**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ**

**Кафедра технологий программирования**

**РЕАЛИЗАЦИЯ ПРИЛОЖЕНИЯ НА МИКРОСЕРВИСНОЙ АРХИТЕКТУРЕ С ИСПОЛЬЗОВАНИЕМ MICROSOFT AZURE**

Курсовая работа

|  |
| --- |
| Доскоча Романа Дмитриевича |
| студента 3 курса, специальность  «прикладная информатика» |
| Научный руководитель: |
| кандидат технических наук, |
| доцент И.С. Войтешенко |

|  |  |
| --- | --- |
|  |  |

Минск, 2022

РЕФЕРАТ

Курсовой проект, 33 c., 8 рис.

**Ключевые слова:** Azure, CDN, MSA, Docker, Angular, ASP.NET.

**Объект исследования –** использование возможностей сервисов Azure, контейнеризация проектов в Docker. В качестве предмета исследования выбрана разработка и исследование характеристик видеочата.

**Цели работы –** проектирование видеосервиса основанного на микросервисной архитектуре с использованием сервисов Azure, а также реализовать приложение видеочата.

**Методы исследования –** изучение документации работы сервисов Azure. Обобщение опыта работы структуры веб приложения для внедрения работы сервисов в контроллеры API будущего приложения.

**Результатами являются –** сетевой видео чат с возможностью нескольких пользователей и комнат. Также результатом является настроеное подключение к Azure Media сервисам.

**Область применения –** для любого пользователя интернета.

РЭФЕРАТ

Курсавы праект, 33 с., 8 рыс.

**Ключавыя словы**: Azure, CDN, MSA, Docker, Angular, ASP.NET.

**Аб'ект даследавання** **–** выкарыстанне магчымасцяў сэрвісаў Azure, контейнеризация праектаў у Docker. У якасці прадмета даследавання абраная распрацоўка і даследаванне характарыстык відыачат.

**Мэты працы** **–** праектаванне відэасэрвісу заснаванага на мікра сэрвіснай архітэктуры з выкарыстаннем сэрвісаў Azure, а таксама рэалізаваць прыкладанне відыачат.

**Метады даследавання** **–** вывучэнне дакументацыі працы сэрвісаў Azure. Абагульненне вопыту работы структуры вэб прыкладання для ўкаранення працы сэрвісаў у кантралёры API будучага прыкладання.

**Вынікамі з'яўляюцца** **–** сеткавы відыачат з магчымасцю некалькіх карыстальнікаў і пакояў. Так-жа вынікам з'яўляецца наладжанае падключэнне да Azure Media сэрвісаў.

**Вобласць ужывання** **–** для любога карыстальніка інтэрнэту.

ESSAY

Course project, 33 p., 8 illustrations.

**Keywords**: Azure, CDN, MSA, Docker, Angular, ASP.NET.

**Object of research –** using the capabilities of Azure services, containerization of projects in Docker. The subject of the study is the development and study of the characteristics of video chat.

**Purpose –** designing a video service based on a micro service architecture using Azure services, as well as implementing a video chat application.

**Methods of research** **–** studying the documentation of the work of Azure services. Generalization of the experience of the web application structure for the implementation of services in the API controllers of the future application.

**The results are** **–** network video chat with the possibility of multiple users and rooms. Also, the result is a configured connection to Azure Media services.

**Scope** **–** for any internet user.

**ПЕРЕЧЕНЬ УСЛОВНЫХ ОБОЗНАЧЕНИЙ**

|  |  |
| --- | --- |
| MSA | Микросервисная архитектура (Microservices Architecture), |
| Azure | облачная платформа Microsoft. Предоставляет возможность разработки, выполнения приложений и хранения данных на серверах, расположенных в распределённых дата-центрах, |
| SOA | cервисно-ориентированная архитектура (Service-Oriented Architecture), |
| Docker | ПО для автоматизации развёртывания и управления приложениями, |
| CDN | сеть доставки содержимого (Content Delivery Network), |
| AWS | облачная платформа от Амазона (Amazon Web Services), |
| Angular | открытая и свободная платформа для разработки веб-приложений, написанная на языке TypeScript. |
|  |  |

**СОДЕРЖАНИЕ**

[**ВВЕДЕНИЕ** 7](#_Toc103895572)

[**Глава 1** 9](#_Toc103895573)

[**МОНОЛИТНАЯ И МИКРОСЕРВИСНАЯ АРХИТЕКТУРА В ПРОМЫШЛЕННЫХ ПРОЕКТАХ** 9](#_Toc103895574)

[**1.1Общие сведения** 9](#_Toc103895575)

[**1.1.1 Преимущества микросервисной архитектуры** 10](#_Toc103895576)

[**1.1.2 Недостатки микросервисной архитектуры** 10](#_Toc103895577)

[**1.2 Микросервисы, монолитная архитектура и SOA** 11](#_Toc103895578)

[**1.3 Ключевые технологии и инструменты** 12](#_Toc103895579)

[**1.4 Особенности эксплуатации приложений с микросервисной архитектурой** 14](#_Toc103895580)

[**1.5 Выводы** 14](#_Toc103895581)

[**Глава 2** 16](#_Toc103895582)

[**ПРОЕКТИРОВАНИЕ АРХИТЕКТУРЫ ВИДЕОСЕРВИСА** 16](#_Toc103895583)

[**2.1 Общее представление видеосервисов** 16](#_Toc103895584)

[**2.2 Анализ дизайна облачной архитектуры микросервисов Netflix** 16](#_Toc103895585)

[**2.2.1 Архитектура** 16](#_Toc103895586)

[**2.2.2 Использование хранилища данных** 18](#_Toc103895587)

[**2.2.3 Конвейер обработки потока** 19](#_Toc103895588)

[**2.2.4 Заключение** 20](#_Toc103895589)

[**2.3 Основы Azure. PaaS, SaaS, IaaS** 20](#_Toc103895590)

[**2.3.1 Microsoft Azure как SaaS** 21](#_Toc103895591)

[**2.3.2 Microsoft Azure как PaaS** 21](#_Toc103895592)

[**2.3.3 Microsoft Azure как IaaS** 22](#_Toc103895593)

[**2.3.4 Преимущества Microsoft Azure** 23](#_Toc103895594)

[**Глава 3** 24](#_Toc103895595)

[**РАЗРАБОТКА ВИДЕОСЕРВИСА НА ОСНОВЕ МИКРОСЕРВИСНОЙ АРХИТЕКТУРЫ AZURE** 24](#_Toc103895596)

[**3.1 Архитектура сервиса** 24](#_Toc103895597)

[**3.2 Content Delivery Network** 25](#_Toc103895598)

[**3.3 Процесс реализации приложения** 28](#_Toc103895599)

[**3.3.2 Подключение к Azure Services (Приложение А)** 29](#_Toc103895600)

[**3.3.3 Создание LiveEvent (Приложение Б)** 29](#_Toc103895601)

[**3.3.3 Создание живого вывода (Приложение В)** 29](#_Toc103895602)

[**3.3.4 API Controller & Startup** 30](#_Toc103895603)

[**3.3.5 Angular rooms (Приложение Е)** 30](#_Toc103895604)

[**ЗАКЛЮЧЕНИЕ** 32](#_Toc103895605)

[**СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ** 33](#_Toc103895606)

[**ПРИЛОЖЕНИЯ** 34](#_Toc103895607)

[*Приложение А* 34](#_Toc103895608)

[*Приложение Б* 35](#_Toc103895609)

[*Приложение В* 36](#_Toc103895610)

[*Приложение Г* 37](#_Toc103895611)

[*Приложение Д* 38](#_Toc103895612)

[*Приложение Е* 39](#_Toc103895613)

# **ВВЕДЕНИЕ**

Стремительный рост технологий в последнее время приводит к тому, что то, что еще недавно было фантастикой, сейчас внедряется в обыденную жизнь и становится привычным инструментом работы. В качестве примера можно привести технологии, связанные с экспертными системами, базами знаний, интерфейсами на основе управления голосом и жестами, нейронными сетями. Возрастающие требования к программному обеспечению приводят к необходимости пересмотра и развития архитектур и подходов к проектированию и разработке информационных систем.

Использование микросервисной архитектуры при разработке приложений является сегодня популярным подходом к созданию современных приложений. Она позволяет органично проводить горизонтальное масштабирование приложений и возлагает на команду разработчиков полную ответственность за разработанный сервис. Требуется уделять особое внимание вопросам использования различных архитектур разработки, необходимо объяснять достоинства и недостатки различных архитектур, отрабатывать навыки разработки микросервисов с использованием различных подходов.

К подходу микросервисной архитектуры хорошо подходит концепция видеосервиса, где пользователи смогут смотреть всевозможный видео контент, а также загружать свой.

Индустрия средств массовой информации и развлечений всегда была в арсенале технологических инноваций. Существует негласный закон: либо измениться, либо умереть. Это то, над чем до сих пор размышляют такие действующие игроки, как Disney, Fox, и Time Warner. Это то, что охватывает такие видеосервисы, как Netflix, Hulu и YouTube.

В этой работе я опишу структуры проектов, построенных на идее разделения задач на микросервисы, возможные трудности и упущения, а также большие возможности нового подхода.

В первой главе будет рассказано про основы и особенности разработки программного обеспечения с помощью MSA. Их плюсы и минусы. А также сравнение с монолитным подходом проектирования и основные инструменты работы с ним.

Во второй главе будет разобран один из самых популярных стриминговых сервисов Netflix. Описаны базовые принципы Azure, его область применения и актуальность. Описание архитектуры своего видеосервиса. Итогом будет

В третьей главе будет описана проектная схема микросервисной архитектуры для видеосервиса, разобраны ее составляющие. А Также предоставлены результаты разработки видеочата, описан принцип работы сервисов Azure и метод внедрения его в ASP.NET и Angular проект.

# **Глава 1**

# **МОНОЛИТНАЯ И МИКРОСЕРВИСНАЯ АРХИТЕКТУРА В ПРОМЫШЛЕННЫХ ПРОЕКТАХ**

# **1.1Общие сведения**

Микросервисная архитектура (**MSA**) – стиль разработки ПО, заключающийся в разбиении монолита системы на отдельные компоненты, которые представляют собой независимые сервисы. Каждый отдельный сервис призван выполнять относительно несложные функции, работа приложения тестируется в режиме реального времени с отслеживанием технических элементов и бизнес-характеристик. Сервисы имеют чёткие физические границы, что делает их масштабируемыми, даёт возможность использовать для написания разные языки программирования. Требуемые изменения вносятся, не затрагивая всё приложение целиком.

Микросервисы завоёвывают всё большую популярность среди разработчиков, они хороши для крупных интернет-проектов, обслуживающих большое количество пользователей. Если случится сбой в одном сервисе, то остальные спокойно продолжат свою работу: это очень важно для больших и сложных систем. При нововведениях можно просто написать новый сервис, он легко впишется во всю систему, не придётся заниматься проблемами совместимости. Более того, в работу можно включать несколько независимых друг от друга команд, использующих разные языки программирования: микросервисы позволяют это сделать.

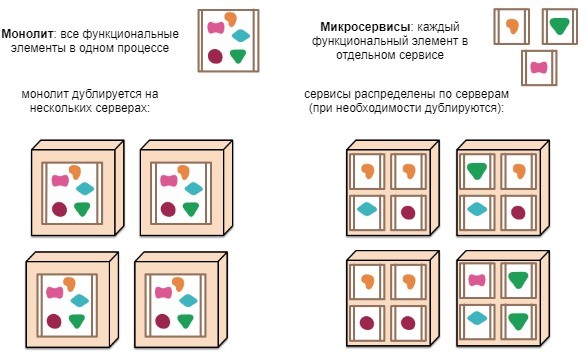


Рисунок 1.1 - Наглядное различие микросервисной и монолитной архитектуры

Коротко говоря, монолитные системы объединяют все сервисы приложения в один процесс, а потом дублируют эту систему на нескольких серверах. Микросервисная архитектура для каждого сервиса имеет отдельную упаковку, а на разных серверах они хранятся также – изолированно друг от друга, дублируясь, если есть необходимость. Что и позволяет отдельно работать с каждым функциональным элементом.

# **1.1.1 Преимущества микросервисной архитектуры**

Преимущества микросервисов оказались достаточно сильны, чтобы такие крупные корпоративные игроки как Amazon, Netflix или eBay внедрили их в свои системы.

* **Высокая отказоустойчивость:** при падении одного из сервисов, все остальные остаются в строю. Таким образом, неполадки в отдельных сервисах не помешают всему рабочему процессу.
* **Гибкость:** можно попробовать внедрить новую технологию малыми усилиями. Это будет значительно быстрее и, при неудаче, откатить изменения просто. Меняя локально один из сервисов, мы не рискуем всей системой и время, требующееся для изменений, меньше.
* **Простота:** чем меньше кода (а каждый отдельный сервис представляет собой цельную систему, поэтому не нужно разбираться в огромном количестве деталей, не касающихся данной конкретной функции), тем проще программистам разобраться, что и как работает. К тому же, на это уйдет меньше времени.
* **Лёгкость** выведения написанного кода в работу. Небольшое количество кода обеспечивает быструю развертку.
* **Масштабируемость.** Самые необходимые и нужные сервисы можно дополнить и расширить, когда появится такая необходимость. Вся система при этом остается прежней.
* **Многоязычность.** Прелесть микросервисов в том, что мы можем писать каждый сервис на отдельной языке программирования, тем самым позволяют связать разных разработчиков для написания общего проекта.

# **1.1.2 Недостатки микросервисной архитектуры**

Несмотря на популярность и большое количество плюсов микросервисной архитектуры, она обладает и минусами.

* **Сообщение между самими сервисами сложное.** Так как каждый функциональный элемент изолирован, требуется особая тщательность при построении между ними грамотной коммуникации, ведь им в любом случае придётся обмениваться запросами и ответами друг с другом. Понятно, что с увеличением количества сервисов сложность в построении их сообщения будет расти.
* Рост числа сервисов также влечет за собой и **рост числа баз данных**, с которыми эти сервисы соотносятся, так как, в отличие от монолитной архитектуры, микросервисы используют не одну общую базу данных.
* **Сложность тестирования** выражается в том, что сначала нужно отдельно разбираться с каждым сервисом, а потом тестировать корректное взаимодействие его с другими микросервисами.
* **Не для всех.** Микросервисы хуже подходят для использования внутри отдельных организаций, для них они могут оказаться неоправданно сложными в применении, в то время как для массовых интернет-сервисов или видеосервисов они подходят отлично.
* **Цена**. Микросервис стоит дорого, так как вам нужно поддерживать разное серверное пространство для разных бизнес-задач.

# **1.2 Микросервисы, монолитная архитектура и SOA**

Чтобы уточнить отличия микросервисов от других архитектур, их чаще всего сравнивают с монолитной архитектурой и сервис-ориентированной архитектурой (SOA).

Микросервисное приложение состоит из множества мелких независимых и слабо связанных между собой сервисов, в то время как в монолите все его компоненты тесно взаимосвязаны и работают как единый сервис. Помимо прочего, это значит, что, если какой-то один процесс в приложении с монолитной архитектурой становится более востребованным, приходится масштабировать всё приложение в целом. Сбой в каком-то одном процессе может поставить под угрозу всю систему. Наконец, такая сложность ограничивает возможности модернизации и затрудняет внедрение новых идей.

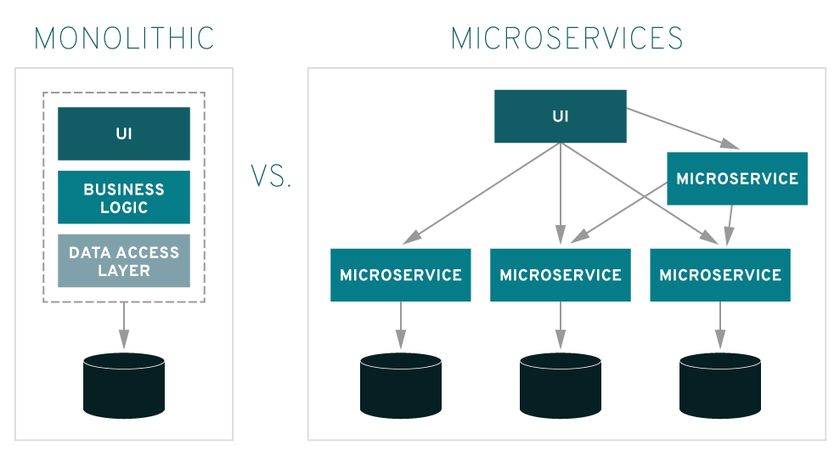


Рисунок 1.2 Общая схема MSA проекта

Монолитная архитектура отличается тесной взаимосвязью между компонентами, включая бизнес-логику и слой доступа к данным, и выступает как единый сервис. В микросервисной архитектуре клиент через общий пользовательский интерфейс получает доступ к отдельным слабо связанным между собой микросервисам.

Отличия микросервисов от SOA не столь очевидны. Можно пойти по сложному пути и перечислить множество технических деталей, но можно поступить проще и оценить уровни, на которые распространяются эти архитектуры. Если SOA — это архитектура уровня предприятия, призванная стандартизировать взаимодействие всех служб, то микросервисы относятся к какому-то одному конкретному приложению.

# **1.3 Ключевые технологии и инструменты**

Модули в микросервисной архитектуре могут быть реализованы на основе практически любого современного языка программирования или инструмента, но существует набор ключевых инструментов, ставших для неё обязательными и определяющими.

Прежде всего, речь идёт о Docker — ПО для развертывания и управления приложениями на основе контейнеризации — модели вычислений, наиболее тесно ассоциируемой с микросервисами. Поскольку индивидуальные контейнеры для приложений не обладают всеми атрибутами полноценной операционной системы, они меньше и легче по объёму, чем обычные виртуальные машины. Благодаря этому они запускаются и отключаются быстрее, и тем самым идеально подходят для небольших и лёгких микросервисов.

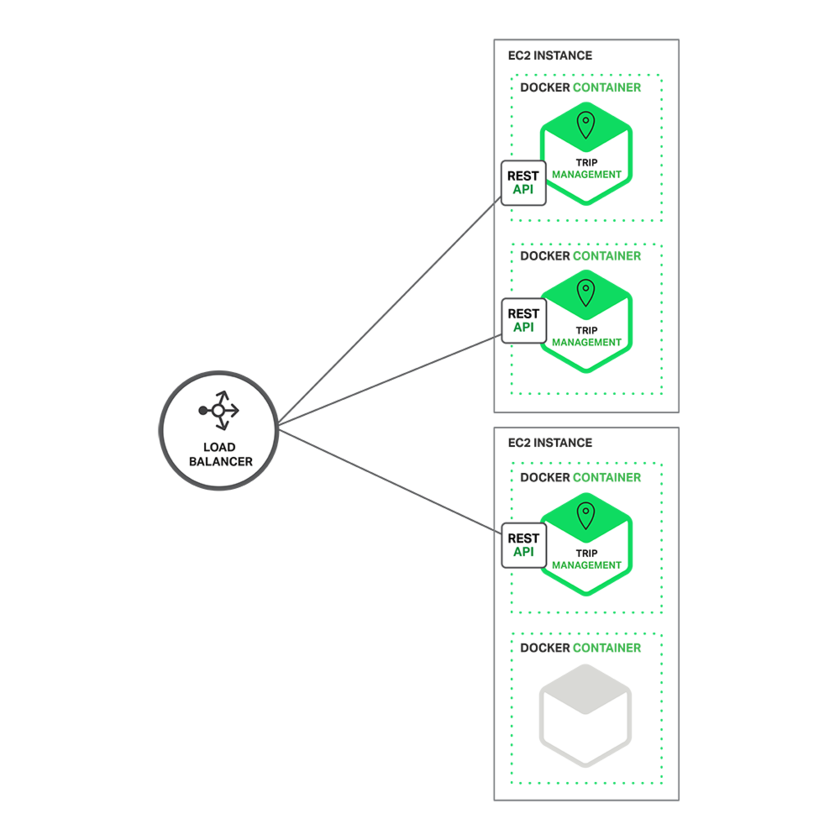


Рисунок 3 Пример организации сервиса

На схеме мы видим сервис по аренде автомобиля с водителем (Trip Management) на облачной платформе Amazon EC2. В процессе работы сервис отвечает на несколько запросов, каждый из которых обрабатывается в отдельном Docker-контейнере. Передача информации осуществляется через REST API по протоколу HTTP. Распределением запросов занимается балансировщик нагрузки (Load Balancer), такой как nginx.

С появлением множества сервисов на базе контейнеризации стали чрезвычайно востребованы средства автоматизации управления большими наборами контейнеров. Одной из самых популярных в мире технологий «оркестровки» контейнеров, то есть автоматического развёртывания, управления, масштабирования и сетевого подключения на сегодняшний день является Kubernetes.

Микросервисы часто взаимодействуют через API, особенно при первоначальном установлении связей. Несмотря на то, что клиенты и сервисы могут общаться напрямую, шлюзы API выступают полезным промежуточным элементом с ростом числа сервисов в приложении. Помимо функций реверсивного прокси и маршрутизатора, они обеспечивают дополнительный уровень безопасности для приложений.

Однако связь через API не является эффективным и практичным способом оперативного взаимодействия в реальном времени, поэтому наряду с ним в этих случаях применяются обмен сообщениями или потоками событий. С этой задачей лучше всего справляются брокеры сообщений и платформы потоков событий, такие как Apache Kafka.

Еще один способ организации микросервисной архитектуры являются бессерверные вычисления — стратегия, доводящая до логического завершения некоторые ключевые особенности облачных и микросервисных технологий. При её реализации исполнительный блок представляет собой даже не просто маленький сервис, а лишь функцию, которая часто выражается в нескольких строчках кода. Грань, отделяющая бессерверную функцию от микросервиса, довольно условна, поэтому функция обычно рассматривается как ещё более мелкий процесс, чем микросервис.

# **1.4 Особенности эксплуатации приложений с микросервисной архитектурой**

* **Осуществление мониторинга**

Со временем небольшие проблемы могут привести к серьёзным. Поэтому своевременный, а в случае микросервисов, непрерывный, мониторинг – необходим. Мониторинг микросервисной архитектуры гораздо более сложен в настройке и требует больших затрат на свою поддержку, нежели мониторинг монолитного приложения.

* **Реализация отказоустойчивости**

Как монолитная, так и микросервисная архитектура, не могут избежать отказов при работе приложения. Важным принципом при создании приложения на микросервисах является установка на то, что отказы системы рано или поздно произойдут. И с таким изначальным предположением и создаётся приложение.

* **Тестирование**

Перед выпуском в свет код должен быть тщательно протестирован. Тем не менее, всегда есть вероятность, что какая-то ошибка будет упущена. Именно для таких случаев придумана система хаос-тестирования. Ошибки и проблемы создаются автоматически и намеренно – чтобы заранее изучить их и придумать быстрое решение или способ их предотвратить.

* **Масштабирование**

Облачная природа микросервисов и виртуализация, лежащая в их основе, главным преимуществом имеют масштабирование. Каждый сервис выполняет определённую функцию, все сервисы относительно небольшие – все эти характеристики архитектуры и делают масштабирование более простым и безболезненным.

# 

# **1.5 Выводы**

Микросервисы хуже подходят для использования в качестве архитектуры для разработки корпоративных информационных систем. В таком варианте мы получим излишне сложную с точки зрения эксплуатации и сопровождении систему. А увеличение эксплуатационных затрат не окупится за счет преимуществ использования микросервисной архитектуры, поскольку они не проявятся в полной мере внутри одной, даже крупной организации.

Основная область эффективного применения микросервисов – крупные интернет-сервисы, работающие в режиме реального времени для большого количества пользователей. Как раз один из таких сервисов и будет разобран в следующей главе.

# **Глава 2**

# **ПРОЕКТИРОВАНИЕ АРХИТЕКТУРЫ ВИДЕОСЕРВИСА**

# **2.1 Общее представление видеосервисов**

Видеосервис это – огромная архитектура взаимодействия всевозможных сервисов по предоставлению видео контента пользователю. Из-за столь большого количества функционала логичнее всего строить данные сервисы на архитектуре MSA, где каждый компонент и отдельный ресурс будет представлять свою изолированную область, что позволит обеспечить гибкость и расширяемость.

Платформы потокового видео, такие как **YouTube, Netflix, Twitch**, Hulu и др., стали популярными в наши дни. Новый режим просмотра фильмов, сериалов и онлайн-шоу становится все более виртуальным. И это неудивительно, поскольку люди во всем мире становятся свидетелями популяции смартфонов и интернета. Фильмы стали возможны даже в дороге.

# **2.2 Анализ дизайна облачной архитектуры микросервисов Netflix**

Рассмотрим архитектуру на примере самого популярного стримингового сервиса - Netflix. Netflix уже много лет является одним из лучших онлайн-сервисов потокового видео на основе подписки в мире, на долю которого приходится более 15% мировой пропускной способности Интернета.

Успех компании может строиться на решении миграции ИТ-инфраструктуры из центров обработки данных в общедоступное облако и замена монолитных программ небольшими управляемыми программными компонентами архитектурой микросервисов. Netflix выбрала облако AWS для миграции своей ИТ-инфраструктуры, поскольку AWS может предложить высоконадежные базы данных, крупномасштабное облачное хранилище и несколько центров обработки данных по всему миру. За счет использования облачной инфраструктуры построена и поддерживается AWS.

# **2.2.1 Архитектура**

С точки зрения архитектуры программного обеспечения Netflix состоит из трех основных частей: клиента, сервер и сети доставки контента (CDN).

**Клиент** — это любой поддерживаемый браузер на ноутбуке или настольном компьютере или приложение Netflix на смартфонах или смарт-телевизорах. Netflix разрабатывает собственные приложения для iOS и Android, чтобы обеспечить наилучшее качество просмотра для каждого клиента и устройства. Управляя своими приложениями и другими устройствами с помощью своего SDK, Netflix может прозрачно адаптировать свои потоковые сервисы при определенных обстоятельствах, таких как медленные сети или перегруженные серверы.

Сервервключает сервисы, базы данных и хранилища, полностью работающие в облаке AWS. Серверв основном обрабатывает все, кроме потокового видео*.* Некоторые компоненты Сервер с соответствующими сервисами AWS перечислены ниже:

* Масштабируемые вычислительные экземпляры (AWS EC2)
* Масштабируемое хранилище (AWS S3)
* Масштабируемые распределенные базы данных (AWS DynamoDB, Cassandra)
* Обработка больших данных и аналитика (Hadoop, Spark, Flink, Kafka)
* Обработка и кодирование видео.

Сети доставки контента (**CDN)** — это сеть серверов под названием Open Connect Appliances (OCA), оптимизированная для хранения и потоковой передачи больших видео. Эти серверы OCA размещаются внутри сетей интернет-провайдеров и точек обмена интернет-трафиком по всему миру. OCA отвечают за потоковую передачу видеонепосредственно клиентам.

Когда подписчики нажимают кнопку «Воспроизвести» в своих приложениях или устройствах, клиент будет общаться как с серверной частью на AWS, так и с OCA на Netflix CDN для потоковой передачи видео. На следующей диаграмме показано, как работает процесс воспроизведения:



Рисунок 2.1 Архитектура воспроизведения для потокового видео

1. OCA постоянно отправляет отчеты о состоянии своей рабочей нагрузки, возможности маршрутизации и доступных видео в службу Cache Control, работающую в AWS EC2, чтобы приложения для воспроизведения могли обновлять клиентам последние исправные OCA.
2. Запрос воспроизведения отправляется с клиентского устройства в сервис Netflix Playback Apps, работающий на AWS EC2, чтобы получить URL-адреса для потоковой передачи видео.
3. Служба представления приложения должна определить, что запрос воспроизведения будет действителен для просмотра конкретного видео. Такие проверки будут проверять тарифный план подписчика, лицензирование видео в разных странах и т. д.
4. Сервис приложений воспроизведения обращается к управляющему сервису, также работающему в AWS EC2, чтобы получить список соответствующих серверов OCA для запрошенного видео. Служба управления использует IP-адрес клиента и информацию об интернет-провайдерах, чтобы определить набор подходящих OCA, которые лучше всего подходят для этого клиента.
5. Из списка 10 различных серверов OCA, возвращенных службой представления приложения, клиент проверяет качество сетевых подключений к этим OCA и выбирает самый быстрый и надежный OCA для запроса видеофайлов для потоковой передачи.
6. Выбранный сервер OCA принимает запросы от клиента и запускает потоковую передачу видео.

# **2.2.2 Использование хранилища данных**

При переходе своей инфраструктуры на облако AWS, Netflix использовала разные хранилища данных, такие как SQL, и NoSQL:

* Базы данных MySQL используются для управления названиями фильмов и транзакций / выставления счетов.
* Hadoop используется для обработки больших данных на основе пользовательских журналов.
* ElasticSearch обеспечил поиск заголовков для приложений Netflix
* Cassandra — это распределенное хранилище данных NoSQL на основе столбцов, предназначенное для обработки большого количества запросов на чтение без единой точки возврата. Чтобы оптимизировать задержку при выполнении больших запросов на запись, используется Cassandra из-за ее согласованной способности (Consistency).

# **2.2.3 Конвейер обработки потока**

Конвейер потоковой обработки данных стал основой данных Netflix для бизнес-аналитики и задач персонализированных рекомендаций. Он отвечает за создание, сбор, обработку, агрегирование и передачу всех событий микросервисов другим обработчикам данных почти в реальном времени. На рисунке ниже показаны различные части платформы.

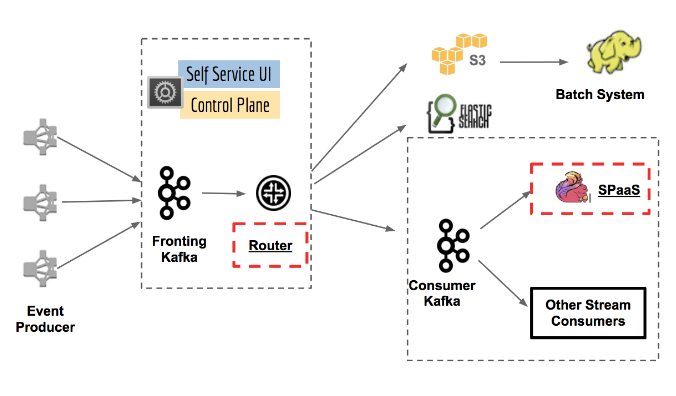


Рисунок 2.2 Платформа обработки потокового видео Keystone в Netflix

* Платформа потоковой обработки принимает триллионы событий и петабайты данных в день. Она также будет автоматически масштабироваться по мере увеличения количества подписчиков.
* Модуль **Router** обеспечивает маршрутизацию к различным приемникам данных или приложениям, в то время как Kafka отвечает за маршрутизацию сообщений, а также за буферизацию для нисходящих систем.
* **Потоковая** обработка как услуга ( **SPaaS Seismic processing as a service**) позволяет инженерам по обработке данных создавать и контролировать свои настраиваемые приложения для управляемой потоковой обработки, в то время как платформа позаботится о масштабируемости и операциях.

# **2.2.4 Заключение**

В этой главе описана основная часть облачной архитектуры стриминговых сервисов Netflix*.* Облачная архитектура Netflix*,* проверенная их производственной системой для обслуживания миллионов пользователей, работающих на тысячах виртуальных серверах, продемонстрировала высокую доступность с оптимальной задержкой, сильную масштабируемость за счет интеграции с облачными сервисами AWS и способность противостоять сбоям сети и сбоям системы в глобальном масштабе*.*

Большая часть внутренней архитектуры развивается через доступные онлайн ресурсы. Несмотря на то, что прямых ресурсов, описывающих внутреннюю реализацию микросервисов, а также инструментов и систем для мониторинга их производительности, не так много.

Архитектура Netflix **может служить эталонной реализацией того, как должна быть построена любая производственная система.**

# **2.3 Основы Azure. PaaS, SaaS, IaaS**

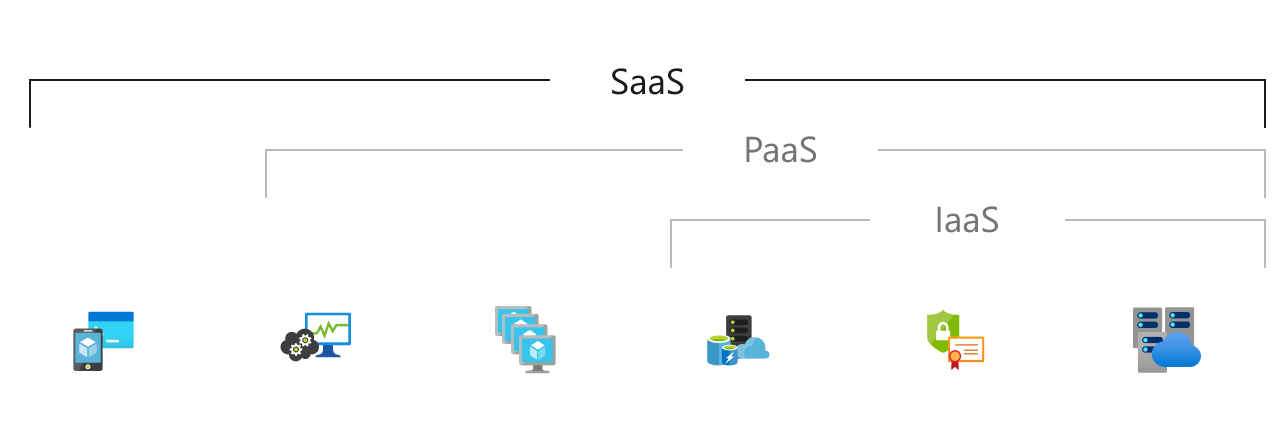


Рисунок 2.3 Область применения каждого подхода

Microsoft Azure – [облачная платформа](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D0%BB%D0%B0%D1%87%D0%BD%D1%8B%D0%B5_%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F) компании [Microsoft](https://ru.wikipedia.org/wiki/Microsoft). Предоставляет возможность разработки, выполнения приложений и хранения данных на серверах, расположенных в распределённых [дата-центрах](https://ru.wikipedia.org/wiki/%D0%94%D0%B0%D1%82%D0%B0-%D1%86%D0%B5%D0%BD%D1%82%D1%80).

Облако Azure было анонсировано в октябре 2008 года под кодовым названием «Project Red Dog». Релиз состоялся 1 февраля 2010 года под названием «Windows Azure». В 2014 году платформа была переименована в Microsoft Azure.

Azure реализует такие облачные модели платформы как сервиса (PaaS) и инфраструктуры как сервиса (IaaS). Возможно использование как сторонних, так и сервисов Microsoft в качестве модели ПО как сервиса (SaaS).

Поначалу американская корпорация Microsoft ориентировалась на службы PaaS для Azure, и они были сфокусированы на существующей базе разработчиков, которые используют технологии Microsoft. Но прошло не так много времени, и Microsoft расширила свое внимание и на сервисы Linux, и на IaaS. Американская корпорация старалась сделать Azure все более комфортной, в результате чего появилась поддержка API для различных сервисов.

В наши дни почти все сервисы Microsoft Azure имеют интерфейс взаимодействия API, построенный на основе ограничений для распределённых систем REST, с помощью чего разработчики могут использовать облачные сервисы с любой ОС, устройства и платформы.

Основополагающий принцип работы Azure состоит в [развертывании удалённой виртуальной машины (ВМ)](https://www.plus-aliance.ru/news/tekhnoblog/create-a-virtual-machine-in-azure-part-1-the-azure-portal/) под каждый тип приложения или сервиса. При этом клиент самостоятельно решает какой объём хранилища данных и вычислительных мощностей ему нужен для решения корпоративных задач, а затем подключается к платформе. Используя Azure, вы сможете без труда настроить ВМ Windows или Linux, на которой можно будет разместить любое ПО, какое только пожелаете. К примеру, на ВМ можно разместить и рабочий стол Windows или Linux в облаке, и подключаться к нему удаленно.

# **2.3.1 Microsoft Azure как SaaS**

Программное обеспечение как услуга (SaaS – Service as a Service) позволяет пользователям подключаться к облачным приложениям и использовать их через Интернет. Стандартными примерами могут быть электронная почта, ведение календаря и офисные средства (например, Microsoft Office 365).

SaaS предоставляет полноценный набор программного обеспечения, который вы оплачиваете [поставщику облачных служб](https://azure.microsoft.com/ru-ru/overview/choosing-a-cloud-service-provider/) по мере использования. Вы арендуете использование приложения для вашей организации, и ваши пользователи подключаются к нему через Интернет, обычно с помощью веб-браузера. Вся базовая инфраструктура, ПО промежуточного слоя, программное обеспечение приложений и данные приложений находятся в центре обработки данных поставщика. Поставщик служб управляет оборудованием и программным обеспечением на основе соответствующего соглашения об обслуживании и обеспечивает доступность и безопасность приложений и данных.

# **2.3.2 Microsoft Azure как PaaS**

Платформа как услуга (PaaS – Platform as a Service) – это модель предоставления облачных вычислений, при которой потребитель получает доступ к использованию информационно-технологических платформ: ОС, СУБД, связующему ПО, средствам разработки и тестирования, размещённым у облачного провайдера. PaaS чрезвычайно полезен для оптимизации рабочих процессов, например, когда несколько разработчиков работают над одним проектом.

Пожалуй, главным преимуществом PaaS будет то, что пользователю не нужно переживать о работе ОС либо обновлениях, а также об аппаратных обновлениях. Именно на плечи провайдера возложена забота о своевременном обновлении вашей ОС, функциях платформы, к примеру, базовая платформа .NET или ядро ​​базы данных SQL, а также по запросу он осуществляет обновление оборудования согласно требованиям компании. Таким образом, вы покупаете ресурсы у провайдера, оплачиваете их по мере использования и подключаетесь к ним по безопасному интернет-подключению.

# **2.3.3 Microsoft Azure как IaaS**

Модель предоставления инфраструктуры (IaaS – Infrastructure as a Service) позволяет арендовать самые различные ресурсы, как виртуальные серверы, хранилища и сетевое оборудование. Управление всей инфраструктурой производится провайдером, а пользователь управляет лишь ОС и установленными приложениями. То есть, клиент сам разворачивает в арендуемом облаке масштабируемую и отказоустойчивую IT-инфраструктуру, но при этом экономит немалые средства на покупке и обслуживании своего парка оборудования и ПО.

В основном продукция IaaS обеспечивает инфраструктуру вычислений, хранилища и сети. Как правило, эти службы предоставляются в виде виртуальной машины, содержание которых находится под полным контролем у пользователя.

Используя Azure IaaS, вы получаете все преимущества высоко доступной, масштабируемой и безопасной облачной инфраструктуры, однако при этом оплачиваете лишь те ресурсы, которые непосредственно используете для решения бизнес-задач компании. Соответственно, Azure IaaS позволит вашей компании:

* Иметь полный контроль над приложениями и инфраструктурой, а также адаптировать виртуальную машину под запросы компании и бизнес-проекты;
* Оптимизировать используемые ресурсы;
* Повысить безопасность и защищенность корпоративных данных;
* Сэкономить финансы, выполнив переход в облако и получив больше дохода от уже вложенных средств.

Помимо всего прочего, компании могут создавать и управлять собственными сервисами, используя визуальный веб-интерфейс портала Azure. С помощью портала можно самостоятельно настраивать сервисы, редактировать права доступа, отслеживать состояние ресурсов и управлять оплатой.

Платформа Azure позволит компании сосредоточиться на развертывании собственных приложений, и легко настроить масштабирование приложений в сторону увеличения или уменьшения в зависимости от бизнес-проектов.

# **2.3.4 Преимущества Microsoft Azure**

Azure предлагает открытые и гибкие платформы для компаний, позволяя им быстрее реагировать и более эффективно решать собственные бизнес-задачи. Ее преимущества заключены в том, что Microsoft Azure – это:

1. Совокупность Iaas, Paas и SaaS в одном решении;
2. Более 600 различных инструментов;
3. Возможность развёртывания IT-инфраструктуры любой сложности и в максимально короткие сроки;
4. Неограниченные, высоко масштабируемые вычислительные ресурсы;
5. Широкий выбор ОС, популярных языков и средств платформенной разработки. Для виртуальной машины доступны образы следующих ОС: Windows Server, CoreOS, Ubuntu Server, Red Hat, Clear Linux OS, Debian.

# **Глава 3**

# **РАЗРАБОТКА ВИДЕОСЕРВИСА НА ОСНОВЕ МИКРОСЕРВИСНОЙ АРХИТЕКТУРЫ AZURE**

Одно из способов реализации видеосервиса является подход потоковой передачи в реальном времени позволяет записывать видео в режиме реального времени и транслировать его потребителям, например, при потоковой передаче интервью, конференций и спортивных мероприятий в Интернете.

В этом решении видео захватывается видеокамерой и отправляется на конечную точку ввода Live Event. Live Event получает входной поток и делает его доступным для потоковой передачи через конечную точку потоковой передачи в веб-браузер или мобильное приложение. Live Event также предоставляет конечную точку мониторинга предварительного просмотра для предварительного просмотра и проверки вашего потока перед дальнейшей обработкой и доставкой. Live Event также может записывать и сохранять загруженный контент для последующей потоковой передачи (видео по запросу).

Этот способ построен на управляемых службах Azure: хранилище BLOB-объектов, CDN и медиаплеер Azure. Эти службы работают в среде высокой доступности, исправлены и поддерживаются, что позволяет нам сосредоточиться на своем решении, а не на среде, в которой они работают.

# **3.1 Архитектура сервиса**

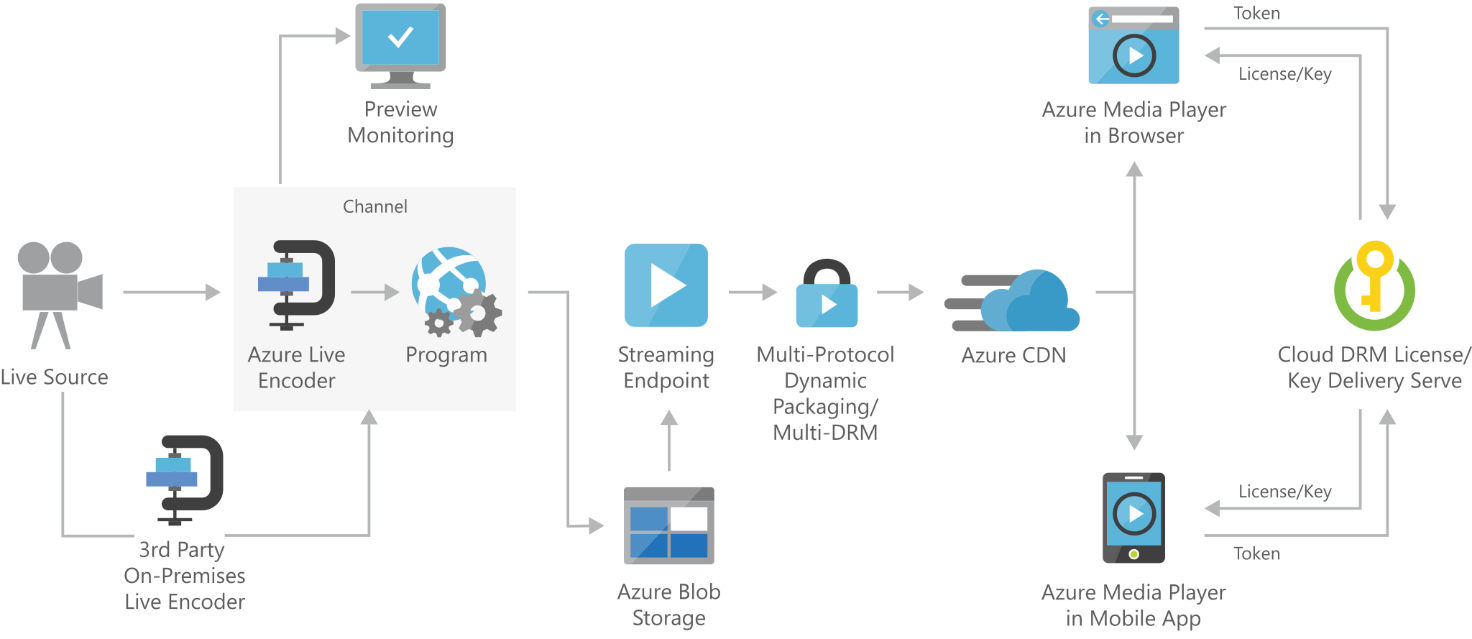


Рисунок 3.1 Архитектура видеосервиса на Azure

Компоненты

1. **Partner on-premises live encoder.** Для реализации нам понадобиться локальный программный кодировщик, который кодирует поток с нашей камеры и отправляет его в службу потокового вещания Media Services через протокол обмена сообщениями в реальном времени (RTMP). Самым популярным вариантом считается **Open Broadcaster Software** (OBS studio).
2. **Blob Storage.** Хранилище BLOB-объектов: хранит большие объемы неструктурированных данных, таких как текстовые или двоичные данные, к которым можно получить доступ из любой точки мира через HTTP или HTTPS. Вы можете использовать хранилище BLOB-объектов для публичного предоставления данных миру или для частного хранения данных приложения.
3. **Azure Media Services Encoder.** Кодировщик служб мультимедиа Azure: задания кодирования - одна из наиболее распространенных операций обработки в службах мультимедиа. Вы создаете задания кодирования для преобразования файлов мультимедиа из одной кодировки в другую.
4. **Azure Media Services Streaming Endpoint.** Конечная точка потоковой передачи служб мультимедиа Azure: служба потоковой передачи, которая может доставлять контент непосредственно в приложение клиентского проигрывателя или в сеть доставки контента (CDN) для дальнейшего распространения.
5. **Content Delivery Network.** Сеть доставки контента: обеспечивает безопасную и надежную доставку контента с широким глобальным охватом и богатым набором функций.
6. [**Azure Media Player**](https://docs.microsoft.com/en-us/azure/media-services/azure-media-player/azure-media-player-overview)**.** Проигрыватель мультимедиа Azure: использует отраслевые стандарты, такие как HTML5 (MSE / EME), для обеспечения широких возможностей адаптивной потоковой передачи. Независимо от используемой технологии воспроизведения, у разработчиков есть унифицированный интерфейс JavaScript для доступа к API.
7. **Multi-DRM content protection.** Защита контента с использованием нескольких DRM (Digital rights management): безопасная доставка контента с использованием нескольких DRM или шифрования с открытым ключом AES

# **3.2 Content Delivery Network**

Прежде чем преступить к реализации видеосервиса поговорим подробнее о такой важной составляющей как сетевой инфраструктуре CDN.

**CDN (Content Delivery Network)** – это географически распределённая сетевая инфраструктура, обеспечивающая быструю доставку контента пользователям веб-сервисов и сайтов. Входящие в состав CDN серверы географически располагаются таким образом, чтобы сделать время ответа для пользователей сайта/сервиса минимальным.

Платформа Azure также предоставляет сеть доставки содержимого **Azure CDN** предлагает разработчикам глобальное решение для быстрой доставки контента пользователям с высокой пропускной способностью путем кэширования их контента в стратегически расположенных физических узлах по всему миру. Azure CDN также может ускорить динамическое содержимое, которое невозможно кэшировать, используя различные сетевые оптимизации с помощью точек присутствия CDN. Например, оптимизация маршрута для обхода протокола пограничного шлюза (BGP).

Преимущества использования Azure CDN для доставки ресурсов веб-сайта включают:

* Лучшая производительность и улучшенный пользовательский интерфейс для конечных пользователей, особенно при использовании приложений, в которых для загрузки контента требуется несколько циклов обмена данными.
* Большое масштабирование для лучшей обработки мгновенных высоких нагрузок, таких как запуск события запуска продукта.
* Распределение пользовательских запросов и обслуживание контента непосредственно с пограничных серверов, чтобы на исходный сервер направлялось меньше трафика.

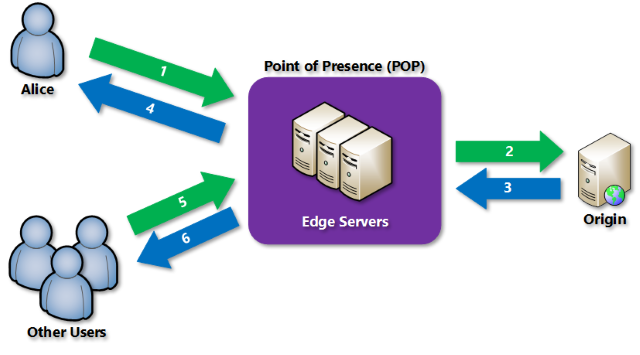
Как это устроено

Рисунок 3.2 Общая схема работы Azure CDN

1. Пользователь (Alice) запрашивает файл (также называемый ресурсом), используя URL-адрес со специальным доменным именем, например <endpointname>.azureedge.net. Это имя может быть именем хоста конечной точки или персональным доменом. DNS направляет запрос в точку присутствия с наилучшей производительностью, которая обычно является точкой присутствия, географически ближайшей к пользователю.
2. Если ни один пограничный сервер в точке присутствия не имеет файла в своем кэше, точка доступа запрашивает файл с исходного сервера. Исходным сервером может быть веб-приложение Azure, облачная служба, или любой общедоступный веб-сервер.
3. Исходный сервер возвращает файл на пограничный сервер в точке присутствия (PoP).
4. Пограничный сервер в точке присутствия кэширует файл и возвращает файл исходному запрашивающему (Alice). Файл остается в кэше на пограничном сервере в POP до тех пор, пока не истечет время жизни (TTL), указанное в его HTTP-заголовках. Если исходный сервер не указал срок жизни, срок жизни по умолчанию составляет семь дней.
5. Затем дополнительные пользователи могут запросить тот же файл, используя тот же URL-адрес, который использовала Alice, и также могут быть перенаправлены на ту же точку присутствия.
6. Если срок жизни файла не истек, пограничный сервер возвращает файл непосредственно из кэша. Этот процесс приводит к более быстрому и отзывчивому взаимодействию с пользователем.

# **3.3 Процесс реализации приложения**

Это приложение демонстрирует многостороннее видеоприложение, созданное с помощью Angular, SPA Web-Server, ASP.NET Core технологий.

Демонстрацию процесса использования можно увидеть на следующий изображениях:

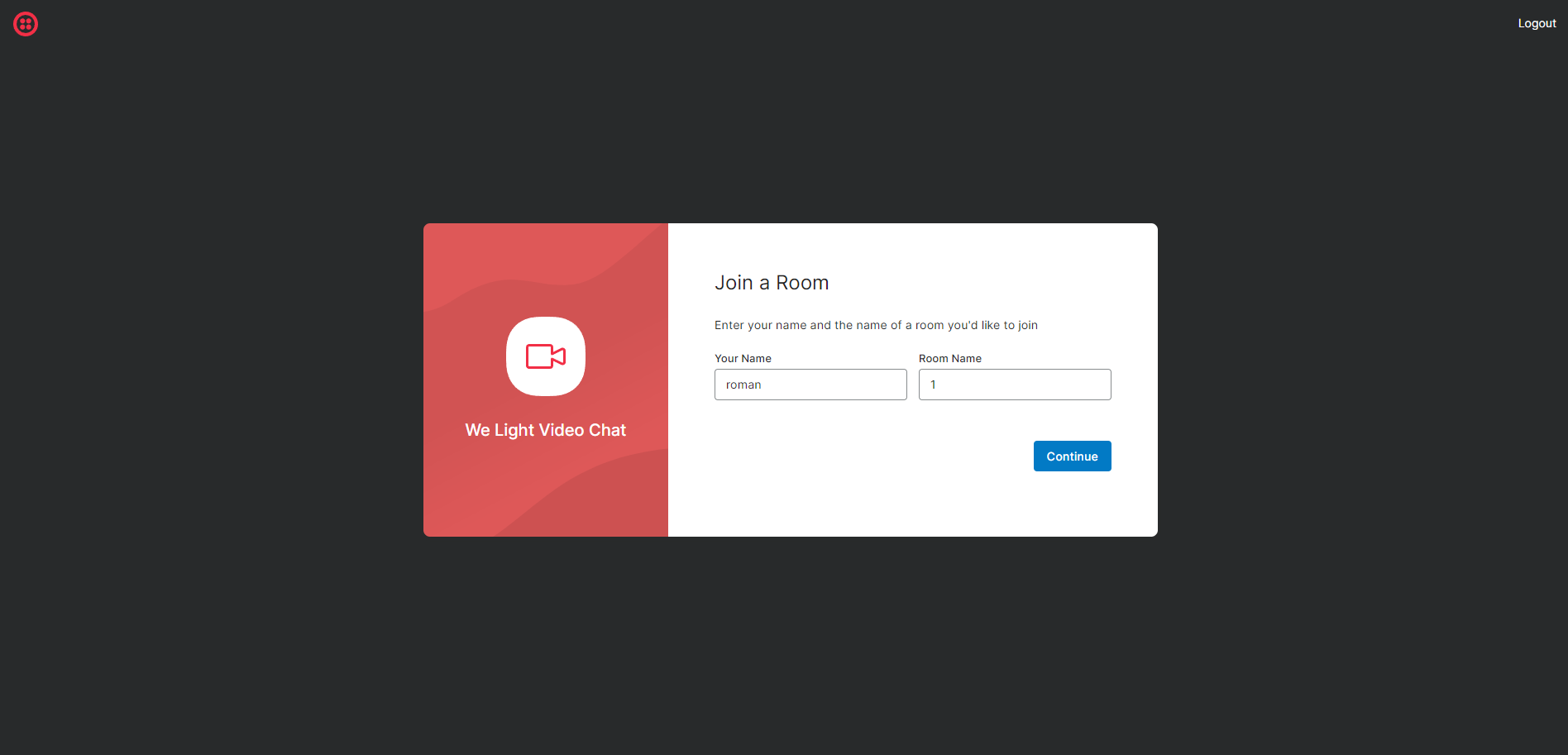


Рисунок 3.3.1 Авторизация

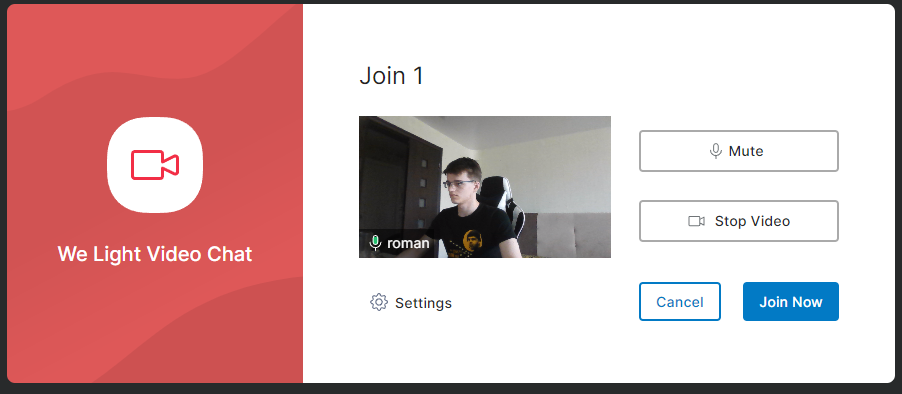


Рисунок 3.3.2 Настройка видео

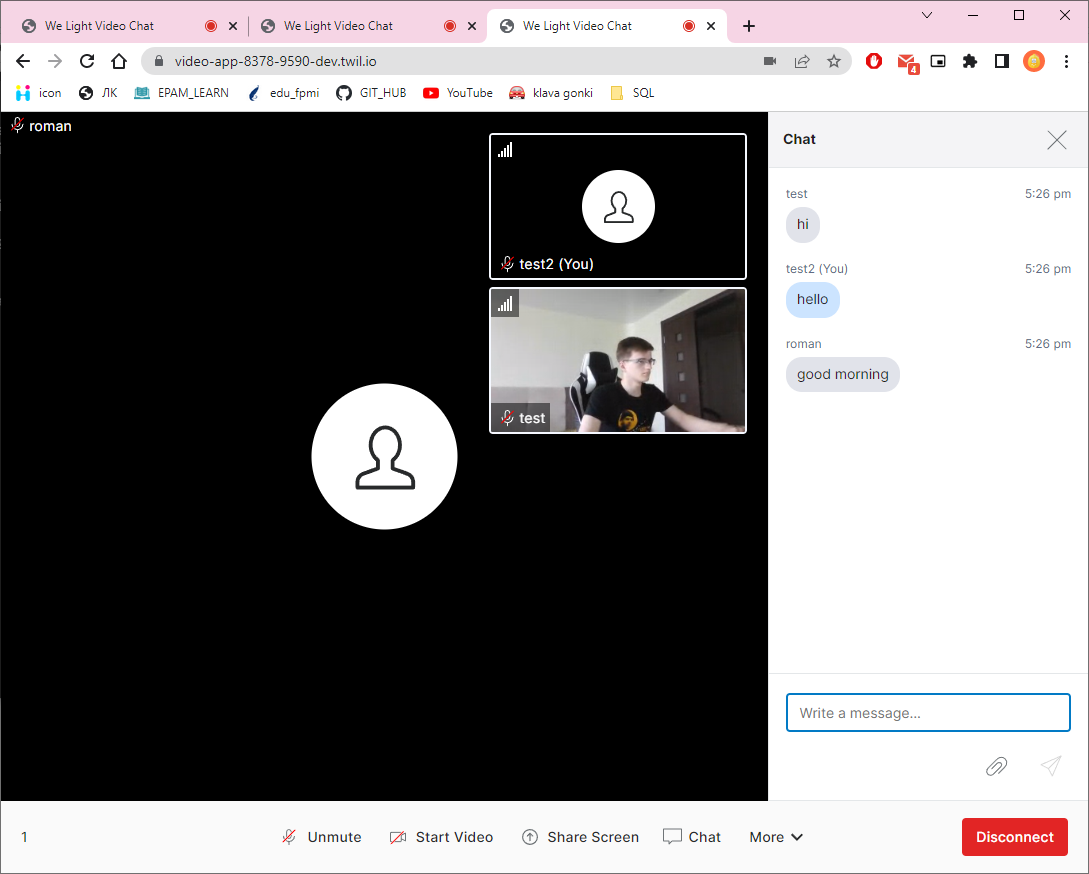


Рисунок 3.3.3 Демонстрация чата с несколькими пользователями

# **3.3.2 Подключение к Azure Services (Приложение А)**

Функция CreateMediaServicesClientAsync cоздает AzureMediaServicesClient объект, используя учетные данные.

GetCredentialsAsync функция создает ServiceClientCredentials объект на основе учетных данных, предоставленных в локальном файле конфигурации (appsettings.json) в корне проекта.

Объект AzureMediaServicesClient позволяет начать использовать API-интерфейсы служб мультимедиа с .NET. Чтобы создать объект, вам необходимо предоставить учетные данные для подключения клиента к Azure с помощью Azure Active Directory, реализованного в GetCredentialsAsync (Приложение А).

# **3.3.3 Создание LiveEvent (Приложение Б)**

При создании **LiveEvent** мы можем использовать следующие вещи:

* **Протокол загрузки для трансляции** . В настоящее время Azure поддерживает протоколы RTMP, RTMPS и Smooth Streaming. Мы остановимся на RTMP.
* **Ограничения по IP для загрузки и предварительного просмотра** . Вы можете определить IP-адреса, которым разрешено передавать видео на это событие в прямом эфире. Разрешенные IP-адреса можно указать одним из следующих вариантов:
  + Один IP-адрес (например, 10.0.0.1)
  + Диапазон IP-адресов, в котором используется IP-адрес и маска подсети бесклассовой междоменной маршрутизации (CIDR) (например, 10.0.0.1/22)
* **Автозапуск события по мере его создания** . Если для автозапуска установлено значение true, событие прямой трансляции начнется после создания.

# **3.3.3 Создание живого вывода (Приложение В)**

Прямые трансляции начинаются, когда они создаются, и прекращаются, когда они удаляются. Когда вы удаляете вывод в реальном времени, вы не удаляете базовый актив или контент в активе.

Думайте об этом как об извлечении «ленты». Актив с записью будет длиться столько, сколько вам захочется. Когда он будет извлечен (имеется в виду, когда прямой вывод будет удален), он будет немедленно доступен для просмотра по запросу (приложение В).

# **3.3.4 API Controller & Startup**

После успешного подключения к сервисам Azure Мы можем приступить к реализации Api controller, который будет служить мостом между UI и сервисами.

VideoController (Приложение Г) предоставляет HTTP GET запросы такие как api/video/token и api/video/rooms по которым мы сможем иметь доступ к нашему приложению. Так же определим основные модели UserSettings и RoomDetails, по которым мы сможем передавать данные о пользователе и комнате подключения.

Настроем наш файл Startup.cs для правильной настройки подключения всех сервисов (Приложение Д).

# **3.3.5 Angular rooms (Приложение Е)**

Для демонстрации работы приложения мною была использована платформа Angular 11.

Теперь, когда основные службы созданы, как они должны взаимодействовать друг с другом и как себя вести? Пользователи должны иметь возможность создавать комнаты или присоединяться к ним. Комната — это ресурс Azure, в котором может быть один или несколько участников. Участник также является ресурсом Azure.

Участники и комнаты используют общие камеры, которые отслеживают публикации как для аудио-, так и для видеодорожек. Приложение имеет компоненты Angular для каждого из них.

Сейчас я покажу принцип работы компонента комнаты. Он предоставляется RoomsComponent – интерфейс для создания комнат путем ввода roomName через <input type=’text’> элемента и <button> элемента, привязанного к onTryAddRoom методу класса.

RoomsComponent Подписывается на videoChatService.$roomsUpdated соыбтие. Каждый раз, когда создается комната, RoomsComponent будет сигнализировать о ее создании через это событие, и NotificationHub служба будет это прослушивать.

С помощью Azure Endpoint NotificationHub это сообщение передается всем другим подключенным клиентам. Этот механизм позволяет коду на стороне сервера предоставлять клиентским приложениям веб-функции в режиме реального времени. В этом приложении RoomsComponent будет автоматически обновляться список доступных комнат.

Под капотом, когда пользователь создает комнату, он подключается к этой комнате через Azure SDK.

Компонент комнаты ожидает имя комнаты и массив LocalTrack объектов. Эти локальные дорожки поступают из предварительного просмотра локальной камеры, которая предоставляет как аудио-, так и видеодорожку. Объекты LocalTrack публикуются в комнатах, к которым присоединяется пользователь, чтобы другие участники могли подписаться на них и получать их.

# **ЗАКЛЮЧЕНИЕ**

Созданная схема архитектуры хорошо адаптирована для реальных проектов.

В ходе выполнения работы были разработаны и решены следующие задачи:

1. Сравнение монолитной и микросервисной архитектуры.
2. Рассмотрены основные методы работы с контейнерами.
3. Разобрана архитектура стримингового сервиса Netflix.
4. Описаны основные концепции платформы Azure.
5. Реализовано приложение видеочата, основанного на сервисах Azure.

В планах на дальнейшее развитие этой темы – реализация полноценного видеосервиса с возможностью просмотра видео контента не только как Life Stream, но и реализация такой возможности как видео по запросу похожую функциональность можно встретить в таких сервисах как Netflix, Hulu и YouTube.

# **СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ**

* 1. **Cesar de la Torre, Bill Wagner**, Mike Rousos. Architecture for Containerized .NET Applications., 6 издание. — 372 с.: ил.
  2. Iain Foulds. Learn Azure in a month of lunches Second Edition., 2 издание 2020 г. — 370 с: ил.
  3. Azure Media Services v3 [Electronic resource] - Mode of access: [*https://docs.microsoft.com/en-us/azure/media-services/latest/*](https://docs.microsoft.com/en-us/azure/media-services/latest/)
  4. Azure Kubernetes Service (AKS) [Electronic resource]. – Mode of access: *https://docs.microsoft.com/en-us/azure/aks/*
  5. Irakli Nadareishvili, Ronnie Mitra. Microservice Architecture: Aligning Principles, Practices, and Culture.,2016 г. — 146 с: ил.
  6. Dom Robinson. Content Delivery Networks: Fundamentals, Design and Evolution., издание 2017 г. ­— 256 с.
  7. Valerio De Sanctis. ASP.NET Core 5 and Angular: Fullstack web development with .NET 5 and Angular 11, 4 издание, 2021 г. — 746 с.: ил.

# **ПРИЛОЖЕНИЯ**

# *Приложение А*

public static async Task<IAzureMediaServicesClient> CreateMediaServicesClientAsync(ConfigWrapper config, bool interactive = false)

{

ServiceClientCredentials credentials;

if (interactive)

credentials = await GetCredentialsInteractiveAuthAsync(config);

else

credentials = await GetCredentialsAsync(config);

return new AzureMediaServicesClient(config.ArmEndpoint, credentials)

{

SubscriptionId = config.SubscriptionId,

};

}

private static async Task<ServiceClientCredentials> GetCredentialsAsync(ConfigWrapper config)

{

var scopes = new[] { config.ArmAadAudience + "/.default" };

var app = ConfidentialClientApplicationBuilder.Create(config.AadClientId)

.WithClientSecret(config.AadSecret)

.WithAuthority(AzureCloudInstance.AzurePublic, config.AadTenantId)

.Build();

var authResult = await app.AcquireTokenForClient(scopes)

.ExecuteAsync()

.ConfigureAwait(false);

return new TokenCredentials(authResult.AccessToken, TokenType);

}

# *Приложение Б*

Console.WriteLine($"Creating a live event named {liveEventName}");

IPRange allAllowIPRange = new(

name: "AllowAll", address: "0.0.0.0", subnetPrefixLength: 0);

// Create the LiveEvent input IP access control object

LiveEventInputAccessControl liveEventInputAccess = new(){

Ip = new IPAccessControl( allow: new IPRange[] { allAllowIPRange })

};

// Create the LiveEvent Preview IP access control object.

LiveEventPreview liveEventPreview = new(){

AccessControl = new LiveEventPreviewAccessControl(

ip: new IPAccessControl(allow: new IPRange[] { allAllowIPRange })

)

};

LiveEvent liveEvent = new(

location: mediaService.Location,

description: "Sample LiveEvent from .NET SDK sample",

useStaticHostname: true,

input: new LiveEventInput(

streamingProtocol: LiveEventInputProtocol.RTMP,

accessToken: "acf7b6ef-8a37-425f-b8fc-51c2d6a5a86a",

accessControl: liveEventInputAccess

),

encoding: new LiveEventEncoding(

encodingType: LiveEventEncodingType.PassthroughStandard

),

preview: liveEventPreview,

streamOptions: new List<StreamOptionsFlag?>(){ StreamOptionsFlag.LowLatency }

);

// Start monitoring LiveEvent events using Event Grid and Event Hub

try{

Console.WriteLine("Creating a new client to process events from an Event Hub...");

var credential = new DefaultAzureCredential();

var storageConnectionString = "DefaultEndpointsProtocol=https;AccountName={ config.StorageAccountName};AccountKey={config.StorageAccountKey}";

var blobContainerName = config.StorageContainerName;

var eventHubsConnectionString = config.EventHubConnectionString;

var eventHubName = config.EventHubName;

var consumerGroup = config.EventHubConsumerGroup;

storageClient = new BlobContainerClient(

storageConnectionString, blobContainerName);

processorClient = new EventProcessorClient(

storageClient, consumerGroup, eventHubsConnectionString, eventHubName);

mediaEventProcessor = new MediaServicesEventProcessor(null, null, liveEventName);

processorClient.ProcessEventAsync += mediaEventProcessor.ProcessEventsAsync;

processorClient.ProcessErrorAsync += mediaEventProcessor.ProcessErrorAsync;

await processorClient.StartProcessingAsync();

}

catch (Exception e){

Console.WriteLine("Failed to connect to Event Hub, please refer README for Event Hub and storage settings. Skipping event monitoring...");

Console.WriteLine(e.Message);

}

# *Приложение В*

string manifestName = "output";

Console.WriteLine($"Creating a live output named {liveOutputName}");

Console.WriteLine();

watch = Stopwatch.StartNew();

LiveOutput liveOutput = new(

assetName: asset.Name,

manifestName: manifestName,

archiveWindowLength: TimeSpan.FromHours(1)

);

liveOutput = await client.LiveOutputs.CreateAsync(

config.ResourceGroup,

config.AccountName,

liveEventName,

liveOutputName,

liveOutput);

elapsedTime = String.Format(":{0:00}.{1:00}", watch.Elapsed.Seconds, watch.Elapsed.Milliseconds / 10);

Console.WriteLine($"Create Live Output run time : {elapsedTime}");

Console.WriteLine();

# *Приложение Г*

namespace VideoChat.Controllers

{

[ApiController, Route("api/video") ]

public class VideoController : ControllerBase

{

readonly IVideoService \_videoService;

public VideoController(IVideoService videoService)

=> \_videoService = videoService;

[HttpGet("token")]

public IActionResult GetToken()

=> new JsonResult(new { token = \_videoService.GetJwt(User.Identity.Name) });

[HttpGet("rooms")]

public async Task<IActionResult> GetRooms()

=> new JsonResult(await \_videoService.GetAllRoomsAsync());

}

}

namespace VideoChat.Models

{

public record RoomDetails(

string Id,

string Name,

int ParticipantCount,

int MaxParticipants);

}

namespace VideoChat.Options

{

public class UserSettings

{

public string AccountSid { get; set; }

public string ApiKey { get; set; }

public string ApiSecret { get; set; }

}

}

# *Приложение Д*

public class Startup {

readonly IConfiguration \_configuration;

public Startup(IConfiguration configuration) => \_configuration = configuration;

public void ConfigureServices(IServiceCollection services) {

services.AddControllersWithViews();

services.Configure<UserSettings>(settings => {

settings.AccountSid = Environment.GetEnvironmentVariable("Azure\_ACCOUNT\_SID");

settings.ApiSecret = Environment.GetEnvironmentVariable("Azure\_API\_SECRET");

settings.ApiKey = Environment.GetEnvironmentVariable("Azure\_API\_KEY");

})

.AddTransient<IVideoService, VideoService>()

.AddSpaStaticFiles(config => config.RootPath = "App/dist");

}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {

if (env.IsDevelopment()) {

app.UseDeveloperExceptionPage();

}

else {

app.UseHsts();

}

app.UseHttpsRedirection()

.UseStaticFiles()

.UseSpaStaticFiles();

app.UseRouting();

app.UseEndpoints(endpoints => {

endpoints.MapControllerRoute(

name: "default",

pattern: "{controller}/{action=Index}/{id?}");

})

.UseSpa(spa => {

spa.Options.SourcePath = "App";

if (env.IsDevelopment()) {

spa.UseAngularCliServer(npmScript: "start");

}

});

}

}

# *Приложение Е*

import{ Component, OnInit, OnDestroy, EventEmitter, Output, Input } from '@angular/core';

import { Subscription } from 'rxjs';

import { tap } from 'rxjs/operators';

import { NamedRoom, VideoChatService } from '../services/videochat.service';

@Component({

selector: 'app-rooms',

styleUrls: ['./rooms.component.css'],

templateUrl: './rooms.component.html',

})

export class RoomsComponent implements OnInit, OnDestroy {

@Output() roomChanged = new EventEmitter<string>();

@Input() activeRoomName: string;

roomName: string;

rooms: NamedRoom[];

private subscription: Subscription;

private readonly videoChatService: VideoChatService) { }

async ngOnInit(){

await this.updateRooms();

this.subscription =

this.videoChatService

.$roomsUpdated

.pipe(tap(\_ => this.updateRooms()))

.subscribe();

}

ngOnDestroy() {

if (this.subscription) {

this.subscription.unsubscribe();

}

}

onTryAddRoom() {

if (this.roomName) {

this.onAddRoom(this.roomName);

}

}

onAddRoom(roomName: string) {

this.roomName = null;

this.roomChanged.emit(roomName);

}

onJoinRoom(roomName: string) {

this.roomChanged.emit(roomName);

}

async updateRooms(){

this.rooms = (await this.videoChatService.getAllRooms()) as NamedRoom[];