsync({
       imports: [ConfigModule],
       inject: [ConfigService],
25       useFactory: (configService: ConfigService) => ({
         secret: configService.get<string>('auth.jwtSecret'),
       }),
     }),
   ],
30   controllers: [AuthController, AppController],
   providers: [
     AppService, AuthService, ChunkExchangeService,
     UploadSessionConverterService, FlowConverterService,
     VideoConverterService, UserConverterService,
   ],
35 })
export class AppModule {}

```