# Мобильное приложение: нативное, web, гибридное

Первое решение, которое нужно принять – это тип приложения. Поскольку требуется подключение сторонних датчиков, то вариант использовать чистое web-приложение (например, разработанное по технологии PWA) отпадает. Поскольку приложение должно работать в любой точке, то лучше избегать лишних сетевых соединений, по возможности приложение должно работать автономно, а при подключении к сети синхронизировать информацию с серверной частью. Поэтому нативное приложение следует предпочесть гибридному. Нативное приложение. Можно писать на разных языках (Java/Kotlin для Android и для Swift для IOS) или на кроссплатформенном фреймворке, например, Flutter. Выберем первый вариант т. к. он будет проще в сопровождении, фреймворки могут содержать малоизвестные ошибки, редко обновляться и обычно плохо оптимизированы.

**Риски:** это увеличит объемы разработки (нужно делать два приложения вместо одного), но сократит объемы доработок (высока вероятность с толкнуться с функционалом не предусмотренным фреймворком который будет сложно реализовать) и упростит поддержку.

# Размещение в облаке

**PaaS** (Платформа как сервис): Предоставление полноценной среды для разработки, тестирования, развертывания, размещения и обслуживания приложений в облаке для организации процессов, разработки и тестирования; Поставщик услуг запускает и управляет средами от уровня инфраструктуры до уровня прикладных служб в стеке;

**APaaS** (Инструментальная платформа как сервис): Предоставление компонентов приложений, инструментов развертывания и совместной работы, предварительно настроенных шаблонов и виджетов для быстрого создания приложений. Поставщик услуг запускает и управляет средами и процессом от инфраструктуры до уровня прикладных служб, включая некоторый программный код и инструменты, а также управление данными и пользовательскими интерфейсами;

**IaaS** (Инфраструктура как сервис): Предоставление физического пространства ЦОД, а также физических и виртуальных процессоров, дискового пространства и служб баз данных в качестве услуги; Возможность выбора наиболее удобной модели финансирования для оптимизации затрат; Переход к модели потребления инфраструктурных услуг по подписке.

**SaaS** (Программное обеспечение как сервис). Это полностью готовое решение, которое сразу же можно использовать. Примерами реализации услуг по принципу SaaS являются конструкторы сайтов, почтовые сервисы, различные CRM-системы, 1С в облаке, планировщик Google и т.д. – не рассматривается из-за узкой специализации.

**Гиперконвенгентные решения** представляет собой единую программно-определяемую среду, которая сочетает в себе элементы традиционного оборудования, устанавливаемого в ЦОД. Является интегрированным решение на базе ПО и серверов х86, благодаря чему способно заменять дорогое специализированное оборудование

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | PaaS | APaaS | IaaS | Гиперконвергентны е решения |
| Надежность | Высокая надежность определяется инфраструктурой и качеством операционной поддержки со стороны провайдера |  | Высокая надежность определяется инфраструктурой и качеством операционной поддержки со стороны провайдера |  |
| Масштабируемость | Сокращение времени на выделение дополнительны х ресурсов (возможность масштабирования по требованию) |  | Возможность быстрого наращивания и сокращения ресурсов инфраструктуры провайдеров | Возможность быстрого наращивания ресурсов за счет встроенных механизмов |
| Гибкость | Возможность быстрого изменения объема и набора услуг по требованию | Предоставление гибких инструментов для разработки по требованию | Возможность гибкой конфигурации используемых инфраструктурных компонентов |  |
| Экономическая эффективность |  |  | Снижение общей стоимости владения инфраструктурой за счет эффекта «экономии на масштабе» | Упрощение процесса и снижение стоимости по мере расширения потребности в инфраструктуре |
| Инновационность | Возможность использования передовых тех. стеков для организации процессов разработки и тестирования | Использование инновационных технологических стеков для разработки и развертывания |  | Использование современных стеков ПО (виртуализация и др.) |
| Операционная эффективность |  |  | Снижение затрат на развитие и поддержку ИТ- инфраструктуры (задачи на стороне провайдера) | Более простое управление за счет уменьшения числа управляемых систем |

SaaS и APaaS не подходят т.к. это нишевые решения для приложений определенного типа, PaaS может являться проблемой в плане Vendor Lock – это ограничит переносимость системы и ограничит выбор используемых технологий – кроме проприетарных технологий, PaaS не поддерживает некоторые компоненты, это значит, что часть компонент придется поддерживать самостоятельно, в то время как остальные компоненты будут поддерживаться облачным провайдером. Так же сервера приложения могут быть распределены широко географически – не везде может присутствовать выбранный провайдер PaaS. Оптимальным решением будет IaaS – такое приложение легко перенести от одного провайдера к другому или развернуть новый сервер у другого провайдера. Так же это финансово оптимальное решение.

**Риски:** возможно это будет не так удобно в плане сопровождения, но позволит легко сменить облачного провайдера.

# Интеграция между приложениями компании SSO и DeepLink

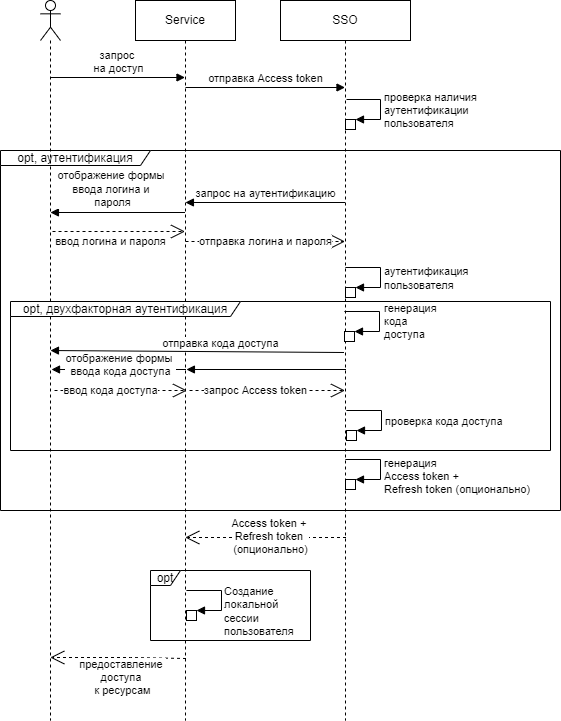
Приложения компании должны легко интегрироваться между собой, мы будем понимать под этим бесшовный переход: например, из нашего приложения из инвентаря/новостей/промоакций можно перейти в приложение-магазин, причем логин/пароль при этом вводить не нужно и переход осуществляется на определенный экран в другом приложении. Так или иначе все опции подразумевают использование общего аккаунта для всех приложений компании или передачу логина/пароля – т.е. использование хотя бы центрального хранилища данных для аутентификации. Будем использовать систему единого входа Single-Sign-On (SSO) – это будет отдельный модуль для всех приложений компании, отвечающий за авторизацию и аутентификацию пользователя. Так же это будет единственный компонент, хранящий персональные данные. (к нему могут применяться другие требования законодательства, например

Описание диаграммы последовательности для получения пользователем доступа к ресурсу:

1. Пользователь через браузер запрашивает доступ к Приложению.
2. Приложение направляет данные о Пользователе в Модуль авторизации.
3. Модуль авторизации проверяет был ли аутентифицирован данный Пользователь. Если нет, то выполняется процесс аутентификация.
4. [опционально] Аутентификация
   1. Модуль авторизации выполняет запрос в приложение о данных о Пользователе (логин и пароль) для проведения процедуры аутентификации.
   2. Приложение отображает форму для ввода логина и пароля.
   3. Пользователь вводит логин и пароль в приложении
   4. Приложение передает логин и пароль в модуль авторизации.
   5. [опционально, если включена двухфакторная аутентификация]
      1. Модуль авторизации генерирует код доступа
      2. Модуль авторизации передает код доступа пользователю (например, SMS)
      3. Модуль авторизации оповещает приложение о том, что запрошен код доступа
      4. Приложение отображает форму для ввода кода доступа
      5. Пользователь вводит код доступа
      6. Приложение передает код доступа в модуль авторизации
      7. Модуль авторизации проверяет код доступа
   6. Модуль авторизации генерирует токен доступа (Access token) и токен обновления (Refresh token), если он определен грантами системы.
5. [опционально] Приложение создает локальную сессию пользователя
6. Приложение предоставляет Пользователю доступ к ресурсам приложения.

Для перехода на нужный экран в другом приложении будем использовать технологию DeepLink, если у пользователя на мобильном устройстве установлено соответствующее мобильное приложение, то ссылка откроется в нем, если нет – ссылка откроется в браузере. Здесь мы подразумеваем что приложение магазина существует как в web-версии, так и в версии мобильного приложения.

**Риски:** SSO – единый компонент для всех систем компании и единая точка отказа, так же для разных приложений компании нужен разный уровень безопасности и для нашего приложения он может показаться избыточным.

хранение на территории определенных стран или развертывание в приватном облаке).

# Выделение доменов

Глобально функционал приложения можно разделить на три домена:

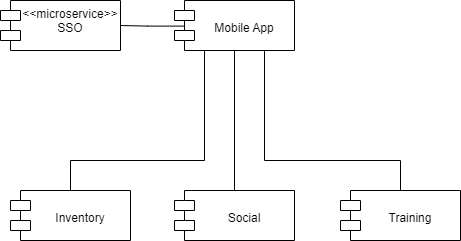
Social – группы, чаты, поиск пользователей, уведомления, списки друзей и групп для пользователей. Работа с этим доменом требует подключения к сети.

Training – тренировки и все что с ними связано: подключаемые датчики, расписание тренировок, список участников, маршрут, таблица результатов. Работа с тренировками может производиться автономно на устройстве пользователя. Так же тренировка может быть групповой – тогда требуется подключение к сети.

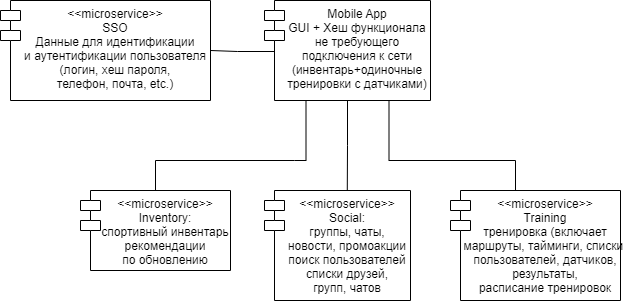
Inventory – то, что связано только с пользователем – его спортивный инвентарь.

Так же техническими компонентами будут собственно *мобильное приложение* ‑ компонент, установленный на стороне клиента и *SSO* – компонент общий для всех приложений компании.

Так и разобьём приложение на микросервисы: три микросервиса соответствующие доменам, SSO и само приложение



Распределим функции по доменам:



Взаимодействие сервисов: будет 2 типа взаимодействия

**Риски:** разделение на сервисы по доменам приводит к увеличению трафика, возможно стоило разделить на 2 домена – функции требующие подключения к сети – обязателен запрос к backend и те что можно выполнять на стороне клиента без подключения к сети а потом синхронизировать (инвентарь и тренировки объединить в один сервис).

# Размещение в кластере Kubernetes

Будем использовать контейнеризацию Docker и оркестрацию Kubernetes – это просто стандарт отрасли и выбором, по сути, не является. Использование оркестрации – это единственное решение, позволяющее приложению масштабироваться автоматически при увеличении/уменьшении нагрузки. Искать аналоги широко распространенных open-source решений тоже смысла нет. Размещение в Kubernetes автоматически добавляет в архитектуру 2 компонента

**Service Discovery** ‑ Сервисы в Kubernetes имеют динамические IP (т.е. после падения сервис восстанавливается на новом IP). Service Discovery регистрирует сервисы – если сервису 1 нужно получить доступ к сервису 2, то он обращается в Service Discovery и узнает где находятся все реплики сервиса 2

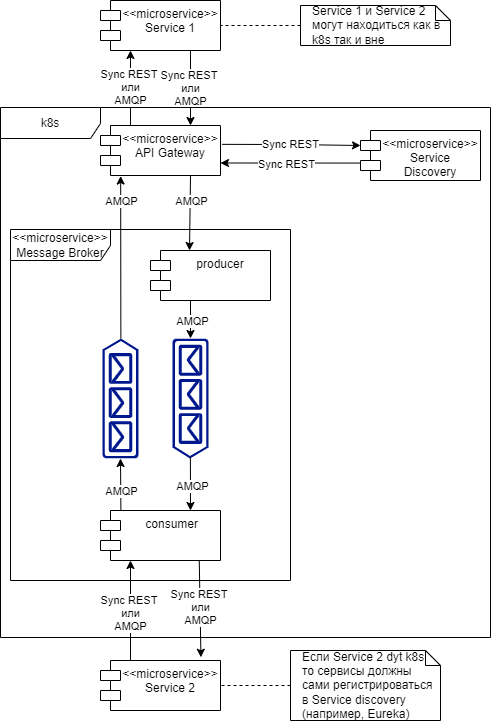
**API Gateway** или **Ingress Controller** – выполняет роль маршрутизации запросов извне к сервисам (консолидация запросов – все запросы извне идут через единую точку входа) и выполняет балансировку нагрузки между репликами сервиса. Информацию о том, где находятся сервисы он получает из Service Discovery.



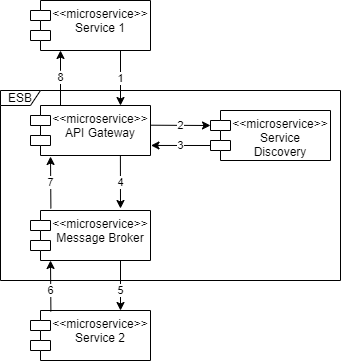
**Риски:** Сложнее сопровождение.

# Синхронное VS Асинхронное взаимодействие

Приложение должно работать в сетях с низким уровнем подключения (например, за городом/в дикой местности/во время движения). При нестабильной сети приложение должно работать автономно, а при восстановлении соединения синхронизироваться сервером, так же сеть может пропасть до того, как пришел ответ – т. е. нужно использовать асинхронные вызовы API и механизм гарантированной доставки сообщений (в обе стороны) – брокер сообщений. Так же это решит проблему с резкими всплесками количества посетителей (например, на соревнованиях). В идеальном варианте все API должны быть асинхронными (например, взаимодействовать по протоколу AMQP –Advanced Message Queuing Protocol), но на практике асинхронные API сложно описывать и тестировать, иногда требуется выставлять наружу REST API, иногда требуется использовать внутри сервиса внешние REST API. Брокер состоит из следующих компонент: Producer (отправляет сообщения в нужную очередь), Consumer (считывает сообщения из очереди и обрабатывает их) и самих очередей (могут быть типа точка-точка или издатель-подписчики). Consumer при этом может в качестве обработки события вызывать синхронный REST API. API Gateway так же может выставлять наружу синхронный API. Т.к. и брокер сообщений и API Gateway являются обычными микросервисами они могут масштабироваться независимо и при резком увеличении нагрузки будет реплицироваться API Gateway – на него будут идти синхронные подключения без таймаутов, которые будут преобразовываться в сообщения и затем обратно в REST API (уже с таймаутами).Но по возможности будем использовать везде нативным образом асинхронный API.



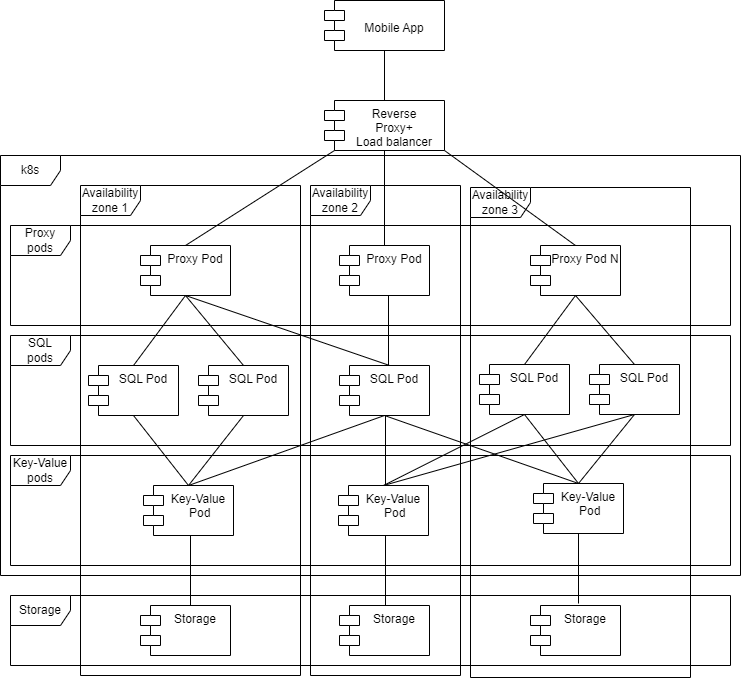
По сути API Gateway, Service Discovery и Message Broker представляют из себя минимальный набор для использования в качестве сервисной шины предприятия (Enterprise Service Bus, ESB)



**Риски:** Брокер сообщений — это единая точка отказа – он должен иметь большую доступность чем все остальные сервисы. Так же он создает дополнительную Latency.

# Выбор БД и консистентность

Приложение будет широко распространено географически и хорошо было бы так же иметь распределенную БД (Не репликацию, где копии находятся в одной географической зоне а распределенную БД где копии могут быть разнесены на значительные расстояния – это уменьшает время доступа клиентов из разных точек, но усложняет распределенные транзакции – между копиями имеется заметная Latency – поэтому использование SQL БД c применением блокировок (в основе которых лежат двухфазные или трехфазные коммиты) будет не лучшим выбором, так же это не будет работать с брокером сообщений. Альтернативный вариант – Использование NoSQL БД с применением кворума, например в Elasticsearch все ноды в любой момент времени возвращают одинаковый результат – голосование всех копий простым большинством.) Для взаимодействия между сервисами можно применять шаблон Saga. Но оптимальным решением будет применение коробочного решения Distributed SQL – распределенная NoSQL БД, предоставляющая SQL API. При этом внутри используется алгоритм консенсуса – все ноды в любой момент времени так же отдают одинаковый результат голосования всех нод. При этом взаимодействие с БД со стороны разработчика происходит так же, как одиночной SQL БД в синхронном режиме. Выберем open-source БД от Google – CockroachBD.

 Для обеспечения согласованности данных CockroachBD требует, чтобы все операции записи распространялись на кворум данных, для этого используется протокол согласия Raft.

CocroachDB состоит из следующих слоев:

“SQL layer” выставляет наружу SQL API и конвертирует SQL-запросы в низкоуровневые запросы на чтение/запись в нижележащий “слой Key-Value”. Состоит из следующих компонент: “SQL API” – формирует пользовательский интерфейс, “Parser” – конвертирует SQL в abstract layer tree (AST), “Cost-based optimizer” – оптимизирует AST и конвертирует в логический запрос, “Physical planner” – конвертирует логический запрос в физический на одной или нескольких нодах, “SQL Execution Engine” – выполняет физический запрос – чтение/запись в следующий “Слой Key-Value”.

“Transaction layer” реализует поддержку ACID транзакций, путем координации конкурентных операций. Используется протокол коммитов “Parallel Commits”.

“Distribution layer” делает всю информацию в БД доступной из любой ноды.

“Distribution layer” – копирует информацию между нодами для поддержания согласованности. Используется протокол консенсуса Raft.

“Storage Layer” – записывает информацию на диск. Используется движок хранения информации “Pebble”. Информация хранится в формате “Ключ-значение”

Каждый кластер для одного региона имеет как минимум три узла, распределенных по трем зонам доступности (“Availability Zone”, AZ) в регионе поставщика облачных услуг. Кластеры с несколькими регионами аналогичны кластерам с одним регионом, где узлы распределены по трем или более зонам доступности в каждом регионе

Перед кластером CocroachDB ставится Load Balancer и Proxy выполняющий роль API Gateway и дополнительной балансировки.

**Риски:** Возможная излишняя репликация. Все данные в CocroachDB реплицируются минимум три раза, но не все данные одинаково важны с точки зрения бизнеса. Так же нет примеров успешного долгосрочного использования в больших компаниях – есть риск столкнуться с малоизвестной проблемой.

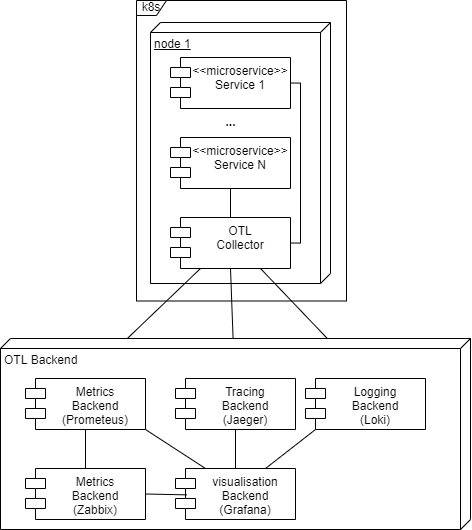
# Выбор стратегии Observability

Единственный выбор здесь по сути – использовать стандарт для всех телеметрических данных (OpenTelemetry) или нет. Использование стандарта усложнит разработку, но упростит смену ПО, используемого для сбора телеметрических данных. Что касается развёртывания то коллектор (коллектор OTL или отдельные коллекторы для разного ПО) можно размещать рядом с установленным ПО (в Pod или на ноде) или отдельно (например, вне k8s).

Выберем использование стандарта OTL и размещение коллектора рядом с установленным ПО

Само ПО для сбора телеметрии при таком выборе роли не играет -его в любой момент можно заменить – выберем самый популяный стек технолгогий: Prometeus, Zabbix, Jaegger, Loki, Grafana

**Риски:** увеличение сроков разработки



# Выбор брокера сообщений

Не будем рассматривать нишевые коммерческие решения (типа IBM MQ) и малоизвестные open-source (типа Celery на Python). Так же по возможности не будем использовать технологии зависимые от языка программирования – например очереди на JMS (Java Message Server) типа Active MQ. Будем рассматривать open-source решения не зависимые от языка программирования (поддерживающие протокол AMQP). Так же не будем рассматривать брокеры без гарантированной доставки сообщений (типа Zero MQ). По сути остается два брокера сообщений которые сейчас широко используются и являются стандартом в индустрии: RabbitMQ и Apache Kafka.

RabbitMQ – open source брокер сообщений основанный на очередях сообщений. RabbitMQ принимает сообщения от поставщиков и отправляет им подтверждение о приеме, а затем перенаправляет их подписчикам. Получатели подтверждают, что сообщение доставлено, либо сигнализируют о неудаче. Во втором случае сообщение остается в очереди, пока не будет доставлено. А после доставки оно удалится из системы. Основная фишка RabbitMQ — это гибкая маршрутизация сообщений между различными поставщиками и потребителями событий. Решение не ограничивается созданием простой очереди данных между двумя сторонами. В сервере реализована концепция принимающих события узлов (эксчейнджей) — они маршрутизируют данные в разные очереди сообщений RabbitMQ. Основные характеристики:

* Акцент на обмене сообщениями с возможностью поддержки больших потоков данных.
* Основная особенность — расширенный функционал маршрутизации.
* Хостинг на месте и поддержка сторонних модулей.
* Хотя изначально обработка потоков данных не поддерживалась и сообщения, как правило, обрабатывались однократно без возможности повторной обработки, эти слабые стороны были устранены по мере развития RabbitMQ
* Взаимодействие по протоколу AMQP, а через дополнительные модули — и с помощью некоторых других протоколов: MQTT, HTTP и так далее
* RabbitMQ реализует концепцию push-доставки: поставщик может направить новые события в сеть, но получатель не может их запросить у поставщика. При этом система не гарантирует порядок доставки сообщений.

Apache Kafka – open source брокер сообщений основанный на логах. Kafka реализует архитектурную концепцию распределенного лога, в который заносят информацию разные поставщики данных. Затем информацию из этого лога читают другие программы и микросервисы. Kafka ориентирован на высокую пропускную способность потока данных. Основные характеристики:

* Акцент на потоковом контенте, работа с большими потоками данных.
* Основные возможности: обеспечение сохранности сообщений и их многократная повторная обработка.
* Хостинг на месте и поддержка сторонних модулей.
* Сообщения сохраняются в брокере, подписчики подписываются на них, при этом гарантируется, что сообщения находятся в той же последовательности, в которой поступили.
* Параллелизм и распределенная архитектура хорошо сказываются на надежности: даже выход из строя части кластера не нарушит доставку сообщений.
* Для маршрутизации сообщений могут применяться Routing Keys, похожие на те, что используются в RabbitMQ. Но, в отличие от RabbitMQ, Apache Kafka гарантирует порядок доставки сообщений.
* Следует выбрать **RabbitMQ**, если нужна надежность и гибкость маршрутизации, а порядок доставки сообщений безразличен.
* Следует выбрать **Apache Kafka**, если предполагаются большие нагрузки, важна масштабируемость, доставка сообщений в правильном порядке и возможность просматривать историю сообщений.

В данном случае выбор не принципиален.

# Выбор API Gateway:

API Gateway должен выполнять следующие функции:

* Маршрутизация запросов – функция аналогичная реверс-прокси, создание единой точки входа в приложение.
* Балансировка нагрузки
* Преобразование между протоколами (в частности, синхронный (REST) в асинхронный (например, AMQP))

Так же следует рассмотреть возможность использования двух API Gateway – внутреннего – для общения между микросервисами внутри k8s и внешнего – для запросов из мобильного приложения.

Нам не нужна функция агрегации API поэтому мы не будем рассматривать использование API шлюза на основе GraphQL. Так же не будем рассматривать платформо-зависимые решения (типа Spring Cloud Gateway).

Нам не нужны авторизация и аутентификация – этим будет заниматься SSO.

Нам не принципиальны возможности мониторинга, но мы не собираем отдельно бизнес-метрики, так что сбор метрик типа количества вызовов API клиентом, отчетов по ошибкам, SLI будут плюсом при прочих равных условиях.

Так же при прочих равных плюсом будут канареечные релизы.

Т. к. эта инфраструктура может переиспользоваться в других приложениях и развиться в общую ESB для компании целесообразно рассматривать API Management System, а не просто API Gateway – там добавятся функции для разработки, публикации и тестирования API.

Выберем Open Source решение WSO2.

# Прочая инфраструктура:

Для балансировки нагрузки всех входящих соединений извне в кластер k8s и из кластера k8s в кластер баз данных будем использовать реверс-прокси с балансировкой нагрузки. Выберем open-source решение HAProxy – у него хорошая балансировка нагрузки, и его можно реплицировать через keepliveId.

Все соединения извне внутрь корпоративной сети должны идти через Web Application Firewall, расположенный в DMZ.

В кластере можно выделить Configmap или Сервис конфигурирования – для управления конфигурациями во всех остальных сервисах.

# Итоговая архитектура:

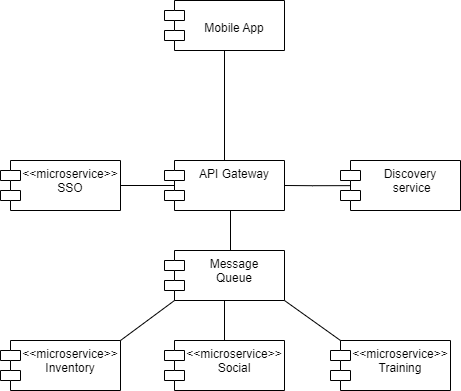
Диаграмма компонент:  


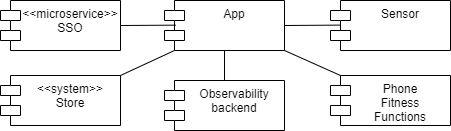
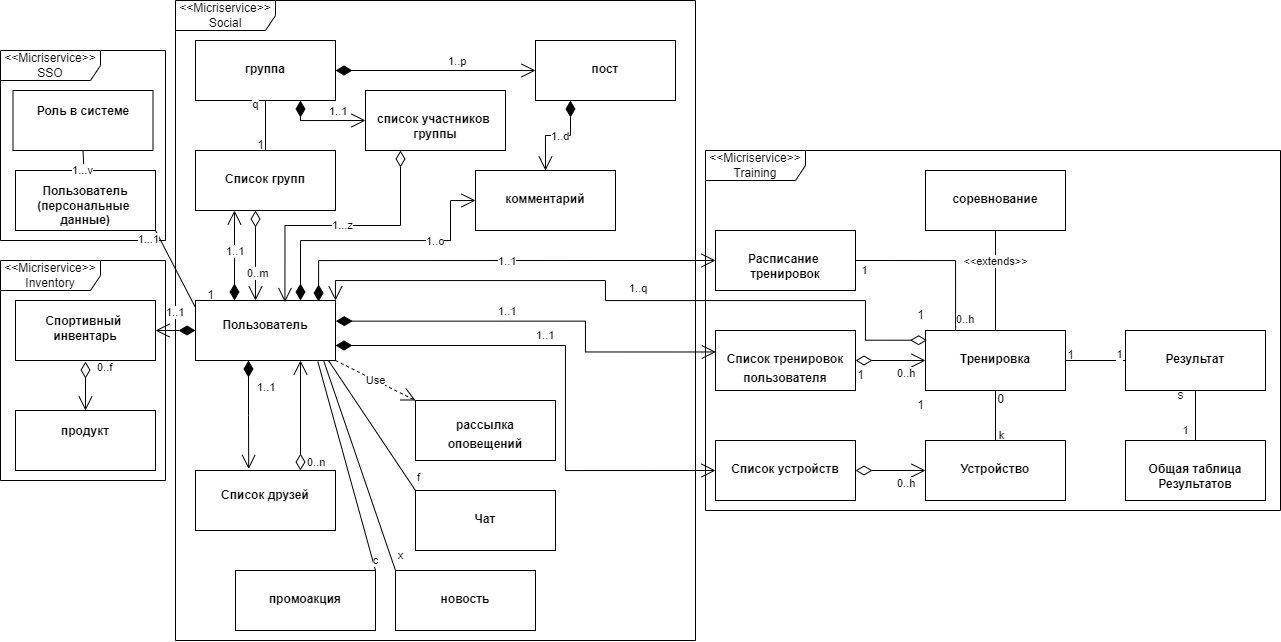
Диаграмма контекста:  


Диаграмма сущностей:  


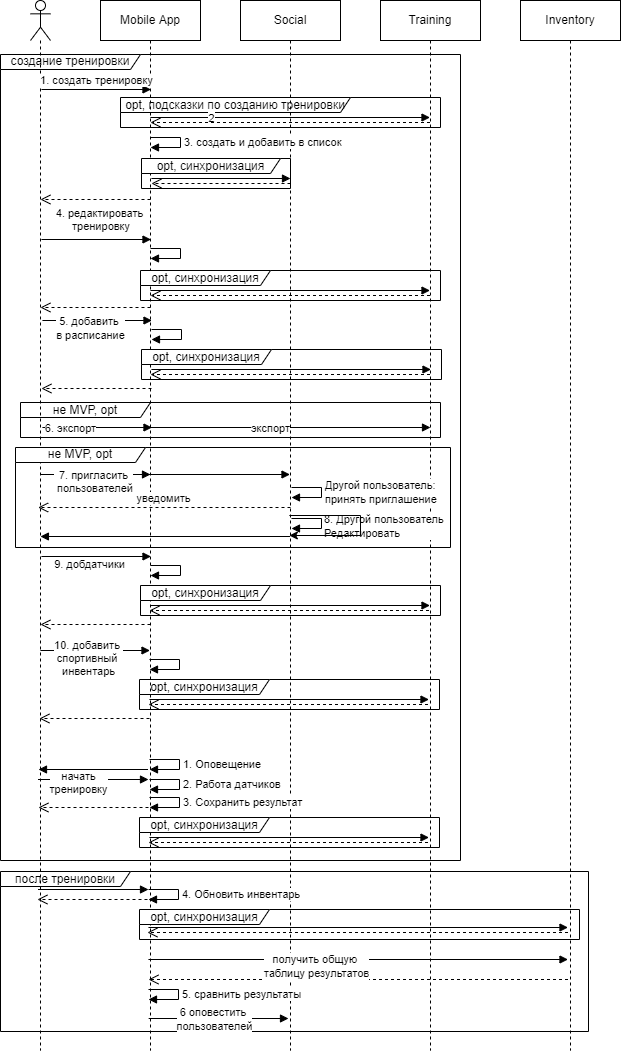
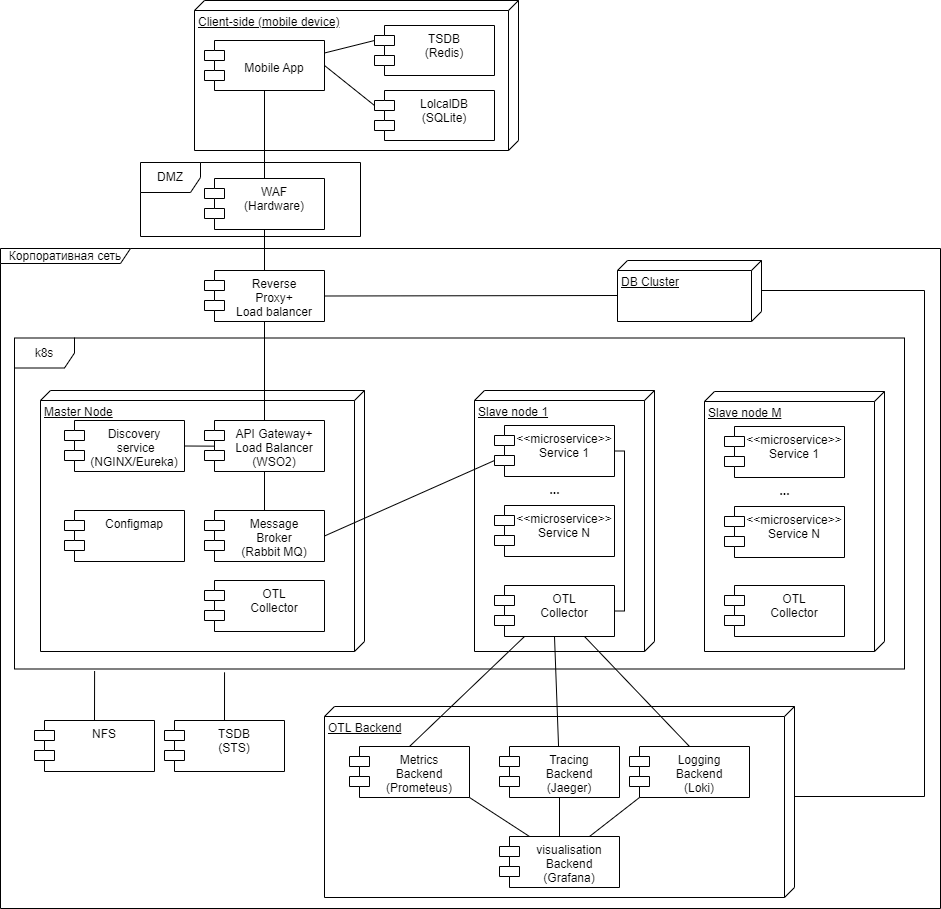
Пример диаграммы последовательности:

Диаграмма развертывания:



Оценка требуемых ресурсов: Одно ядро CPU в среднем дaет 200 одновременных подключений, при этом для стабильной работы требуется около 1Gb оперативной памяти. Итого для нагрузки, например, в 10000 одновременных подключений потребуется около 50 ядер и 50Gb оперативной памяти (на самом деле немного меньше), что при использовании обычных 8-ядерных процессоров составит около 6 машин. (Соответственно если в пике проектируемая нагрузка удваивается, то облако должно предоставлять возможность добавления еще 50 ядер и 50Gb RAM). Здесь не учитывается возможность кеширования и совмещения Master и Worker нод, а потому, удвоим RAM для машин с Kubernetes. Внешний Proxy будет в 2 экземплярах на каждый географический регион (допустим у нас минимум три региона: Европа, Азия, Америка). Отведем под них еще 6 машин по 4 ядра и 8Gb оперативной памяти на экземпляр (итого+24 ядра, +24 RAM)

Оценим аудиторию в 100000 пользователей (в требованиях 10000 одновременных подключений – каждый десятый). Выделим каждому пользователю минимум по 100 Мб итого нам нужно около 10 TB дискового пространства, с учетом трехкратной репликации: 30 TB. Это то, что хранится в БД Distributed SQL – тренировки, инвентарь, переписка, списки групп, посты. Если пользователь может бесконтрольно загружать медиаконтент (фото) то можно смело добавлять еще столько же и даже в два раза больше (с учетом того что медиа контент будет храниться в NFS и не будет реплицироваться). Накинем еще 24 ядра и RAM на сами БД.

Итого грубо оценим 100 ядер, 150 RAM, 30ТБ дискового пространства с возможностью добавить 50 ядер 100 RAM. Взята оценка по максимуму – пользователь заполнит 100 Мб (не используя медиаконтент) за несколько лет (далее старый контент можно удалять). Можно оценить, что количество пользователей (и требуемых ресурсов может удвоиться за несколько лет и выйти на плато – все-таки это нишевое приложение, врядли количество пользователей может резко вырасти).