Санкт-Петербургский государственный университет
Направление: 02.03.02 «Фундаментальная информатика и информационные технологии
ООП: Программирование и информационные технологии

ОТЧЕТ О НАУЧНО-ИССЛЕДОВАТЕЛЬСКОЙ РАБОТЕ

Тема задания: Построение серверной составляющей веб-приложения для медиа-платформы магазина дизайнерских эскизов одежды с интеграцией фриланс-биржи портных.

Выполнил: Панюшин Даниил Васильевич ___ 19.Б12-пу
Фамилия И. О. номер группы

Руководитель научно-исследовательской работы: Раевская Анастасия Павловна, доцент кафедры математической теории экономических решений Санкт-Петербургского Государственного Университета, кандидат физико-математических наук
должность, ученая степень, ФИО

Санкт-Петербург
2022

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Описание проекта

Существует множество различных брендов, создающих вещи самых разных ценовых категорий. В таких компаниях трудится множество дизайнеров, чьи работы обречены выпускаться под именем компании их работодателя. Наша платформа позволит талантливым авторам выкладывать свои работы под собственным именем и зарабатывать этим.

Данная платформа позволит неизвестным дизайнерам сделать себе имя и предоставит простым пользователям возможность приобретать вещи, недоступные в прочих магазинах.

Стоит отметить, что платформа является прослойкой между тремя сторонами: покупателем, дизайнером и исполнителем. Последний необходим для избавления авторов эскизов от ручного создания продукта. Это позволит дизайнерам и портным заниматься своим делом.

Пользователь регистрируется как покупатель, дизайнер или исполнитель:

- Покупатель может добавлять эскизы в избранное, а также заказывать их. При заказе покупатель может связаться с дизайнером и задать ему все необходимые вопросы.
- Дизайнер выкладывает свои эскизы на платформу. После поступления заказа от покупателя, дизайнер отвечает на вопросы, связанные с заказом. После оплаты покупателем дизайнер приступает к выполнению заказа. Он может самостоятельно изготовить товар, либо воспользоваться услугами исполнителя, найдя его через нашу платформу.
- Исполнитель может просмотреть список доступных заказов от дизайнеров, связаться с ними и договориться об условиях.

После выполнения заказа дизайнер или исполнитель отправляет готовое изделие покупателю.

Проделанная работа

На данный момент выбраны и разобраны технологии, используемые для создания приложения:

Язык программирования – Java.

Причинами такого решения являются надёжность и безопасность, предоставляемая языком Java. Это необходимо для нашего проекта, так как имеет место обработка приватных данных клиентов.

Другой причиной такого решения – поддержка данного языка, сильно облегчающая работу с ним.

Система управления базами данных – PostgreSQL, MongoDB и Redis.

Первая СУБД – реляционная. Она выбрана для того, чтобы быть использованной как в аутентификации и авторизации, так и при создании заказов. Причинами такого выбора являются: строгая схема данных, большое количество связей между хранимыми сущностями и необходимость использования транзакций. PostgreSQL выбрана в качестве конкретной СУБД так как она масштабируема, имеет множество встроенных функций и графических интерфейсов, облегчающих использование.

Вторая СУБД – документоориентированная. Она выбрана так как: данные слабо связны, меняющие свою структуру по мере развития проекта (например, добавляются новые поля), необходима высокая скорость обработки запросов, которую предоставляют NoSQL базы данных, потенциальная необходимость серьёзного масштабирования (NoSQL базы лучше подходят для хранения больших объёмов данных и их легче горизонтально масштабировать как раз за счёт слабой связности данных и денормализации[1]). Именно MongoDB выбрана, так как наиболее популярна и обладает расширенным функционалом по сравнению с аналогами[17].

Третья СУБД – in-memory[23] типа ключ-значение. Она служит для кэширования[2] на уровне приложений. Данная база данных находится между веб приложением и постоянным хранилищем данных.

Фреймворки – Spring, Swagger.

Spring – наиболее развитая и популярная экосистема для разработки веб приложений. Функционал позволяет создавать и настраивать практически все аспекты целевого приложения, в том числе работу с базами данных, защиту приложения, реактивное программирование и прочее.

[1] Денормализация - попытка увеличить скорость чтения данных за счёт уменьшения скорости записи. Избыточные копии данных записываются в несколько таблиц для избежания сложных операций соединения данных.

[2] Кэширование – метод разработки инфраструктуры системы, который улучшает время загрузки страницы и может уменьшить нагрузку на сервисы и базы данных. При таком подходе, диспетчер кэширования вначале проверяет, делался ли запрос ранее, чтобы найти ответ, который уже на него возвращался, сократив при этом время выполнения текущего запроса. Если такого запроса ранее не производилось, то полученный ответ сохраняется в кэш.

Swagger – популярный фреймворк, предоставляющий удобный графический интерфейс для как визуализации сгенерированной документации, так и для создания различных запросов к сервису.

Система сборки – Gradle.

Было решено использовать её из-за следующих преимуществ:

- позволяет выполнять инкрементные сборки;
- имеет возможность безопасного кеширования;
- позволяет определять пользовательские правила указания версии для динамической зависимости и разрешать конфликты версий;
- имеет полностью настраиваемую модель выполнения;
- имеет возможность использования пользовательских дистрибутивов;
- позволяет настраивать среды сборки на основе версий без необходимости настраивать их вручную.

Средства разработки – IntelliJ IDEA, PgAdmin 4, MongoDBCompass, Postman.

InteliJ IDEA – самый популярная среда разработки на Java.

PgAdmin 4 – инструмент для управления базами данных на движке PostgreSQL.

MongoDBCompass – инструмент для управления базами данных на движке Mongo.

Postman – инструмент для создания HTTP запросов.

Глава 1. Подходы к построению архитектуры системы

1.1 Необходимость правильной архитектуры

По данным исследования[38] американской компании Akamai[3]:

- 47% пользователей ожидают, что веб-страница загрузится в течение 2 секунд;
- 40% посетителей могут уйти с сайта, который грузится более 3 секунд;
- 52% утверждают, что быстрая загрузка влияет на их лояльность;
- секунды ожидания уменьшают лояльность клиентов примерно на 16%.

То, что скорость загрузки сайта влияет на поведение пользователей, подтверждается и в другом исследовании — The Gomez Peak Time Internet Usage Study[39], проведённом Equation Research[4]:

- в часы пик 75% посетителей ушли на сайты конкурентов, не дождавшись загрузки страницы;
- 88% заявили, что вряд ли вернутся на сайт после неудачной попытки его открыть;
- больше половины пользователей выразили менее позитивное мнение о компании в целом, если сайт медленно открывался;
- более трети поделились негативным впечатлением со знакомыми.

Именно из-за приведённых выше особенностей правильная архитектура системы крайне важная составляющая успеха проекта.

Скорость и надёжность приложения напрямую зависят от выбора стратегии обработки данных, оптимальности работы компонент системы, способа масштабирования и прочих факторов.

Так же стоит отметить, что хорошо продуманная архитектура при дальнейшем развитии проекта позволяет избежать многих проблем, связанных с потерей или нарушением целостности данных, использованием недостаточного или избыточного количества вычислительной мощности, сложностью сопровождения и т.п.

1.2 Основная функциональная составляющая

Заранее отметим, что разрабатываемый сервис построен по принципам REST[18]. Следовательно, сервис разбит на «клиентов» и «серверов». Клиенты — это как браузерные, так и мобильные приложения, описание которых выходит за рамки данной работы. Серверы

[3] Akamai - поставщик услуг для акселерации веб-сайтов, провайдер платформ доставки контента и приложений.

[4] Equation Research - агентство стратегических исследований полного цикла.

– приложения, представляющие собой лишь API[19] с которыми некоторым образом взаимодействуют клиенты.

Существует три основных подхода к построению архитектуры сервиса: монолитный, микросервисный и бессерверный.

Монолитная архитектура подразумевает наличие единого сервиса, в котором сконцентрирована вся бизнес-логика системы.

«Монолит» обладает следующими достоинствами:

- Монолитная архитектура гораздо проще в реализации, управлении и развёртывании.
- Транзакции и прочие межсервисные взаимодействия не вызывают сквозных проблем, так как всё сконцентрировано в одном коде внутри единого приложения.
- В монолитной архитектуре легче выполнять тестирование.

Естественно присутствуют и недостатки:

- Сложность разработки и поддержки из-за роста объёма кода.
- Риск потери работоспособности всего приложения при наличии ошибки в одном компоненте программы.
- Масштабирование становится проблематичным, когда одной из частей системы требуются дополнительные ресурсы.

При микросервисном подходе приложение представляет собой набор легковесных сервисов, обрабатывающих лишь часть функционала всего приложения и взаимодействующих между собой по сети.

Достоинства микросервисной архитектуры:

- Легкость масштабирования – на отдельны серверных узлах реплицируются модули по мере необходимости.
- Возможность по мере необходимости обновлять приложение по частям – при монолитной архитектуре приходится заново развёртывать приложение целиком, что влечёт за собой куда больше рисков.
- Микросервисы имеют большую доступность: даже если один из них сбоит, это не приводит к сбою всего приложения.
- Возможность использовать разные необходимые технологии внутри каждого модуля.

Недостатки:

- Сложность реализации транзакций.

- Сложное тестирование.
- Необходимость координирования команд, разрабатывающих отдельные микросервисы.
- Для разворачивания микросервисной системы требуется больше ресурсов чем для монолитной системы.

При микросервисном подходе компоненты системы могут взаимодействовать друг с другом с помощью обычных HTTP-запросов или брокеров сообщений, таких как RabbitMQ[35] или Apache Kafka[36]. Брокеры преобразуют сообщения от разных формальных протоколов обмена сообщениями, позволяя взаимозависимым службам напрямую «общаться» друг с другом, даже если они написаны на разных языках или работают на других платформах. Они валидируют, маршрутизируют, хранят и доставляют сообщения назначенным получателям. Брокеры действуют как посредники между приложениями, позволяя отправлять сообщения, не зная о количестве, активности и местонахождении адресатов.

Различия между RabbitMQ и Apache Kafka:

- RabbitMQ использует определенный ограниченный поток данных. Продюсер создает и отправляет сообщения, а консьюмер их принимает. Apache Kafka использует неограниченный поток данных, при этом пары «ключ-значение» непрерывно передаются в назначенную тему.
- RabbitMQ отлично подходит для запросов пользователей и транзакционных данных, таких как создание и размещение заказов. Kafka лучше справляется с операционными данными, такими как технологические процессы, статистика аудита и сбора данных, активность системы.
- RabbitMQ отправляет пользователям сообщения, которые удаляются из очереди после их обработки и подтверждения. В свою очередь Kafka это журнал. Он использует непрерывные цепочки сообщений, сохраняющиеся в очереди определённое время.
- RabbitMQ использует модель умный брокер/тупой консьюмер. Брокер последовательно доставляет сообщения консьюмерам и отслеживает их статус. Apache Kafka использует модель тупого брокера/умного консьюмера. Этот инструмент не отслеживает сообщения, которые прочитал каждый пользователь. Kafka запоминает только непрочитанные сообщения, сохраняя их в течение установленного периода времени. Консьюмеры должны самостоятельно следить за своей позицией в каждом журнале.

- RabbitMQ использует топологию обмена очереди: сообщения отправляются на обмен, откуда затем рассылаются в различные привязки очередей для использования консьюмерами. Kafka использует топологию Publish/Subscribe, отправляя сообщения через поток в соответствующие топики, которые затем потребляются пользователями в разных авторизованных группах.

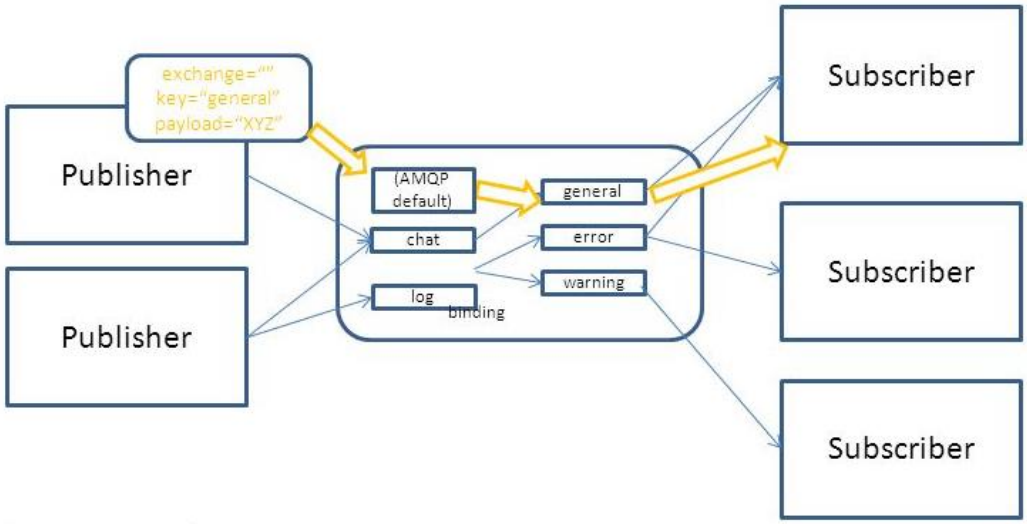

Рисунок 2. Внутреннее устройство RabbitMQ.

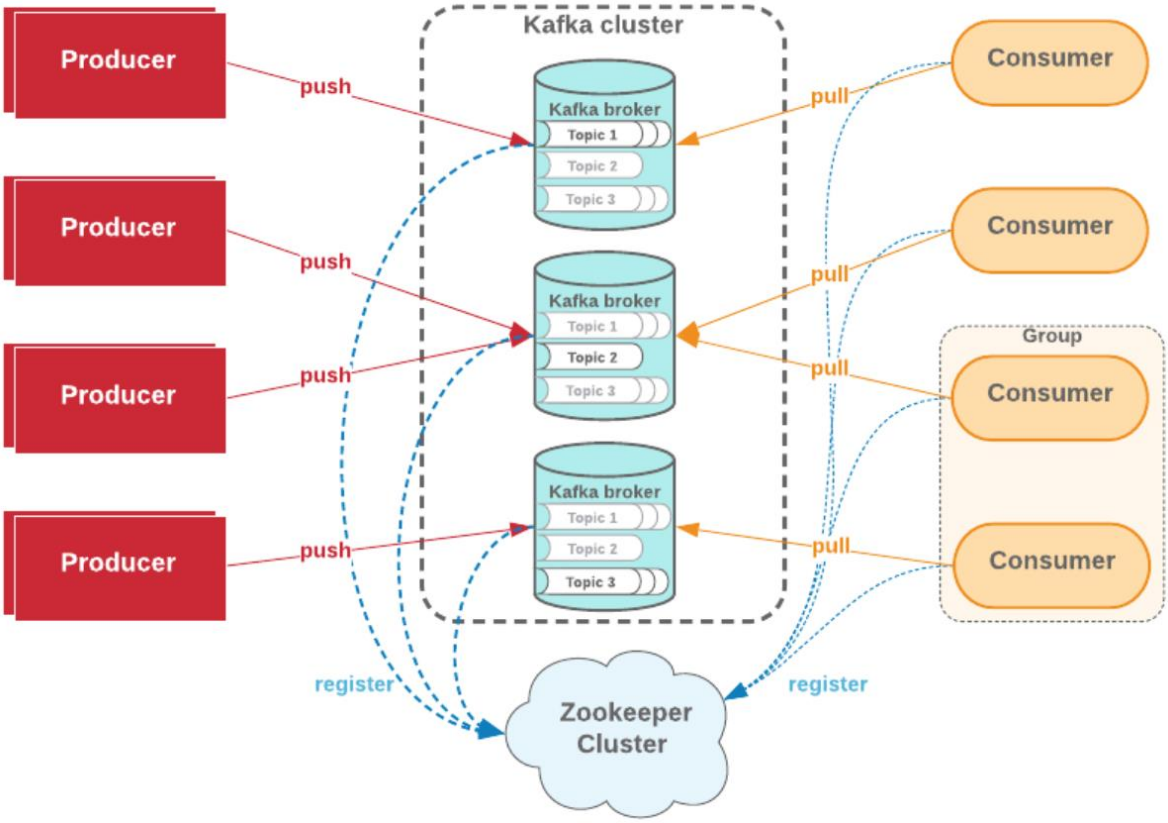

Рисунок 1. Внутреннее устройство Apache Kafka.

Последний из вышеперечисленных подход к построению архитектуры сервиса – бессерверный. Он позволяет создать и разворачивать приложение без необходимости

думать о базовой инфраструктуре, которая его поддерживает. В бессерверной модели вместо предварительной подготовки серверов для удовлетворения потребностей, нужно написать код (с учётом определённых требований) и отправить его на соответствующую платформу в облаке (например, AWS Lambda).

Главным достоинством является отсутствие необходимости думать о традиционном управлении сервером (виртуальными машинами, контейнерами или другой традиционной единицей инфраструктуры). Ответственность за масштабирование берёт на себя платформа провайдер услуг. Разумеется, такие услуги не бесплатны, но стоимость зависит лишь от нагрузки на приложение, в отличии от остальных подходов, где необходимо платить даже за время простоя.

Недостатки:

- Теряется контроль над реализацией кода и обработкой исходных данных (например, невозможно подключиться по SSH к инфраструктуре для ручной настройки).
- Существуют повышенные потенциальные риски для безопасности, ввиду зависимости от третьих лиц – провайдера бессерверных вычислительных мощностей.
- Трудоёмкость локального тестирования.

1.3 Базы данных

Так как количество пользователей может быть велико, то запросы к единственной базе данных будут выполняться достаточно долго из-за внутренних механизмов управления параллелизмом. Подобные задержки приведут к замедлению работы всей системы, а возможно и к прекращению её работоспособности. Для решения данных проблем применяются различные методы горизонтального масштабирования СУБД, а именно репликация (master-slave, master-master), шардинг[5] и федерализация[6]. Аналогично масштабированию макросервисов, масштабирование баз данных влечёт за собой увеличение количества серверов СУБД, выполняющих функции чтения и записи, что снижает среднюю нагрузку на каждый сервер за счёт равномерного распределения запросов между всеми серверами.

[5] Шардинг - это прием, который распределяет данные по разным базам данных таким образом, что каждая база данных может управлять только подмножеством данных.

[6] Федерация - разделение баз данных по функциям. Например, вместо одной монолитной базы данных, можно создать три отдельных базы данных: форум, пользователи и товары, что приведет к меньшему количеству операций чтения и записи в каждую базу данных.

Так как операций чтения в системах, подобных нашей, существенно больше операций записи, то целесообразно выделять для них отдельные серверы, содержащие так называемые реплики – копии основной СУБД с различной степенью актуальности хранимых данных. Важно упомянуть, что информация на репликах должна обновляться информацией с сервера основной базы по мере возможности (частота обновлений зависит от общей нагрузки системы). Такие реплики служат не только для ускорения работы операций чтения, но и для хранения резервных копий – при падении основной СУБД её обязанности берет на себя одна из реплик. При возникновении непредвиденных обстоятельств эта стратегия позволяет терять лишь небольшое количество самых последних данных, а не все.

В свою очередь, сервер основной базы данных так же не обязан быть лишь в единственном числе – федерализация и шардинг распределяют актуальные данные в соответствующие СУБД. Это позволит снизить среднюю нагрузку на каждую базу (естественно, если шардинг и(или) федерализация выполнены оптимальным образом).

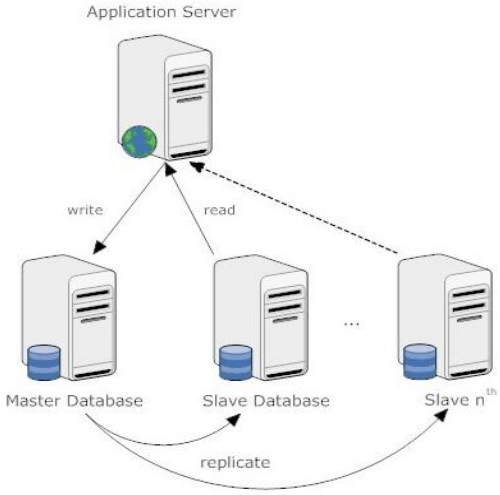

Рисунок 3. Репликация базы данных.

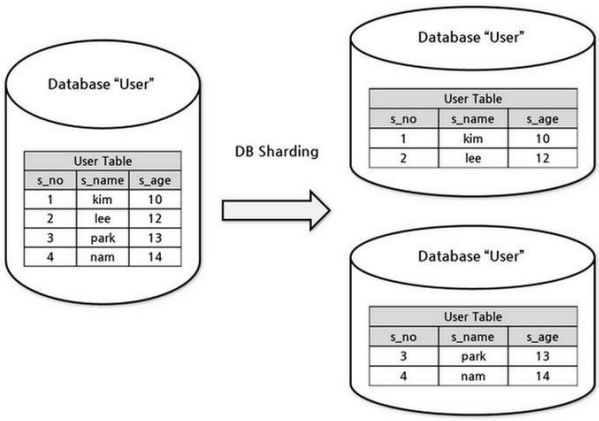

Рисунок 4. Шардинг базы данных.

1.4 Сопутствующая инфраструктура

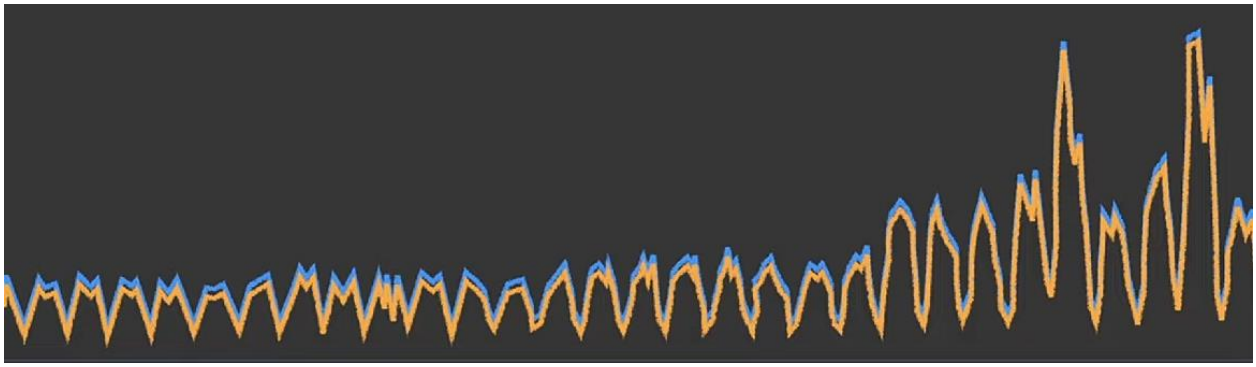

Рисунок 3. Ноябрьский трафик сайта Amazon. Синим цветов - максимальная допустимая нагрузка на систему, желтым - действительная нагрузка [1].

Наше приложение будет использовать серверный способ разворачивания – сервисы запускаются в контейнерах[5], таких как OpenVZ[41], LXC[42], Docker[6] и прочих. Они находятся под управлением системы оркестрации, такой как Docker Swarm[7], Kubernetes[8], Nomad[9] или другой. При этом данные приложения будут храниться в облачных хранилищах данных одного или нескольких провайдеров: Яндекс[10], Google[11], Amazon[12], Microsoft[13] или других.

Ввиду нестабильности нагрузки, вычислительные мощности для постоянного обслуживания миллиона пользователей будут простаивать (возможно, большую часть времени). Для решения подобной проблемы необходимо использовать автоматическое масштабирование. Оно позволяет динамически выделять оптимальное количество ресурсов, соответствующее текущей нагрузке на приложение.

В данном случае, масштабирование именно горизонтальное, то есть необходимая пропускная способность достигается за счёт количественного, а не качественного изменения потребляемых ресурсов – в системе работает лишь оптимальное количество контейнеров. При этом машины, на которых запущены контейнеры не меняются – их вычислительные мощности остаются прежними.

При увеличении контейнеров в инфраструктуре системы, появляется необходимость использовать балансировщики нагрузки и обратные прокси:

- Балансировщики нагрузки распределяют входящие клиентские запросы между контейнерами с приложениями или базами данных, возвращая ответ от конкретного контейнера клиенту, от которого пришел запрос. Они используются для предотвращения запроса на неработающий сервер, предотвращения чрезмерной нагрузки ресурсов, избежания появления единой точки отказа, SSL-терминации[7] и

[7] SSL-терминация - расшифровка входящих запросов и шифровка ответов. При использовании SSL-терминации бэкенд-сервера не тратят свои ресурсы на эти потенциально трудоемкие операции.

сохранения сессий. Балансировщики могут быть аппаратными или программными. Для защиты от сбоев, можно использовать вместе несколько балансировщиков в режимах «Активный-Пассивный»[8] или «Активный-Активный»[9]. Балансировщики распределяют нагрузку различными образами:

- случайно;
- наименее загруженные сервер;
- сессия/куки;
- взвешенный циклический;
- Layer 4[15];
- Layer 7[16].

- Обратный прокси – веб-сервер, который централизует внутренние сервисы и предоставляет для них унифицированный интерфейс для доступа из публичной сети. Клиентские запросы перенаправляются на сервер, который их будет обрабатывать, и затем обратный прокси возвращает ответ клиенту. Дополнительные преимущества, по сравнению с описанными выше балансировщиками нагрузки:

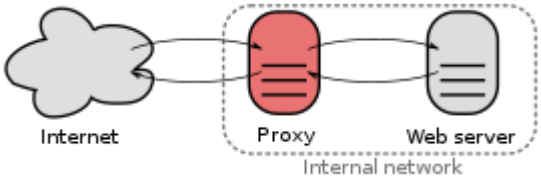

Рисунок 4. Обратный прокси.

  - скрывает информацию о бэкенд-серверах, блокирует IP адреса, ограничивает допустимое количество соединений на клиента;
  - клиентское приложение знает только IP адрес прокси-сервера, таким образом можно менять количество серверов или изменять их конфигурацию;
  - сжатие ответов сервера клиенту;
  - возвращает ответы для закэшированных запросов;
  - раздаёт статику напрямую:
    - HTML/CSS/JS;
    - Фотографии
    - Видео
    - и т.д.

Выбор между этими двумя технологиями зависит от особенностей проектируемой системы, но стоит помнить, что:

[8] В таком режиме, активный и пассивный сервер, находящийся в режиме ожидания, обмениваются специальными сообщениями - heartbeats. Если такое сообщение не приходит, то пассивный сервер получает IP адрес активного сервера и восстанавливает его работу.

[9] В таком режиме оба сервера обрабатывают клиентские запросы, распределяя нагрузку между собой.

- Обратный прокси-сервер может быть полезен даже при использовании одного контейнера с приложением, так как предоставляет вышеописанные возможности.
- Использование обратного прокси-сервера увеличивает сложность системы в целом.
- Использование одного прокси-сервера создает единую точку отказа. Настройка нескольких обратных прокси-серверов еще больше усложняет систему.

Глава 2. Реализация архитектуры системы

2.1 Микросервисы

На старте проекта количество пользователей довольно мало – не более тысячи. Для такой нагрузки подходит следующий подход: монолитное приложение с единой реляционной базой данных, развернутое на не самой производительной машине (вплоть до персонального компьютера).

Мы не будем использовать этот подход, а постараемся продумать архитектуру, подходящую для работы с более высокой нагрузкой – около миллиона постоянных пользователей. Для достижения этой цели, сервис должен быть именно распределённой системой, способной масштабироваться. Именно поэтому мы делаем выбор в пользу микросервисной архитектуры.

Опишем микросервисы приложения, их структуру и способы взаимодействия.

На данном этапе развития проекта выделены следующие основные сервисы:

Сервис авторизации и аутентификации

Отвечает за управлением пользователями и их персональными данными внутри системы, а именно:

- Регистрация
- Логин
- Обновление персональной информации: никнейм в системе, имя, фамилия, адреса, аватары, описание профиля, пароль
- Удаление аккаунта
- Работа с токенами – их создание, обновление и удаление.

Данный сервис работает с СУБД Postgres ввиду реляционности данных.

При регистрации нового пользователя создаётся запись в базе данных. Затем, после успешного логина генерируются и возвращаются клиенту access токен и refresh токен. Первый представляет собой JWT[20] токен, хранящий внутри себя в закодированном виде адрес электронной почты, никнейм и роли пользователя. Второй – уникальная UUID[21] строка. Access токен не сохраняется в базу, в отличии от refresh токена – это необходимого для отслеживания прав доступа refresh токена и его валидации. Для своевременного обновления access токена, имеющего заранее определённый срок годности, пользователи посылают на соответствующий адресс запрос, содержащий refresh токен и получают новый access токен. Таким образом логин происходит на данном сервисе по адресу электронной почты и паролю, а авторизация и аутентификация во всех остальных сервисах происходит

по access токену, получаемому от пользователя при каждом запросе из хэдера «Authorization»[22] – проверяется достоверность цифровой подписи JWT токена и его срок годности. Если токен валидный, то пользователю предоставляются соответствующие ему права.

Система поддерживает авторизацию через сторонние сервисы с помощью OAuth 2.0, например, через Google. В таком случае клиенту так же возвращаются токены, но пользовательские данные берутся со стороннего сервиса.

Сервис контента

Отвечает за CRUD[10] операции с контентом, которым являются созданные пользователями эскизы. Поиск осуществляется по многим свойствам данных. Реализованы различные типы видимости контента – публичный, приватный и по сгенерированной ссылке.

Данный сервис хранит данные в СУБД MongoDB, так как данные нереляционные, транзакции отсутствуют, а скорость обработки (получение данных в особенности) должна быть максимальной.

Сервис метаданных

Отвечает за CRUD операции с метаданными – оценками, комментариями, отзывами, корзиной покупок и пр. Несмотря на то, что данные реляционные, они хранятся в СУБД MongoDB, так как скорость работы имеет наивысший приоритет.

Сервис чатов

Отвечает за обработку чатов – возвращает истории сообщений, открывает сокеты для обмена сообщениями в реальном времени и отправляет уведомления о сообщениях по необходимости.

Из-за необходимости минимальных задержек, история сообщений сохраняется в MongoDB.

Сервис оплаты

Отвечает за оплату, создание заказов и бронирование товаров.

Данные хранятся в СУБД Postgres ввиду потребности в транзакциях.

Сервис администрирования

Отвечает за контроль за всеми данными, хранимыми в системе. Реализованы CRUD операции над всеми используемыми СУБД.

[10] CRUD - Create, Read, Update, Delete

Сервис фоновых задач

Отвечает за задачи, не требующие моментального отклика: рассылка электронной почты, подтверждения регистрации и прочее. Использует все СУБД системы.

2.2 Базы данных

Структура СУБД PostgreSQL

Таблицы базы для сервиса авторизации:

- User – представляет сущность пользователя. Хранит данные, необходимые для авторизации и аутентификации, создания заказа и создания эскизов (по необходимости).
- Role – представляет сущность роли.
- Token – сущность для хранения refresh токенов. Хранит значение, владельца и срок годности.

Таблицы базы для сервиса оплаты:

- Order – представляет сущность заказа. Хранит время, сумму, эскиз, адрес доставки, покупателя, создателя, исполнителя и некоторую метаинформацию.
- Reservation – представляет сущность брони товара. Хранит эскиз, заказчика, дату и срок годности брони.

Структура СУБД MongoDB

Документ базы для контента:

- Sketch – представляет сущность эскиза. Хранит название, дату публикации, цену, названия вложенных файлов (сами файлы хранятся в распределённой файловой системе) и прочую информацию о эскизе.

Документ базы для метаинформации:

- Like – хранит информацию о том, каким пользователям какие скетчи понравились.
- Comment – хранит информацию о том, какие комментарии под какими скетчами оставили пользователи.
- Review – хранит информацию о том, какие отзывы каким эскизам оставляют пользователи.

2.3 Инфраструктура

В качестве стандартной технологии контейнеризации используется Docker[Ошибка! Источник ссылки не найден.], являющийся де-факто стандартным инструментом из-за следующих достоинств:

- Благодаря Docker можно легко запускать контейнер в облачной инфраструктуре и на любом локальном устройстве. Можно создать базовые шаблоны контейнеров и использовать повторно бесконечное число раз. Бесшовная переносимость и простота развертывания — важные преимущества этой технологии.
- В контейнерах Docker содержится всё, что требуется для запуска приложения, поэтому перенос приложений из одной среды в другую не вызывает затруднений. Исчезает проблема, когда у разработчиков всё функционирует как надо, а на боевом сервере — нет.
- Так как настраивать окружение для разработки, тестирования и боевого режима больше не нужно, время развертывания сокращается в несколько раз.
- Docker позволяет использовать любые языки программирования и стек технологий на сервере, избавляя от проблемы несовместимости разных библиотек и технологий.
- Существует огромная библиотека контейнеров с открытым исходным кодом. Можно скачать нужный образ для конкретной задачи или обратиться за помощью к большому комьюнити разработчиков, которые используют Docker.
- С учетом инструментов управления трафиком можно построить процесс обновления приложения так, чтобы обновление одних контейнеров не влияло на работоспособность системы и оказание услуг пользователям.
- С помощью Docker легче перенести контейнер с одного хоста на другой, запустить сразу несколько образов, обновить группы контейнеров и откатиться к старой версии.
- Контейнеры в Docker частично изолированы друг от друга на уровне процессов и ОС, поэтому запуск большого количества контейнеров на одной машине не несет рисков.
- Контейнеры легковесны и производительны, а благодаря использованию Docker можно эффективнее управлять имеющимися ресурсами и сократить расходы на хостинг.

- Отказ от монолитной архитектуры в пользу микросервисной позволяет более гибко развивать продукт, добавлять в него новые функции.

Для автоматического масштабирования контейнеры с работающими микросервисами реплицируются. Некоторые из них можно впоследствии разделить на сервисы чтения и сервисы записи (синхронные и асинхронные), запущенные в раздельных контейнерах. В таком случае имеет смысл создать очередь для асинхронного исполнения запросов на запись. Она будет наполняться по мере поступления таких запросов и опустошаться по мере их выполнения сервисами.

В качестве брокера сообщений выбран RabbitMQ, а не его аналог Apache Kafka. Такой выбор обусловлен особенностями нашего приложения:

- нет необходимости использовать бесконечные потоки ввиду непостоянства потока сообщений;
- транзакционные данные в сообщениях
- нет необходимости в длительном хранении сообщений в журнале;
- модель умный брокер/тупой консьюмер позволяет отслеживать исполнение сервисом задачи при получении сообщения;
- топология очереди способна эффективно обрабатывать поток сообщений системы.

Используется кэширование с помощью in-memory базы данных Redis для ускорения сервисов – запрос пользователя сначала уходит в Redis и проверяет наличие значения по соответствующему ключу. Если оно есть, то пользователю будет дан ответ из Redis, а не напрямую из основной базы данных. Иначе БД сообщит Redis запомнить связку ключ-значение для этого запроса.

Сервисы баз данных и кэширования также горизонтально масштабируются. Redis использует асинхронную master-slave[11] репликацию[24]. В это же время master-slave репликации используется и для масштабирования баз данных MongoDB и PostgeSQL. Вдобавок для этих СУБД можно применять шардинг. В MongoDB шардинг встроенный, а в PostgreSQL придётся либо реализовывать его вручную на уровне приложения, либо использовать расширение, например, Citus[26].

[11] Ведущий сервер работает на чтение и запись, реплицируя записи на один или более ведомых серверов. Ведомый сервер работает только на чтение. Ведомые сервера могу реплицировать на дополнительные ведомые сервера (как в древовидной структуре). Если ведущий сервер перестает работать, система продолжает работать в режиме только на чтение до тех пор, пока один из ведомых серверов не станет ведущим, или пока новый ведущий сервер не будет создан.

Вся инфраструктура приложения управляется системой оркестрации контейнеризованных приложений Kubernetes. Она предоставляет инструменты для развертывания, управления, распределения нагрузки, масштабирования (в том числе и автоматического) и прочего. Данная система выбрана ввиду её широчайших возможностей и популярности – в текущий момент она становится де-факто стандартом развертывания приложений и поддерживается наиболее популярными провайдерами облачных услуг, такими как GCP[27], AWS[28], Microsoft Azure[29], YCP[30]. Благодаря стандартному интерфейсу Kubernetes, можно абстрагироваться от выбора конкретного хостингового сервиса – адаптация для развертывания на другом облаке не требует масштабных изменений скриптов развертывания.

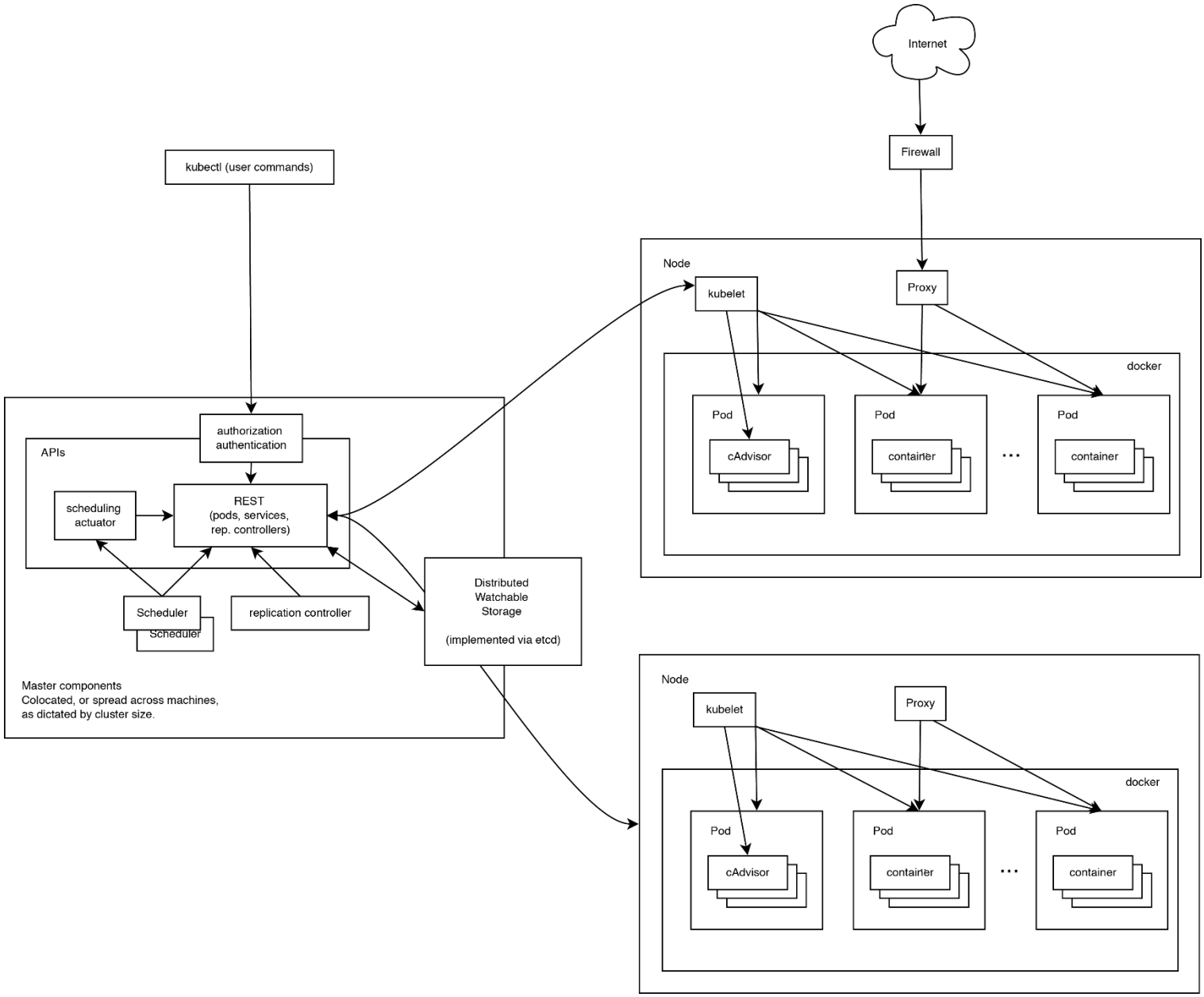

Рисунок 5. Внутреннее устройства кластера Kubernetes.

В качестве балансировщика нагрузки можно использовать встроенные возможности Kubernetes. В таком случае применяется циклическая[31] layer 4[15] балансировка. Для layer 7[16] балансировки необходимо применять сторонние инструменты, например, Nginx[32] – Nginx Plus интегрируется с Kubernetes[33], позволяя кастомизировать распределение нагрузки. Так же балансировка нагрузки происходит на уровне DNS[34] – при запросе пользователи получают IP-адрес географически ближайшего сервера. Это необходимо для минимизации транспортных задержек, появляющихся из-за больших расстояний между клиентом и сервером.

Используются CDN[12] для ускорения раздачи статического и динамического контента. При получении клиентом ответа, контент может быть загружен не с сервера приложения, а из CDN. Стратегия получения контента самой CDN – pull. Такие CDN загружают новое содержимое при первом обращении пользователя. Содержимое остаётся на сервере, но обновляются адреса, чтобы они указывали на CDN. В результате, первый запрос обрабатывается медленнее, ожидая пока содержимое будет закэшировано в CDN, но последующие запросы получают контент быстрее.

[12] 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.

ЗАКЛЮЧЕНИЕ

Мы описали архитектуру сервиса, обрабатывающего миллион постоянных пользователей.

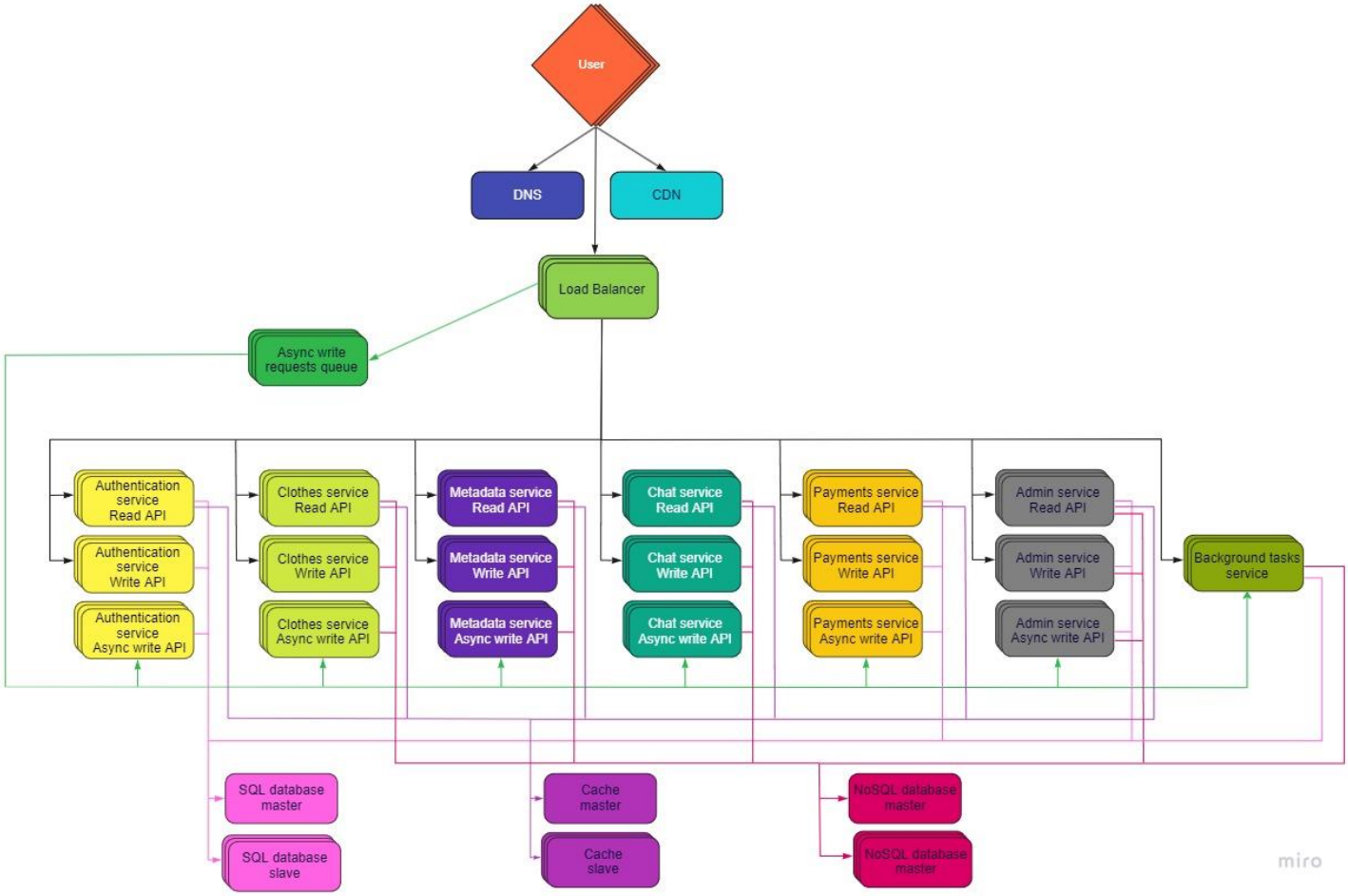

Рисунок 6. Схематичное изображение архитектуры разрабатываемого приложения.

Не стоит считать описанный выше инфраструктуру приложения единственной возможной или единственной правильной. Многие элементы могут изменяться со временем ввиду аспектов дальнейшего развития системы. Так же необходимость в изменениях архитектуры может появляться в следствии мониторинга работоспособности приложения или предпочтений пользователей по расширению функционала.

Следующим шагом является реализация системы, основывающейся на построенной архитектуре, а также исследование её эффективности в контексте поставленной задачи и используемого аппаратного обеспечения.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Официальная конференция Amazon – AWS re:Invent – https://www.youtube.com/watch?v=kKjm4ehYiMs
2. OpenSource учебник по системному дизайну – https://github.com/donnemartin/system-design-primer#denormalization
3. CAP теорема – https://en.wikipedia.org/wiki/CAP_theorem
4. Микросервисная архитектура – https://habr.com/ru/company/vk/blog/320962/
5. Развитие способов развёртывания приложений – https://kubernetes.io/docs/concepts/overview/#going-back-in-time
6. Docker – https://docs.docker.com
7. Docker Swarm – https://docs.docker.com/engine/swarm/
8. Kubernetes – https://kubernetes.io
9. Nomad – https://www.nomadproject.io
10. Yandex.Cloud Object Storage – https://cloud.yandex.ru/services/storage
11. Google Cloud Storage – https://cloud.google.com/storage
12. AWS Cloud Storage – https://aws.amazon.com/ru/products/storage/
13. Microsoft Azure Cloud Storage – https://azure.microsoft.com/en-us/products/category/storage
14. AWS Lambda – https://aws.amazon.com/ru/lambda/
15. Layer 4 балансировка нагрузки – https://www.nginx.com/resources/glossary/layer-4-load-balancing/
16. Layer 7 балансировка нагрузки – https://www.nginx.com/resources/glossary/layer-7-load-balancing/
17. MongoDB vs CouchDB – https://www.mongodb.com/compare/couchdb-vs-mongodb
18. Принципы REST – https://habr.com/ru/post/590679/
19. API – https://habr.com/ru/post/464261/
20. JWT – https://jwt.io
21. UUID – https://ru.wikipedia.org/wiki/UUID
22. HTTP – https://ru.wikipedia.org/wiki/HTTP
23. In-memory СУБД – https://en.wikipedia.org/wiki/In-memory_database
24. Репликация Redis – https://redis.io/docs/management/replication/
25. Автоматическое масштабирование СУБД – https://habr.com/ru/company/oleg-bunin/blog/666856/
26. Citus PostgreSQL – https://www.citusdata.com

27. Kubernetes в GCP – https://cloud.google.com/kubernetes-engine
28. Kubernetes a AWS – https://aws.amazon.com/ru/eks/
29. Kubernetes в Microsoft Azure – https://azure.microsoft.com/ru-ru/products/kubernetes-service/
30. Kubernetes в Яндекс.Облако – https://cloud.yandex.ru/services/managed-kubernetes
31. Round-robin балансировка нагрузки – https://en.wikipedia.org/wiki/Load_balancing_(computing)#Round-robin_scheduling
32. Nginx – https://www.nginx.com
33. Интеграция Nginx с Kubernetes – https://www.nginx.com/blog/load-balancing-kubernetes-services-nginx-plus/
34. Domain Name System – https://ru.wikipedia.org/wiki/DNS
35. RabbitMQ – https://www.rabbitmq.com
36. Apache Kafka – https://kafka.apache.org
37. Основы Kubernetes – https://habr.com/ru/post/258443/
38. Исследование Akamai – http://www.akamai.com/html/about/press/releases/2009/press_091409.html
39. The Gomez Peak Time Internet Usage Study – http://www.mcrinc.com/Documents/Newsletters/201110_why_web_performance_matters.pdf
40. Как устроен Docker и почему он популярен – https://cloud.yandex.ru/blog/posts/2022/03/docker-containers
41. OpenVZ – https://openvz.org
42. LXC – https://linuxcontainers.org/lxc/
43. Книга Мартина Клеппмана «Высоконагруженные приложения. Программирование, масштабирование, поддержка» – https://www.rulit.me/data/programs/resources/pdf/Vysokonagruzhennye-prilozheniya-Programmirovanie-masshtabirovanie-podderzhka_RuLit_Me_648844.pdf