МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра Автоматики
(полное название кафедры)

Утверждаю

Зав. кафедрой Жмудь В. А.,
д.т.н. доцент
(подпись, инициалы, фамилия)

«___» _____ 2019 г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Кудряш Игорь Игоревич
(фамилия, имя, отчество студента – автора работы)

Проектирование и разработка серверной части проекта «*MemeBattle*»
(тема работы)

Факультет автоматики и вычислительной техники
(полное название факультета)

Направление подготовки 09.03.01. «Информатика и вычислительная техника»
(код и наименование направления подготовки бакалавра)

Руководитель

от НГТУ

Басыня Е.А.
(фамилия, имя, отчество)
к.т.н., доцент
(ученая степень, ученое звание)

Автор выпускной квалификационной работы

Кудряш И.И.
(фамилия, имя, отчество)
АВТФ, АВТ-518
(факультет, группа)

Новосибирск 2019 г.

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра Автоматики
(полное название кафедры)

УТВЕРЖДАЮ

ЗАВ. КАФЕДРОЙ

Жмудь В. А.

(фамилия, имя, отчество)

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

студенту Кудряшу Игорю Игоревичу
(фамилия, имя, отчество)

Направление подготовки 09.03.01. «Информатика и вычислительная техника»
(код и наименование направления подготовки бакалавра)

Факультет автоматики и вычислительной техники
(полное название факультета)

Тема Проектирование и разработка серверной части проекта «*MemeBattle»*
(полное название темы выпускной квалификационной работы бакалавра)

Исходные данные (или цель работы) Разработка и проектирования модуля для веб-проекта проекта «MemeBattle»

Структурные части работы
Реферат
Введение
Проектирование
Анализ технологических решений
Организация технологического конвейера
Программная реализация

Тестирование

Заключение

Задание согласовано и принято к исполнению.

Руководитель

от НГТУ

Басыня Е.А.
(фамилия, имя, отчество)

к.т.н., доцент
(ученая степень, ученое звание)

Студент

Кудряш И.И.
(фамилия, имя, отчество)

АВТФ, АВТ-518
(факультет, группа)

Тема утверждена приказом по НГТУ № 1389/2 от «18» марта 2019 г.

изменена приказом по НГТУ № _____ от «____» _____ 2019 г.

ВКР сдана в ГЭК № _____, тема сверена с данными приказа

_____
(подпись секретаря государственной экзаменационной комиссии по защите ВКР, дата)

_____
(фамилия, имя, отчество секретаря государственной экзаменационной комиссии по защите ВКР)

РЕФЕРАТ

Пояснительная записка выпускной квалификационной работы бакалавра Кудряш И.И. «Проектирование и разработка серверной части проекта *MemeBattle*» представлена на 75 страницах. Состоит из введения, заключения, 9 глав и списка используемых источников. Содержит: 11 рисунков, 3 таблиц, и 11 источников.

Ключевые слова: МИКРОСЕРВИС, СЕРВЕР, ВЕБСОКЕТЫ, REST API, ПОЛНОТЕКСТНЫЙ ПОИСК, ХРАНИЛИЩА ДАННЫХ.

Целью данной работы является разработка и реализация программного комплекса, способного выступать в качестве системы обработки данных.

В ходе работы был проведён сравнительный анализ существующих готовых реализаций устройств схожего функционала, с целью выявления их преимуществ и недостатков. На основании результатов сравнения и выдвинутого списка требований, на базе подобранных аппаратных решений и программной части, было разработано и сконфигурировано объединение сервисов. Для упрощения и автоматизации интеграции разработанного решения, была проведена работа по настройке системы доставки и развертывания.

Результатом работы стала совокупность микросервисов. Программная часть итоговой реализации может быть централизовано развёрнута на любом сервере в автоматизированном режиме, с помощью программных средств.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Цель данной дипломной работы – проектирование и разработка программной реализации серверной части микросервисов для медиа-платформы «*MemeBattle*».

Исходя из цели, в дипломной работе поставлены и решены следующие задачи:

- проведение анализа технических возможностей и характеристик различных микросервисов
- проведение анализа и сравнение с существующими аналогами
- выделение проблем в разработке серверной части приложения
- разработка технического решения для эффективной разработки информационной системы
- проведение сравнительного анализа различных серверных архитектур

ИССЛЕДОВАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ

Исследование предметной области в интересах последующего проектирование информационной системы является задачей, формирующей взгляд на данные, которые в предметной области используются и обрабатываются. Поэтому следует ответить на несколько вопросов:

1. Что такое веб-сервис ?
2. Что такое *backend* ?
3. Что такое *HTTP/HTTPS*?
4. Что такое *API* ?

Веб-сервис - услуга, предлагаемая электронным устройством другому электронному устройству, связывающемуся друг с другом через всемирную сеть.

В веб-сервисе веб-технология, такая как *HTTP*, изначально разработанная для связи между людьми, используется для связи между компьютерами, в частности, для передачи машиночитаемых форматов файлов, таких как *XML* и *JSON*.

На практике веб-служба обычно предоставляет объектно-ориентированный веб-интерфейс для сервера базы данных, используемый, например, другим веб-сервером или мобильным приложением, которое предоставляет пользовательский интерфейс для конечного пользователя. Многие информационные системы, которые предоставляют данные на отформатированных *HTML*-страницах, также предоставляют эти данные на своем сервере в виде *XML* или *JSON*. Другое приложение, предлагаемое конечному пользователю, может представлять собой гибридное приложение, где веб-сервер использует несколько веб-служб на разных компьютерах и собирает контент в один пользовательский интерфейс.

Backend - это все, что происходит на стороне сервера, то есть не в браузере. *Backend* может быть таким же простым, как просто хранение статического контента - контента, который никогда не меняется в зависимости от пользовательских действий.

Лучший способ подумать о разделении труда между браузером и серверной частью заключается в следующем: браузер отвечает за представление данных, отправленных из серверной части, визуально привлекательным способом, но данные происходят из удаленного источника.

Например: человек один человек находится в Москве, а второй в Новосибирске. Этот ответ хранится в Санкт-Петербурге, но контролируется рядом программ, написанных и управляемых программистами. Очевидно, что данные могут жить только в одном месте. В противном случае все видели бы несогласованные копии данных и если бы кто-либо обновил свой ответ, обновление никогда бы не дошло до подписчиков на эту информацию.

Некоторая обработка может происходить на внешнем интерфейсе (в браузере), но большая часть тяжелой работы происходит на сервере, включая все, что необходимо централизовать по соображениям безопасности или для обеспечения многопользовательского доступа.

HTTP (Hypertext Transfer Protocol) **-** это набор правил для передачи файлов (текстовых, графических изображений, звука, видео и других мультимедийных файлов) в *World Wide Web*. Как только веб-пользователь открывает свой веб-браузер, он косвенно использует *HTTP*. *HTTP* - это прикладной протокол, который работает поверх набора протоколов *TCP / IP* (базовых протоколов для Интернета). Понятия *HTTP* включают (как подразумевает гипертекстовая часть имени) идею о том, что файлы могут содержать ссылки на другие файлы, выбор которых вызовет дополнительные запросы на передачу. Любой компьютер веб-сервера содержит, помимо файлов веб-страниц, которые он может обслуживать, демон *HTTP*, программу, предназначенную для ожидания *HTTP*-запросов и их обработки по мере их поступления. Веб-браузер является *HTTP*-клиентом, отправляющим запросы на серверные машины. Когда пользователь браузера вводит файловые запросы, «открывая» веб-файл (вводя унифицированный указатель ресурса или *URL*-адрес) или щелкая по гипертекстовой ссылке, браузер создает *HTTP*-запрос и отправляет его по адресу интернет-протокола (*IP*-адресу) указанного в *URL*. Демон *HTTP* на компьютере сервера назначения получает запрос и отправляет обратно запрошенный файл или файлы, связанные с запросом. На данный момент существует 2 версии протокола: *http*/1.1, *http*/2. А также в разработке находится 3 версия протокола.

***API* (*Application Programming Interface*) -** термин является аббревиатурой и означает «Интерфейс прикладного программирования».

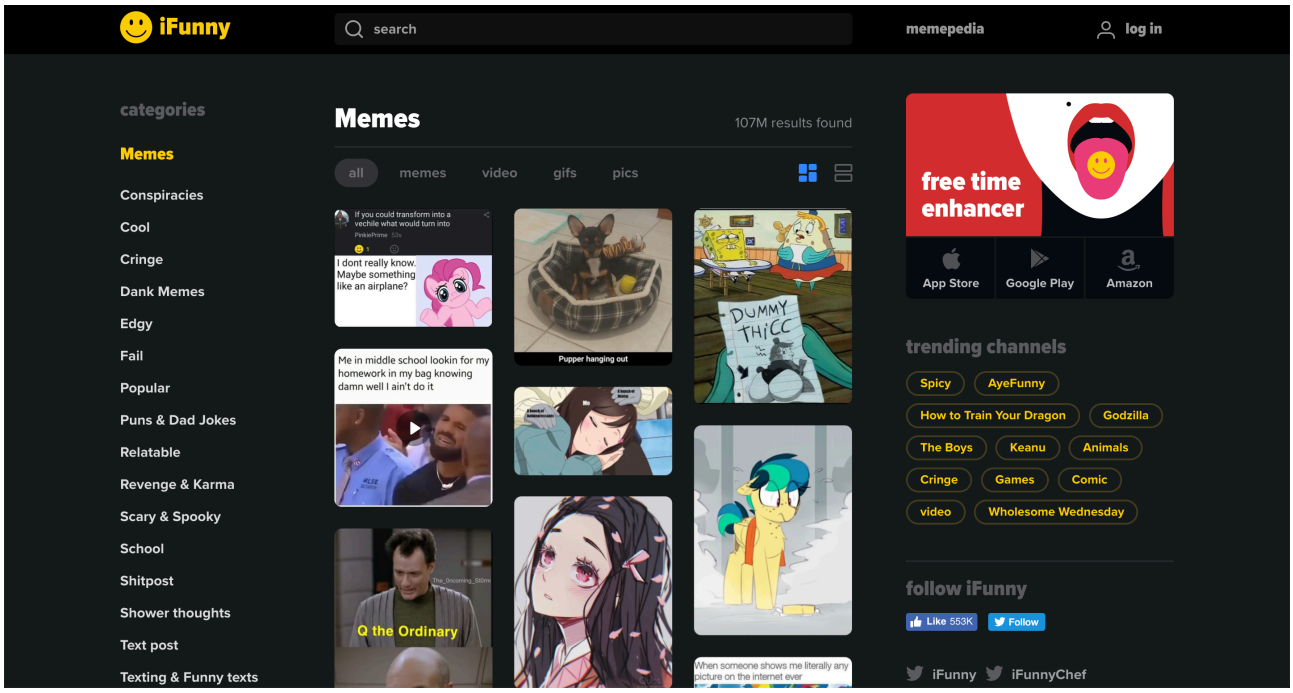
API – это как о «меню» для клиентов информационной системы.

API перечисляет множество операций, которые разработчики могут использовать, вместе с описанием того, что они делают. Разработчику не обязательно знать, как, например, операционная система создает и представляет диалоговое окно «Сохранить как». Ему просто нужно знать, что это доступно для использования в его приложении.

API-интерфейсы позволяют разработчикам экономить время, используя преимущества реализации платформы для выполнения самых сложных задач. Это помогает уменьшить объем кода, который необходимо создавать разработчикам, а также помогает повысить согласованность приложений в одной и той же платформе и увеличить консистентность данных. *API* могут контролировать доступ к аппаратным и программным ресурсам.

АНАЛИЗ ГОТОВЫХ РЕШЕНИЙ

В данной сфере очень мало похожих информационных систем. Но самая близкая к тематике – продукт компании «*FunCorp*». Главным отличием этой развлекательной платформы является отсутствие игровой направленности. Основные действия, совершаемые на платформе: просматривание различных картинок и возможность делиться этими данными в социальных сетях, а также добавлять свои личные в общую галерею пользователей.



ПРОЕКТИРОВАНИЕ

Чтобы спроектировать серверную часть, необходимо начать с выделения основных модулей системы и изучения основных принципов проектирования, а также стоит учитывать недостатки и плюсы конкурентных информационных систем.

На данный момент существует несколько архитектур, которые применяются для разработки *backend* части приложения:

- монолитная архитектура
- микросервисная архитектура

Монолитное приложение

Система состоит из нескольких слоев. Самый верхний слой приложения отвечает за прием запроса от клиента и зачастую производит проверку этого запроса на валидность передаваемых данных, а так же передает информацию на следующий слой приложения. Такой слой называют «контроллером» приложения.

Следующий слой монолитной архитектуры - «сервисный». Слой включает в себя различные адаптеры, для работы со сторонними сервисами, базой данных, системой кэширования и так далее, а так же «сервисный» слой проводит необходимые манипуляции с информацией, переданной из контроллера и возвращает результат своих действий обратно.

Рассмотрим процесс, загрузки системы на рабочие сервера. Данный вид приложения загружается одинм целым. Например: исходный код любого известного фреймворка.

Плюсы:

- простая разработка системы
- удобная загрузка системы на сервера
- простота в администрировании
- простое тестирование
- легкое масштабирование приложения

Минусы:

- привязка к языку разработки
- проблемы в разработке большого приложения

- загрузка на сервер как единое целое
- скорость разработки обратно пропорциональна масштабу системы
- ресурсоемкость
- большая связанность компонентов системы

Микросервисная архитектура

Архитектура представляет собой распределенную систему, состоящую из множества компонентов, отвечающих за узконаправленную логику всего приложения. Часто входной слой системы - балансировщик нагрузки, который распределяет входящие соединения по нескольким идентичным серверам. На входном сервере хранится логика, отвечающая за проверку информации в запросе, аналогично слою «контроллер» в монолитной архитектуре, а также за передачу данной информации в другие компоненты системы - микросервисы. Часто взаимодействие входного сервера и микросервисов происходит на прикладном уровне сети, по протоколу *http/https*.

Плюсы:

- легкая масштабируемость
- простота в разработке
- отсутствие привязанности к языку разработки
- простота в администрировании
- скорость разработки никак не зависит от объема всей системы
- легкая загрузка любого компонента системы на рабочий сервер
- отсутствие привязанности микросервиса ко всей системе

Минусы:

- проблема распределенных вычислений
- ресурсоемкие процессы взаимодействия между микросервисами
- распределенные транзакции
- отказоустойчивость всей системы, как единой целое

Так же необходимо рассмотреть управление компонентами в микросервисной архитектуре. При разработке большой системы с большим количеством микросервисов, управление начнет выходить из-под контроля. Чтобы построить

автоматизированную и самовосстанавливающуюся распределенную систему, понадобятся следующие функции:

- централизованная система версионирования микросервисов
- сервис доставки
- балансировщик нагрузки
- аутентификация и безопасность
- параллельный ввод/вывод
- последовательность выполнения межсервисной логики
- отказоустойчивость
- распределенность сессий
- распределенная системы кэширования
- динамичная маршрутизация
- автоматическое масштабирование микросервисов
- плавное развертывание микросервисов
- независимость платформы
- агрегирование логов

Выбрав один из подходов проектирования *backend* части приложения необходимо провести сравнительный анализ языков программирования, предназначенных для разработки серверной части приложения, а также составить *UML*-диаграммы для более отчетливого понимания взаимосвязей компонентов разрабатываемой информационной системы.

В процессе выбора языка программирования, были выделены следующие кандидаты:

- *PHP*
- *Golang*

Для понимания, какой язык разработки выбрать для определенной задачи необходимо провести небольшой анализ.

Таблица 1 - Анализ PHP

PHP	
Плюсы	Минусы
Один из самых распространенных языков программирования	Плохо спроектирован
Большой выбор обучающих материалов и примеров разработки	Большое количество небезопасных пакетов
Используется большинством распространенных платформ	Из-за скорости развития появилось большое количество устаревших обучающих материалов
Обладает отличным сторонним менеджером пакетов	Интерпретатор имеет излишние права
Начиная с версии 7.* язык стал показывать неплохие результаты в производительности	-
Большое количество готовых решений для различных задач	-
Отличная документация	-
Возможность создания собственного модуля, с помощью интеграции в ядро	-
Быстрая скорость разработки	-

Таблица 2 - Анализ Golang

Golang	
Плюсы	Минусы
Исключительно простое и масштабируемое многопоточное и параллельное программирование	Снижение производительности из-за косвенных вызовов сборщика мусора

Упрощенный *C*-подобный синтаксис	Сложная в понимании системы интерфейсов
Мощная поддержка языка	Слабая система типов
Быстрый компилятор, интегрируемый в обычный *gcc*	-
Отличная документация	-
Встроенное модульное тестирование	-
Предоставляет инструменты для автоматического форматирования кода	-
Простота установки и настройки	-
Отлично подходит для построения сетевых сервисов	-
Поддерживает модульность в виде пакетов	-
Отличный встроенный пакетный менеджер	-
Демонстрирует уникальную, простую концепцию объектно-ориентированного программирования	-
Высокопроизводительный	-

Ниже представлены диаграмма классов, компонентов, развертывания для понимания связанности компонентов информационной системы и архитектурной сложности серверной части, а также выявления сторонних зависимостей и наглядного представления узких мест в реализации *backend* части приложения.

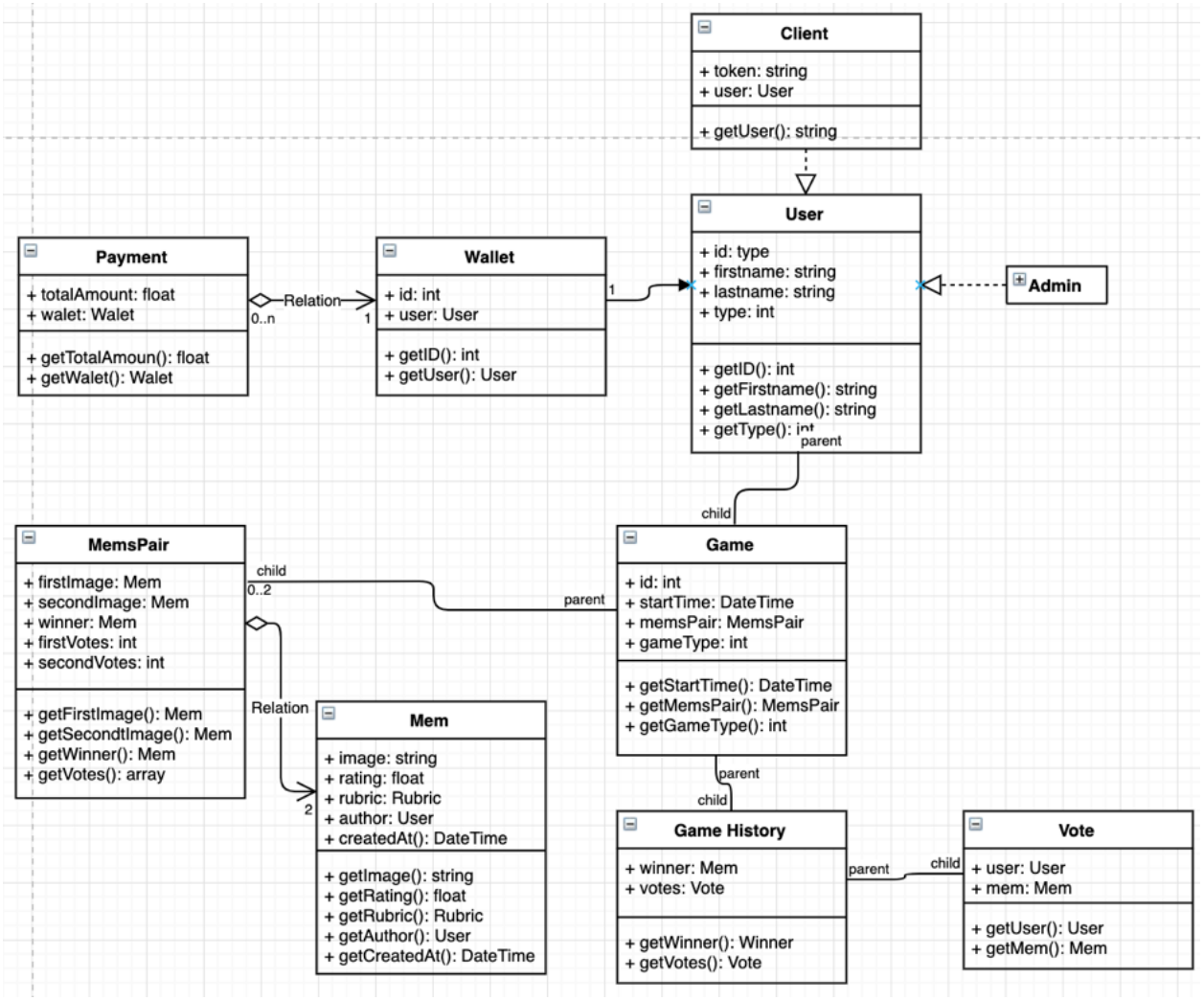


Рисунок 1 - Диаграмма классов

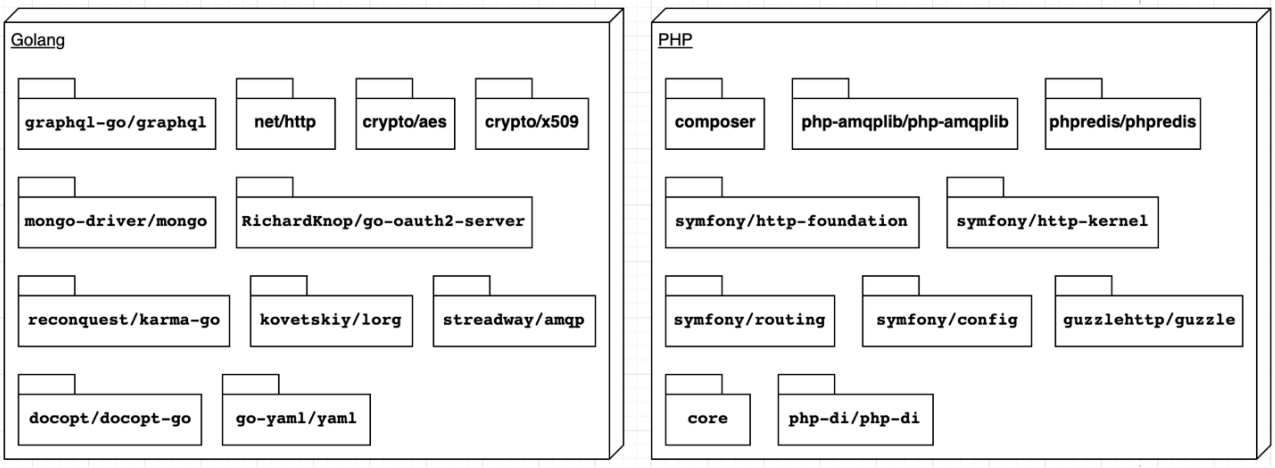


Рисунок 2 - Диаграмма компонентов

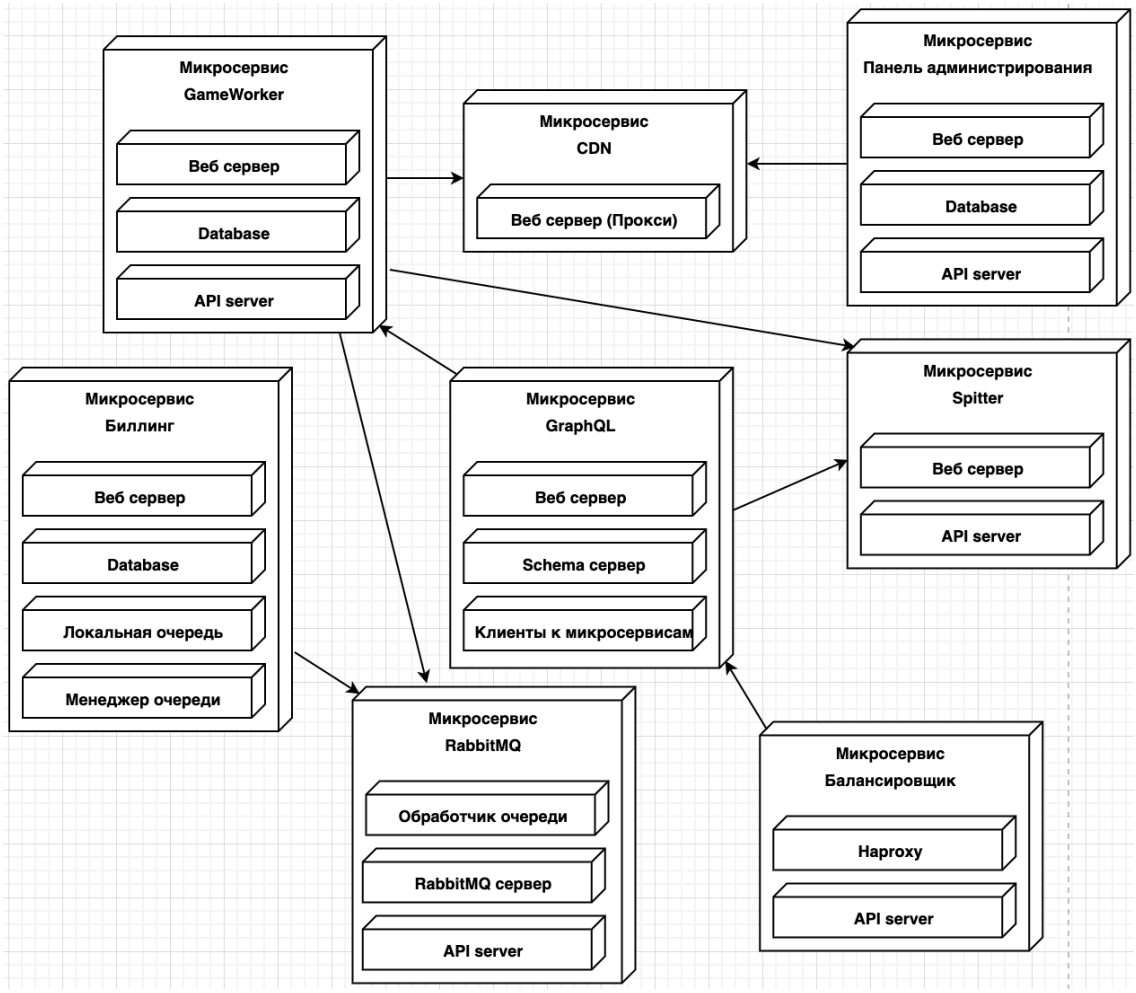


Рисунок 3 - Диаграмма развертывания

Для стандартизации процесса командной и межкомандной работы был спроектирован процесс взаимодействия.

В качестве платформы для взаимодействия членов команды была выбрана *JIRA*. Методология разработки - *Agile*. Процесс взаимодействия строится через менеджера продукта.

Рисунок 4 - Процесс взаимодействия членов команды

АНАЛИЗ ТЕХНОЛОГИЧЕСКИХ РЕШЕНИЙ

Для эффективного выполнения поставленных задач в разработке серверной части информационной системы «*MemeBattle*» необходимо иметь четкое представление о возможностях, производительности и актуальности сервисных технологических решений.

Важной функцией в проекте выступает поиск по различным медиафайлам сервиса, полнотекстный поиск, а также аналитика данных.

Среди функций, которые возможно получить от современных поисковых систем, наиболее популярными являются:

- полнотекстовый поиск (простыми словами и фразами или несколькими формами слова или фразы)
- многопрофильный поиск
- выделение (визуальная индикация слов, введенных в поле поиска)
- поиск по синонимам
- автозаполнение предложений
- фасетный поиск (количество атрибутов. Например, сайты электронной коммерции используют фасеты, чтобы сообщать клиентам, сколько элементов определенной модели, размера, цвета и других атрибутов найдено)

Система должна иметь возможность сузить поиск, используя диапазоны (цена, даты, размеры и т. д.), сортировку (по популярности, дате, цене) и фильтрацию (включая только желательные параметры).

Когда мы говорим о веб-приложениях, в которых информация динамически изменяется (цены, подробности описания, наличие товаров), крайне важно иметь обновления почти в реальном времени; например, в системах электронной коммерции или бронирования, чтобы показать товары и услуги, имеющиеся в наличии на складе.

Помимо общих функций, перечисленных выше, поисковые механизмы могут предоставлять рекомендации при поиске наиболее интересных продуктов или информации, чтобы улучшить взаимодействие с пользователем.

С точки зрения информационной системы, мы должны рассматривать эффективную поисковую систему как мощный инструмент, способный повысить коэффициент производительности. Если механизм поиска информации не дает релевантных результатов или его эффективность поиска слишком низкая, целевой клиент покинет систему и перейдет к его конкуренту.

Исходя из возможностей и производительности сервисов, были выбраны наиболее подходящие:

- *ElasticSearch*
- *Solr*
- *Sphinx*

Все три являются поисковыми решениями с открытым исходным кодом, которые хорошо поддерживаются сообществом авторов. Все они могут похвастаться высокой производительностью, масштабируемостью и гибкостью, хотя у них все еще есть свои особенности.

ElasticSearch

Elasticsearch, абсолютный лидер рейтинга поисковых систем, подтверждает свое имя «по-настоящему эластичным», работая в любой среде. Это технология с открытым исходным кодом, использующая библиотеку *Apache Lucene*. Многие всемирно известные компании используют *Elasticsearch* для своих приложений.

Сильные стороны *ElasticSearch*:

1. Индексация в режиме реального времени

Elasticsearch способен индексировать быстро изменяющиеся данные практически мгновенно (менее чем за 1 секунду). Уместно использовать его в проектах, где база данных постоянно обновляется.

2. Высокая масштабируемость

Когда база данных растет, поиск становится все труднее. Но *Elasticsearch* расширяется, в то время как ваша БД увеличивается, поэтому скорость поиска не замедляется.

3. Хранение

Elasticsearch можно использовать не только как индексатор, но и как хранилище данных. Тем не менее, не рекомендуется использовать его в качестве основного хранилища, и по-прежнему необходимо хранить данные в основной БД для повышения безопасности и надежности, используя *Elasticsearch* только для индексации данных и хранения журналов. С *Elasticsearch*, данные хранятся в нашей БД, быстро индексируется и становится доступными для поиска пользователями мгновенно.

4. Визуализация данных

Это одна из самых модных на сегодня функций, которая отлично реализована в *Elasticsearch*. *Elastic Stack* (комбинация плагинов *ES*, *Logstash* и *Kibana*) делает отличный инструмент для аналитики. Это позволяет в режиме реального времени производить отслеживание трафика в вашем приложении (общее количество посетителей, количество уникальных посетителей, *IP*-адреса, самые популярные запросы, самые запрашиваемые страницы, используемые устройства и браузеры, журналы трафика по времени суток и многое другое).

Эта информация отображается в диаграммах, картах и таблицах на панели инструментов. Это очень полезно для работы с распределенными командами, поскольку каждый может сразу увидеть актуальную информацию, а затем использовать эти данные, чтобы лучше понять свою аудиторию и улучшить контент и *UX* продукта.

5. Аналитика безопасности

Elastic Stack также является отличным инструментом анализа безопасности. Аналитика и визуализация журналов практически в реальном времени позволяют выявлять угрозы безопасности (проблемы с веб-сервером, неработающие ссылки, попытки несанкционированного доступа, места атак и т. д.)

6. Машинное обучение

Elasticsearch может воспользоваться функциями машинного обучения, предоставляемыми коммерческим плагином *X-Pack*. Алгоритмы машинного обучения направлены на обнаружение аномалий и обнаружение выбросов в данных временных рядов.

7. *Amazon Elasticsearch* сервис

Сервис *Amazon Elasticsearch* обеспечивает быструю и простую настройку, работу и масштабирование *Elasticsearch* в облаке без необходимости настройки собственных серверов.

Недостатки *Elasticsearch.*

Хотя *Elasticsearch* в настоящее время номер 1, это все еще молодая технология. Не все желаемые функции выходят из «коробки», и многие должны быть добавлены через различные расширения. Например, в *Elasticsearch* нет функции «Вы имели в виду?».

Solr

Solr - еще одна поисковая система, основанная на *Apache Lucene*, и, таким образом, она имеет много общих черт с *Elasticsearch*. Но, тем не менее, они разные по архитектуре.

Сильные стороны *Solr*:

1. Поиск по лицу

Solr обладает потрясающими возможностями граненого поиска, что делает это решение идеальным для приложений электронной коммерции, которые используют *Solr* для поиска и навигации по 150 000 стилей обуви и других продуктов.

2. Богатый набор особенностей

Solr может похвастаться богатой функцией полнотекстового поиска из коробки, которая легко настраивается (даже больше, чем *Elasticsearch*). *Solr* поддерживает различные реализации подсказок, выделяя функциональность (визуальная индикация слов, введенных в поле) и средства проверки орфографии / «Вы имели в виду?»

3. Богатое содержание документов

Solr - одна из немногих поисковых систем, которая может читать документы с расширенным содержимым, включая *PDF, Word, XML* или простой текст.

Это идеально подойдет для проектов, где необходимо просматривать большое количество файлов *PDF* или *Word* на веб-сайте.

4. Визуализация данных

Banana - это инструмент визуализации («форк» *Kibana*), который работает для *Solr* и позволяет администраторам отслеживать события и журналы на панели инструментов на лету.

Например, в банковской сфере менеджеры смогут получать информацию о неудачных транзакциях и выяснять причину каждой проблемы практически «на лету», что значительно сокращает ручную работу. Это также может уменьшить ручной поиск по журналам.

5. Машинное обучение

Solr в сотрудничестве с *Bloomberg* внедрил Машинное обучение (плагин *Learning-to-Rank*), используя концепцию переупорядочения документов в соответствии с оценкой более сложного запроса. Машинное обучение направлено на то, чтобы предоставить подписчикам еще лучший опыт мгновенного поиска наиболее значимых компаний, людей и новостей.

Недостатки *Solr.*

Solr не так быстр, как *Elasticsearch*, и лучше всего работает со статическими данными, что не требует частых изменений. Причина в кеше. В *Solr* кеши являются глобальными, это означает, что, когда в кеше происходит даже малейшее изменение, вся индексация требует обновления. Обычно это трудоемкий процесс. В *Elastic*, с другой стороны, обновление производится сегментами.

Sphinx

Sphinx, занимает только 5-е место среди поисковых систем, хотя это все еще мощная и популярная технология.

Сильные стороны

1. Быстродействие

Sphinx развивался в последние годы и стал способен обеспечивать поиск практически в реальном времени. Его скорость включает более 500 запросов в секунду к 1000000 документов, при этом наибольшее зарегистрированное число индексаций оценивается в 25 с лишним миллиардов документов.

2. Поиск

Sphinx имеет большой опыт работы с возможностями граненого поиска.

3. Ничего лишнего

Если в информационной нужны общие функции поиска и не нужны никакие дополнительные функции, такие как визуализация и анализ данных, рекомендуется использовать *Sphinx*. Он довольно быстрый и мощный для индексирования и запроса огромных объемов документов с использованием ограниченных вычислительных ресурсов, в отличие от *Elasticsearch*, который потребляет много памяти.
Слабые стороны
Sphinx хорош для структурированных данных, но это не лучший выбор для проектов, которые работают с неструктурированными данными (*DOC, PDF, MP3* и т. д.), так как отнимает у разработчиков много времени и усилия по настройке. Это, наряду с другими трудностями в настройке, делает *Sphinx* менее удобным в использовании, чем его конкуренты.
Наряду с функцией поиска важной частью системы является сервис кэширования. В качестве *key-value* хранилищ были отобраны для сравнения два сервиса: ***Redis и Memcached***.
Redis и *Memcached* являются системами хранения данных в памяти. *Memcached* - это высокопроизводительная служба кэш-памяти с распределенной памятью, а *Redis* - хранилище значений ключей с открытым исходным кодом. Как и в *Memcached*, *Redis* хранит большую часть данных в памяти. Он поддерживает операции с различными типами данных, включая строки, хеш-таблицы и связанные списки среди других.

Сравнение функций

Автор *Redis*, поделился следующими моментами сравнения между *Redis* и *Memcached*:

Операции с данными на стороне сервера

Redis поддерживает операции с данными на стороне сервера, владеет большим количеством структур данных и поддерживает более сложные операции с данными, чем *Memcached*. В *Memcached* обычно нужно скопировать данные на стороне клиента для подобных изменений, а затем вернуть данные обратно. В результате это значительно увеличивает количество операций ввода-вывода в сети и размер данных. В *Redis* эти сложные операции так же эффективны, как и общие

операции *GET* / *SET*. Поэтому, если нужен кеш для поддержки более сложных структур и операций, *Redis* - хороший выбор.

Сравнение эффективности использования памяти

Memcached имеет более высокую степень использования памяти для простого хранения значения ключа. Но если *Redis* примет хеш-структуру, он будет иметь более высокую степень использования памяти, чем *Memcached*, благодаря комбинированному режиму сжатия.

Сравнение производительности

Redis использует только одно ядро, а *Memcached* использует несколько ядер. Таким образом, в среднем *Redis* может похвастаться более высокой производительностью, чем *Memcached*, в небольшом хранилище данных, если измерять его в терминах ядер. *Memcached* превосходит *Redis* для хранения данных размером 100 КБ и выше. Хотя *Redis* также оптимизировал хранение больших данных, он все же уступает *Memcached*.

Теперь стоит обсудить некоторые моменты в поддержку приведенных выше сравнений.

Поддерживаются разные типы данных

В отличие от *Memcached*, который поддерживает только записи данных с простой структурой ключ-значение, *Redis* поддерживает гораздо более богатые типы данных, включая *String, Hash, List, Set* и *Sorted Set*. *Redis* использует *redisObject* для представления всех ключей и значений.

Основная информация о *redisObject* показана ниже:

Рисунок 5 - Структура redisObject

Тип представляет тип данных объекта значения. Шифрование указывает на способ хранения различных типов данных в *Redis*, например *type = string* означает, что значение хранит общую строку, и соответствующее шифрование может быть необработанной или *int*. Если это *int*, *Redis* сохраняет и представляет связанную строку как тип значения. Конечно, предпосылка состоит в том, что можно представить строку значением, таким как строки «111» и «2222». Только после включения функции виртуальной памяти *Redis* он будет распределять поля виртуальной памяти с памятью хранилища. Эта функция отключена по умолчанию. Стоит также рассмотреть некоторые типы данных.

Строка

Часто используемые команды: *set* / *get* / *decr* / *incr* / *mget* и так далее.

Сценарии применения: строка является наиболее распространенным типом данных, а общий ключ / значение относятся к этой категории.

Метод реализации: *String* является символьной строкой по умолчанию в *Redis*, на которую ссылается *redisObject*. При вызове для операций *INCR* или *DECR* система преобразует его в тип значения для вычисления. В настоящее время поле кодирования *redisObject* - *int*.

Хэш

Часто используемые команды: *hget* / *hset* / *hgetall* и так далее.

Сценарии применения: хранение данных объекта информации о профиле пользователя, включая идентификатор пользователя, имя пользователя, возраст и день рождения; получение имени пользователя, возраста или дня рождения через идентификатор пользователя.

Метод реализации: *Hash* в *Redis* - это *HashMap* внутреннего хранимого значения, который предоставляет интерфейс для прямого доступа к этому объекту *HashMap*. Ключ - это идентификатор пользователя, а значение - это *HashMap*. Ключ этой *HashMap* - имя атрибута члена, а значение - значение атрибута. Таким образом, можно напрямую выполнить изменения и получить доступ к данным через ключ внутренней *HashMap* (в *Redis* ключ внутренней *HashMap* называется полем), то есть через ключ (идентификатор пользователя) + поле (тег атрибута) для выполнять операции с соответствующими атрибутами данных.

Существует 2 способа реализации текущего *HashMap*: когда в *HashMap* всего несколько элементов, *Redis* выбирает одномерные массивы для компактного хранения для экономии памяти вместо структуры *HashMap* в реальном смысле. В это время кодирование соответствующего значения *redisObject* представляет собой *zipmap*. Когда число членов увеличивается, *Redis* преобразует их в *HashMap*, и кодировка в это время будет *ht*.

Список

Часто используемые команды: *lpush* / *rpush* / *lpop* / *rpop* / *lrange*.

Сценарии применения. Список *Redis* - это самая важная структура данных в *Redis*. Фактически, можно реализовать следующий список пользователей и список друзей, используя структуру списка *Redis*.

Метод реализации: через двусторонний связанный список, который поддерживает обратный поиск и обход для облегчения операций. Но это также требует некоторых дополнительных затратах памяти. Многие реализации в *Redis*, включая отправку буферных очередей, также принимают эту структуру данных.

Набор

Часто используемые команды: *sadd* / *spop* / *smembers* / *sunion* и так далее.

Сценарии применения: *Redis set* предоставляет функцию внешнего списка, аналогичную функции *list*. Его особенность в том, что набор может автоматически удалять дубликаты. Когда вам нужно сохранить список данных без дублирования, набор является хорошим вариантом. Кроме того, набор предоставляет важный интерфейс для оценки того, находится ли участник в наборе, это функция, не предоставляемая списком.

Метод реализации: Внутренняя реализация *set* - это *HashMap*, значение которого всегда равно *null*. Он на самом деле быстро удаляет дубликаты, вычисляя значения хешей. Фактически, это также причина, по которой набор может судить, входит ли участник в набор.

Сортированный набор

Часто используемые команды: *zadd* / *zrange* / *zrem* / *zcard* и так далее.

Сценарии приложения: Сценарии приложения для отсортированного набора *Redis* аналогичны сценариям для набора. Разница в том, что хотя набор не сортирует данные автоматически, отсортированный набор может сортировать элементы по приоритетному параметру (счету), предоставленному пользователем. Более того, последний также сортирует вставленные данные автоматически. Вы можете выбрать отсортированную структуру данных набора, когда вам нужен упорядоченный список наборов без дубликатов данных, например, общедоступная временная шкала любой социальной сети, которая может принимать время публикации в качестве оценки для хранения с автоматической сортировкой данных, полученных по времени.

Метод реализации: отсортированный набор *Redis* использует *HashMap* и *SkipList* для обеспечения эффективного хранения и порядка данных. *HashMap* хранит сопоставления между участником и счетом; в то время как *SkipList* хранит всех участников. Сортировка основывается на баллах, хранящихся в *HashMap*. Использование структуры *SkipList* может повысить эффективность поиска и упростить реализацию.

Другая схема управления памятью.

В *Redis* не все данные хранятся в памяти. В этом основная разница между *Redis* и *Memcached*. Когда физическая память заполнена, *Redis* может поменять

значения, которые долгое время не использовались, на диск. *Redis* только кэширует всю ключевую информацию. Если обнаруживается, что использование памяти превышает пороговое значение, запускается операция подкачки. *Redis* вычисляет значения для ключей, которые должны быть перенесены на диск. Затем эти значения ключей сохраняются на диске и удаляются из памяти. Эта функция позволяет *Redis* сохранять данные, размер которых превышает объем памяти машины. В памяти машины должны храниться все ключи, и она не поменяет местами все данные.

В то же время, когда *Redis* меняет данные в памяти на диск, основной поток, который предоставляет сервисы, и дочерний поток для операции подкачки будут совместно использовать эту часть памяти. Таким образом, если данные обновятся, которые необходимо поменять, *Redis* заблокирует эту операцию, предотвращая выполнение такого изменения до тех пор, пока дочерний поток не завершит операцию подкачки. Когда происходит чтение данных из *Redis*, если значение ключа чтения не находится в памяти, *Redis* необходимо загрузить соответствующие данные из файла подкачки и затем вернуть их запрашивающей стороне. Здесь существует проблема пула потоков ввода / вывода. По умолчанию *Redis* сталкивается с перегрузкой, то есть он отвечает только после успешной загрузки всех файлов подкачки. Эта политика подходит для пакетных операций при небольшом количестве клиентов. Но если *Redis* применяется в большой информационной системе, она не сможет удовлетворить высокие требования к параллелизму. Однако можно установить размер пула потоков ввода-вывода для работы *Redis* и выполнять параллельные операции для запросов на чтение для загрузки соответствующих данных в файл подкачки, чтобы сократить время перегрузки.

Для систем баз данных на основе памяти, таких как *Redis* и *Memcached*, эффективность управления памятью является ключевым фактором, влияющим на производительность системы. В традиционном языке *C* функции *malloc* / *free* являются наиболее распространенным методом распределения и освобождения памяти. Однако этот метод таит в себе огромный недостаток: во-первых, для разработчиков несравненный *malloc* и *free* легко вызовут утечку памяти; во-вторых,

частые вызовы затруднят переработку и повторное использование большого количества фрагментов памяти, уменьшая использование памяти; и, наконец, системные вызовы потребляют гораздо больше системных ресурсов, чем вызовы общих функций. Поэтому для повышения эффективности управления памятью решения для управления памятью не будут использовать вызовы *malloc* / *free* напрямую. И *Redis*, и *Memcached* используют свои собственные механизмы управления памятью, но методы реализации сильно различаются.

Memcached по умолчанию использует механизм «распределения плиты» для управления памятью. Его основная идея состоит в том, чтобы сегментировать выделенную память на куски заранее определенной длины, чтобы хранить записи данных ключ-значение соответствующей длины, чтобы полностью решить проблему фрагмента памяти. В идеале, структура плиты из механизма размещения должна гарантировать внешнее хранение данных, то есть она облегчает хранение всех данных со значением ключа в системе «распределения плит». Однако применение других запросов к памяти в *Memcached* происходит через обычные вызовы *malloc* / *free*. Обычно это происходит потому, что количество и частота этих запросов определяют, что они не повлияют на общую производительность системы. Принцип «распределение плиты» плиты очень прост.

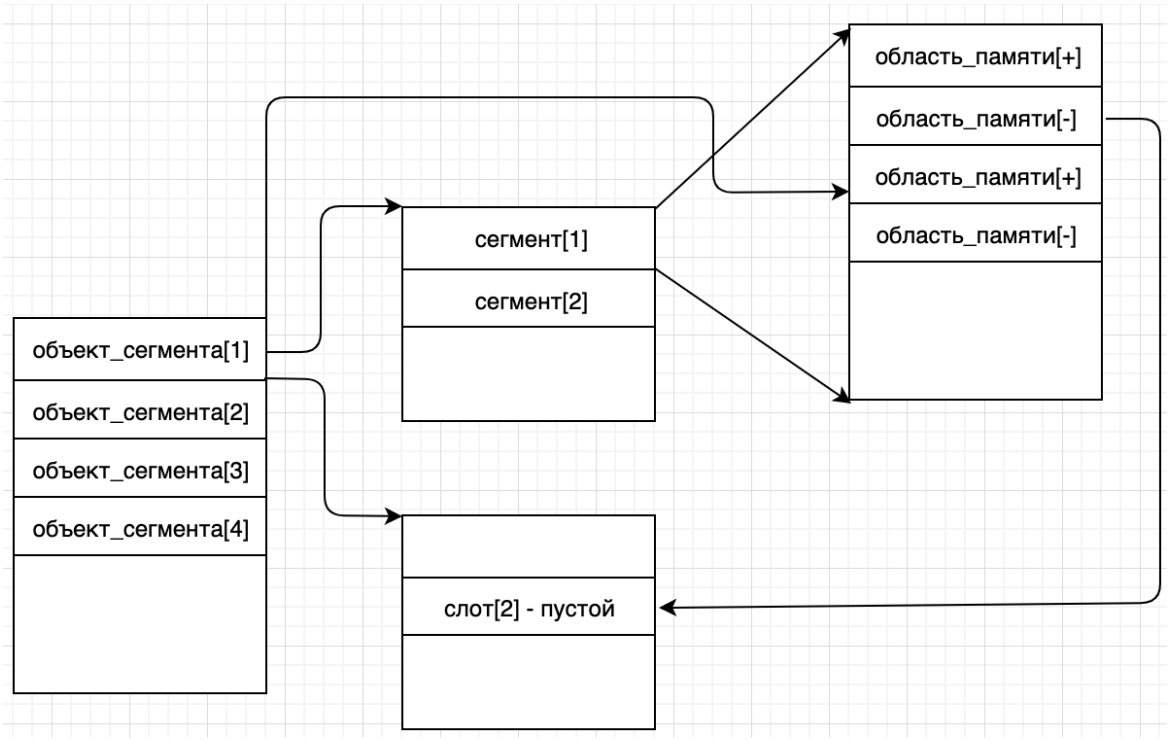


Рисунок 6 - Принцип "разделения плит"

Как показано на рисунке, сначала он запрашивает большую часть памяти операционной системы и сегментирует ее на куски разных размеров, а затем группирует куски одинакового размера в объект сегмента. Среди них блок является наименьшей единицей для хранения данных значения ключа. Можно контролировать размер каждого класса плиты, создав фактор роста при запуске *Memcached*. Предположим, коэффициент роста равен 1,25. Если размер фрагмента в первой группе составляет 88 байтов, размер фрагмента во второй группе будет 112 байтов. Остальные куски следуют тому же правилу.

Когда *Memcached* получает данные, отправленные от клиента, он сначала выбирает наиболее подходящий объект сегмента в соответствии с размером данных, а затем запрашивает список пустых областей памяти, содержащих объект сегмента памяти в *Memcached*, чтобы найти область для хранения данных. Когда срок действия части данных истекает или устарел, и, следовательно, они отбрасываются, можно повторно использовать область памяти, изначально занятый информацией, и восстановить его в списке ожидания.

Из приведенного выше процесса следует, что *Memcached* обладает очень высокой эффективностью управления памятью, которая не приводит к фрагментам памяти. Однако самый большой его недостаток заключается в том, что он может стать причиной огромных потерь. Поскольку система выделяет каждый кусок в области памяти определенной длины, более длинные данные могут не использовать пространство полностью. Как показано на рисунке, когда мы кэшируем данные размером 100 байтов в порцию из 128 байтов, неиспользуемые 28 байтов теряются.

Реализация управления памятью *Redis* в основном осуществляется через два файла *zmalloc.h* и *zmalloc.c* в исходном коде. Чтобы упростить управление памятью, *Redis* сохранит объем памяти в заголовке блока памяти после выделения

памяти.

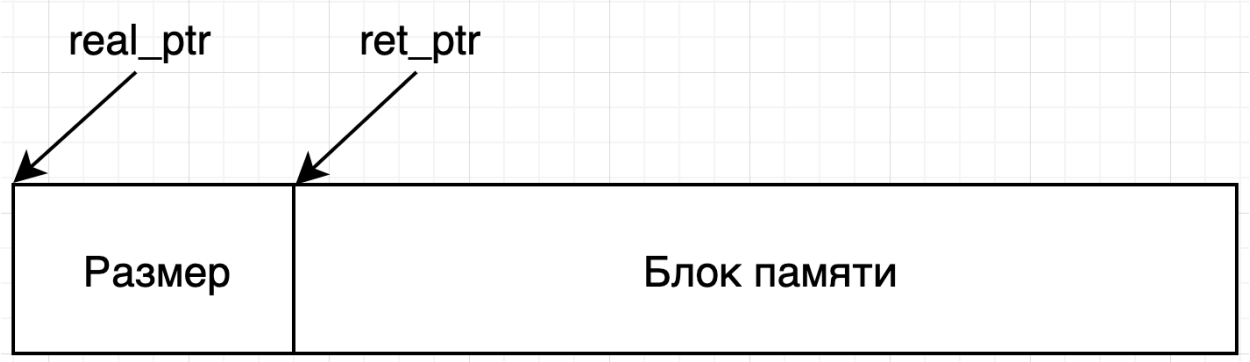


Рисунок 7 - Организация памяти в Redis

Как показано на рисунке, *real_ptr* - это указатель, возвращаемый после того, как *Redis* вызывает *malloc*. *Redis* сохраняет размер блока памяти в заголовке, и память, занимаемая информацией, определяется, то есть система возвращает длину типа *size_t*, а затем *ret_ptr*. Когда возникает необходимость освободить память, система передает *ret_ptr* в программу управления памятью. Через *ret_ptr* программа может легко вычислить значение *real_ptr* и затем передать *real_ptr* для освобождения памяти.

Redis записывает распределение всей памяти, определяя массив, длина которого равна *ZMALLOC_MAX_ALLOC_STAT*. Каждый элемент в массиве представляет количество блоков памяти, выделенных текущей программой, а размер блока памяти является индексом элемента. В исходном коде этот массив является *zmalloc_allocations*.

```
void update_zmalloc_stat_alloc(__n,__size)
{
   do {
      size_t _n = (__n);
      size_t _stat_slot = (__size < ZMALLOC_MAX_ALLOC_STAT) ? __size : ZMAL-
LOC_MAX_ALLOC_STAT;
      if (_n&(sizeof(long)-1)) _n += sizeof(long)-(_n&(sizeof(long)-1));
      if (zmalloc_thread_safe) {
         pthread_mutex_lock(&used;_memory_mutex);
         used_memory += _n;
         zmalloc_allocations[_stat_slot]++;
```

```
        pthread_mutex_unlock(&used;_memory_mutex);
      } else {
        used_memory += _n;
        zmalloc_allocations[_stat_slot]++;
      }
  } while(0)
}
```

Zmalloc_allocations представляет количество блоков памяти, выделенных длиной 16 байтов. В *zmalloc.c* содержится статическая переменная *used_memory* для записи общего размера выделенной в данный момент памяти. *Redis* использует инкапсулированный *malloc* / *free*, что намного проще по сравнению с механизмом управления памятью *Memcached*.

Поддержка сохранности данных.

Хотя хранилище на основе памяти, *Redis* поддерживает сохранение данных в памяти и предоставляет две основные политики сохранения, снимок *RDB* и журнал *AOF*. *Memcached* не поддерживает операции сохранения данных.

Снимок *RDB*

Redis поддерживает хранение снимка текущих данных в файле данных для сохранения, то есть снимок *RDB*. Но как мы можем создать снимок для базы данных с непрерывной записью данных? *Redis* использует механизм копирования при записи команды *fork*. После создания моментального снимка текущий процесс создает подпроцесс, который делает все данные циклическими и записывает их в файл *RDB*. Мы можем настроить время создания снимка *RDB* с помощью команды сохранения *Redis*. Например, если вы хотите настроить создание снимка один раз каждые 10 минут, вы можете настроить создание снимка после каждой 1000 записей. Вы также можете настроить несколько правил для реализации вместе. Определения этих правил находятся в конфигурационных файлах *Redis*. Вы также можете установить правила с помощью команды *CONFIG SET Redis* во время выполнения *Redis* без перезапуска *Redis*.

Файл *RDB Redis*, в некоторой степени, не будет поврежден, потому что он выполняет свои операции записи в новом процессе. После создания нового файла

RDB подпроцесс, сгенерированный *Redis*, сначала записывает данные во временный файл, а затем переименовывает временный файл в файл *RDB* с помощью системного вызова атомарного переименования, так что файл *RDB* всегда доступен всякий раз, когда *Redis* терпит ошибку. В то же время файл *RDB* также является ссылкой во внутренней реализации синхронизации *Redis Master-Slave*. Тем не менее, *RDB* имеет недостаток в том, что как только в базе данных возникает какая-то проблема, данные, сохраненные в файле *RDB*, могут быть устаревшими, и данные будут потеряны в течение периода с момента последнего создания файла *RDB* до сбоя *Redis*.

Журнал *AOF*

Полная форма журнала *AOF* - «Добавить только файл». Это добавленный файл журнала. В отличие от бинарного файла общих баз данных, *AOF* - это узнаваемый открытый текст, а его содержимое - стандартные команды *Redis*. *Redis* будет добавлять только те команды, которые вызывают изменения данных в *AOF*. Каждая команда для изменения данных будет генерировать журнал. Файл *AOF* будет становиться все больше и больше. *Redis* предоставляет еще одну функцию - переписать *AOF*. Функция перезаписи *AOF* заключается в повторной генерации файла *AOF*. Существует только одна операция для каждой записи в новом файле *AOF* вместо нескольких операций для одного и того же значения, записанного в старой копии. Процесс генерации аналогичен моментальному снимку *RDB*, а именно разветвлению процесса, прохождению данных и записи данных в новый временный файл *AOF*. При записи данных в новый файл он записывает все журналы операций записи в старый файл *AOF* и одновременно записывает их в зону буферизации памяти. По завершении операции система запишет все журналы в зоне буферизации во временный файл за один раз. После этого он вызовет команду атомарного переименования, чтобы заменить старый файл *AOF* новым файлом *AOF*.

AOF является операцией записи файла и предназначена для записи журналов операций на диск. Это также включает в себя процедуру операции записи, которую мы упоминали ранее. После того, как *Redis* вызывает операцию записи для *AOF*, он использует опцию *appendfsync*, чтобы контролировать время записи

данных на диск, вызывая команду *fsync*. Три параметра настройки в *appendfsync* ниже имеют уровень безопасности от низкого до сильного.

appendfsync no: когда мы устанавливаем для *appendfsync* значение *no*, *Redis* не будет брать на себя инициативу вызова *fsync* для синхронизации журналов *AOF* с диском. Синхронизация будет полностью зависеть от отладки операционной системы. Большинство операционных систем *Linux* выполняют операцию *fsync* каждые 30 секунд, чтобы записать данные в зоне буферизации на диск.

appendfsync everysec: когда мы устанавливаем *appendfsync* равным *everysec*, *Redis* будет вызывать *fsync* раз в секунду по умолчанию для записи данных в зоне буферизации на диск. Но когда вызов *fsync* длится более 1 секунды, *Redis* примет задержку *fsync*, чтобы подождать еще одну секунду. То есть *Redis* вызовет *fsync* через две секунды. Он выполнит этот *fsync* независимо от того, сколько времени потребуется для его выполнения. В настоящее время, поскольку дескриптор файла будет испытывать перегрузку во время файла *fsync*, текущая операция записи будет испытывать аналогичную перегрузку. Вывод заключается в том, что в подавляющем большинстве случаев *Redis* будет выполнять *fsync* один раз в секунду. В худшем случае он будет выполнять операцию *fsync* каждые две секунды. Большинство систем баз данных называют эту операцию групповой фиксацией, то есть объединением данных нескольких записей и записью журналов на диск за раз.

appednfsync always: когда мы устанавливаем *appendfsync* в значение *Always*, каждая операция записи будет вызывать *fsync* один раз. На данный момент данные наиболее безопасны. Конечно, поскольку он выполняет *fsync* каждый раз, это ухудшит производительность.

Разница в управлении кластерами

Memcached - это система буферизации данных с полной памятью. Несмотря на то, что *Redis* поддерживает постоянство данных, суть высокой производительности заключается в полной памяти. Для хранилища на основе памяти размер памяти физической машины - это максимальная емкость системы для хранения данных. Если размер данных, который предполагается обработать, превышает

размер физической памяти одного компьютера, необходимо создать распределенные кластеры, чтобы увеличить емкость хранилища.

Сам *Memcached* не поддерживает распределенный режим. Распределенное хранение *Memcached* на стороне клиента можно получить только с помощью распределенных алгоритмов, таких как *Consistent Hash*. Прежде чем клиентская сторона отправляет данные в кластер *Memcached*, она сначала вычисляет целевой узел данных с помощью вложенного распределенного алгоритма, который, в свою очередь, напрямую отправляет данные на узел для хранения. Но когда клиентская сторона запрашивает данные, ей также необходимо вычислить узел, который служит в качестве местоположения запрашиваемых данных, а затем отправить запрос на запрос непосредственно для получения данных.

По сравнению с *Memcached*, который может создавать распределенное хранилище только на стороне клиента, *Redis* предпочитает создавать распределенное хранилище на стороне сервера. Последняя версия *Redis* поддерживает распределенное хранилище. *Redis Cluster* - это расширенная версия *Redis*, которая обеспечивает распределенное хранилище и поддерживает *SPOF*. Он не имеет центрального узла и способен к линейному расширению. На рисунке ниже представлена архитектура распределенного хранилища *Redis Cluster*. Связь между узлами следует двоичному протоколу, но связь узел-клиент следует протоколу *ASCII*. В политике размещения данных *Redis Cluster* делит весь числовой диапазон ключа на 4096 слотов хэша и позволяет хранить один или несколько слотов хэша на каждом узле. То есть текущий кластер *Redis* поддерживает максимум 4096 узлов. Распределенный алгоритм, который использует *Redis Cluster*, прост: *crc*16 (ключ) % *HASH_SLOTS_NUMBER*.

Redis Cluster представляет главный узел и подчиненный узел для обеспечения доступности данных в случае *SPOF*. Каждый главный узел в *Redis Cluster* имеет два соответствующих подчиненных узла для резервирования. В результате любые два отказавших узла во всем кластере не повлияют на доступность данных. Когда главный узел существует, кластер автоматически выбирает подчиненный узел, чтобы стать новым главным узлом. Были рассмотрены различия между *Redis* и *Memcached*. Разобраны ключевые отличиями между *Redis* и *Memcached*:

поддерживаемые типы данных, управление кластерами, поддержка постоянства и консистентность данных и схемы управления памятью.

Базы данных.

Также одним важным компонентом в любой информационной системе является функция хранения данных, точнее способ хранения. Поэтому следует сравнить различные подходы в базах данных, а именно: *NoSQL* и *SQL*.

При выборе базы данных, нужно понимать для каких целей она понадобится, какие операции будут проводиться с данными и какого формата буду эти данные.

«*SQL* против *NoSQL*» - не что иное, как сравнение реляционных и нереляционных баз данных. Разница заключается в их архитектуре, в формате данных, и как они их хранят. Реляционные базы данных структурированы, а нереляционные базы данных ориентированы на документы и распределены. Уже более четырех десятилетий базы данных языка структурированных запросов (*SQL*) являются основным механизмом хранения данных. С ростом популярности веб-приложений и опций с открытым исходным кодом, таких как *PostgreSQL*, *MySQL* и *SQLite*, их использование резко возросло в конце 1990-х годов. Базы данных *NoSQL* в последнее время набирают популярность благодаря таким опциям, как *MongoDB*, *CouchDB*, *Apache Cassandra*, хотя они существуют с 1960-х годов. И *SQL*, и *NoSQL* делают одно и то же, хранят данные. За исключением того, что их подходы отличаются. Несмотря на растущую популярность, *NoSQL* не является заменой *SQL*. Это альтернатива. Некоторые проекты лучше подходят для использования базы данных *SQL*, а другие хорошо работают с *NoSQL*. Некоторые могут использовать оба взаимозаменяемо.

SQL

*Я*зык структурированных запросов (*SQL*) оказывается более структурированным и жестким способом хранения данных, как телефонная книга. Чтобы реляционная база данных была эффективной, хранить данные необходимо очень организованно. Базы данных *SQL* остаются популярными, потому что они естественным образом вписываются во многие стеки программного обеспечения,

включая стек *LAMP*. Эти базы данных широко поддерживаются и хорошо понимаются, что может стать большим плюсом, если возникнут проблемы.
Не существует единого решения, подходящего для всего, когда дело доходит до технологий баз данных. Вот почему большинство полагаются на нереляционные и реляционные базы данных для различных задач. Все еще существует много ситуаций, когда высокоструктурированная база данных *SQL* предпочтительнее, даже если базы данных *NoSQL* набирают популярность благодаря своей скорости и масштабируемости.
Преимущества:

1. *ACID* - улучшает и защищает целостность данных. *NoSQL* базы данных часто жертвуют *ACID* в пользу производительности и гибкости системы
2. Данные остаются неизменными и структурированными. Если команда разработчиков не видит массового роста, для которого потребуется больше серверов, и работа идет только с согласованными данными, то, вероятно, нет причин использовать систему, предназначенную для поддержки большого объема трафика и огромного количества типов данных.
3. Инструменты поставляются с улучшенной поддержкой, наборами продуктов и надстройками для управлением базами данных.

Недостатки: Хотя масштабируемость обычно тестируется в производственных средах, она часто ниже, чем у баз данных *NoSQL*. *Sharding* также довольно проблематичен.

NoSQL

Если необходимо работать с огромным количеством неструктурированных данных и требование к этим данным не ясны, лучшим выбором будет использовать документно-ориентированные хранилища. Благодаря этому система будет обладать наиболее высокой гибкостью, чем традиционные аналоги.
Преимущества:

1. Данный вид баз данных гарантируют, что данные не станут, узким местом, когда все другие компоненты серверного приложения спроектированы так, чтобы быть бесперебойными и быстрыми.

2. Хранение большого объема данных практически без какой-либо структуры. Данный вид базы данных не устанавливает никаких ограничений на типы хранимых данных, что позволяет добавлять новый типы данных, по мере необходимости.
3. Хранение данных без предварительного определение схем хранения данных.
4. Легкая масштабируемость и распределение на кластере.
5. Не нужно заранее подготавливать данные *NoSQL*. Нереляционная природа базы данных *NoSQL* позволяет быстро создавать базу данных без необходимости разработки подробной модели базы данных, что экономит вам много времени на разработку.

Недостатки:

1. Относительно *SQL* баз данных, у *NoSQL* слишком маленькое сообщество, которое нуждается в опытных кадрах.
2. Основная проблема - отсутствие инструментов отчетности для тестирования и анализа производительности.
3. Отсутствует стандартизация. Существует множество *NoSQL* баз данных, но до сих пор нет стандартов. Такое отсутствие может вызвать проблемы с миграцией.

Вывод:

Базы данных *NoSQL* становятся основной частью хранилищ данных сегодня. Благодаря их преимуществам, они могут стать настоящим переломным моментом в сфере информационных технологий. Если необходимо интегрировать большие данные, привлекательной опцией является низкая стоимость, более легкая масштабируемость и функции с открытым исходным кодом *NoSQL*. Несмотря на это, *NoSQL* оказывается относительно молодой технологией без набора стандартов баз данных *SQL*, таких как *MySQL*. Однако нужно понимать, выбор той или иной технологии всегда зависит от потребностей системы. В моем случае архитектура системы - микросервисная, поэтому перед постройкой инфраструктуры в рамках каждого отдельного сервиса, проблема выбора базы данных будет подниматься каждый раз. Основными показателями будут:

производительность операций чтения и записи, а также связанность и структурированность данных.

Сравнив все необходимые сервисные решения можно перейти к анализу инструментов и технологий, которые будут применены в проектировании информационной системы.

GraphQl или *Rest API*

GraphQL часто ошибочно считают заменой *REST*. *GraphQL* - это более новая концепция разработки серверных интерфейсов работы, опубликованная компанией *Facebook*. *Rest* является архитектурной концепцией для сетевого программного обеспечения, не имеет официального набора инструментов, не имеет спецификаций. Предназначен только для отделения *API* от клиента.

GraphQL - это язык запросов, спецификация и набор инструментов, предназначенных для работы через единую конечную точку *HTTP*/*HTTPS* и оптимизирующих производительность и гибкость. Одним из основных аспектов *REST* является использование унифицированного интерфейса протоколов, в которых он существует. При использовании *HTTP REST* может использовать типы содержимого *HTTP*, кэширование, коды состояния и т. д., тогда как *GraphQL* создает свои собственные соглашения.

Если *API* не использует элементы управления гипермедиа, то *GraphQL* может быть более подходящим решением. Одной из наиболее распространенных задач, предоставляемых *REST API*, является *CRUD* через *JSON*, но он может сделать гораздо больше, например, выгрузка файлов.

Загрузка изображения в теле *HTTP* может выглядеть примерно так:

POST /*images HTTP*/1.1

Host: *host*:8080

Content-Type: *image*/*png*

Content-Length: 284

raw image content

Используя замечательную часть *HTTP* и, следовательно, *REST* разработчики *API* могут поддерживать запросы *application* / *json* на одной и той же конечной точке,

чтобы обрабатывать загрузку немного по-другому, и предлагать также загрузки на основе *URL*:

POST /*images HTTP*/1.1

Host: *host*:8080

Content-Type: *application*/*json*

{

"*image_url*" : "*https*://*domain*/*image.png*"

}

Если говорить о загрузке видео или других больших файлов, стоит переключиться на другой подход и иметь специальный сервис, который обрабатывает загрузку, оставляя основной *API*-интерфейс только для приема метаданных; заголовок, описание, теги и т. д. Это подход, который нужно использовать - *GraphQL*, так как он позволяет общаться с сервисом бизнес-логики только через поля:

POST /*graphql HTTP*/1.1

Host: *host*:3001

Content-Type: *application*/*graphql*

mutation addImage {

addImageFromUrl(*image_url*: «*https*://*domain*/*image.png*") {

id,

game_id,

image_url

}

}

Существует утверждение, что это более «чисто», но принуждение к созданию другого сервиса излишне для небольших изображений, особенно на ранних этапах. Другой подход заключается в прямой загрузке на сторонний сервис, что вызывает зависимость от клиентов и может привести к утечке ваших ключей безопасности, или придется использовать множественную загрузку, что является затруднительным подходом, который зависит от того, сможет ли сервер и различные клиенты его поддерживать. Это одна из областей, где *REST* остается

сильным. Можно предположить, что *REST*, обрабатывающий *CRUD* и произвольные вещи, сбивает с толку, но это основной принцип того, что делает *REST* таким полезным.

GraphQL и *REST* должны создавать версии с течением времени из-за процесса старения и неконсистентности данных, *GraphQL* помогает разработчикам *API*, когда дело доходит до устаревших версий.

Одна из областей, в которой *GraphQL* превосходит другие - это невероятное удобство мониторинга на техническом уровне. Клиенты *GraphQL* вынуждены указывать поля, которые они хотят вернуть в запросе:

```
POST /graphql HTTP/1.1
Host: host:3001
Content-Type: application/graphql
{
  games(id: "1") {
    images,
    votes,
    result
  }
}
```

Отслеживать это тривиально, но *REST API* работает немного по-другому. Хотя все *API REST* делают базовую конечную точку доступной через / *games* / 1, не все *API* предлагают разреженные наборы полей: / *games* / 1? *Fields* = *images*, *votes*, *result*. Из тех, кто предлагает это, это почти всегда необязательно и непроизводительно.

GraphQL упрощает отслеживание использования определенных полей клиентом, а это означает, что владельцы *API* могут предоставлять доступ только тем клиентам, которые используют определенные заголовочные поля. Также *GraphQL* всегда является наименьшим возможным запросом, в то время как *REST* по умолчанию является наиболее полным. Обычная практика - предлагать такие опции, как? *Fields* = *foo*, *bar* или *partials*.

Рекомендации по созданию *REST API*:

Официальные рекомендации к построению интерфейсов на *GraphQL* отсутствуют.

Даже если по умолчанию *API REST* возвращает только базовый ресурс, он требует большее количество памяти, чем при использовании подхода *GraphQL*. Если клиенту требуется поле, он запрашивает его, а если *API* добавляет новое поле, клиенты не получают его, если только они не обнаружат это поле в документации и не добавят его в запрос *GraphQL*.

Подводя итог из проанализированного материала, следует учесть несколько факторов, которые оказывают большую часть влияние на выбор технологии:

1. Стоит учитывать масштаб серверного приложения, количество ресурсов, которое должно обслуживаться сервисом
2. Необходимо обратить внимание на инфраструктуру всей информационной системы

Фреймворки.

Также обязательным этапом в выборе технологии является отчетливое понимание и распределение вычислительных ресурсов системы.

В настоящее время существует множество фреймворков, написанных на языке программирования *PHP*. В качестве основы были выбраны наиболее известные и высокопроизводительные:

1. *Laravel*
2. *Phalcon*
3. *Symphony*
4. *CodeIgniter*
5. *CakePHP*
6. *Zend*
7. *FuelPHP*
8. *Slim*
9. *Yii* 2

Laravel

Laravel – это инструмент, использующийся для быстрого создания приложений с использованием архитектуры *MVC*. В настоящее время *Laravel* является самой популярной средой *PHP* с огромным сообществом разработчиков.

Он включает в себя множество специфичных для *laravel* пакетов, облегченный шаблонизатор *Blade*, модульное тестирование, *ORM*, систему пакетов, контроллеры *RESTful*, и теперь *Laravel* является первой структурой, которая представляет маршрутизацию абстрактным образом. Она устраняет трудности, связанные с организацией кода.

Управление очередями также является функцией, которая обрабатывает задачи в фоновом режиме, а затем регистрирует действия для всех вас, в то время как задачи обычно выполняются во внешнем интерфейсе. Пакеты могут быть легко добавлены с помощью надежного *Composer*, встроенного в *Laravel*. Он интегрируется с *Gulp* и *Elixir*, поэтому любые пакеты *npm* и пакеты *bower* могут вызываться напрямую через *sh*.

Одной из лучших вещей, с которыми хорошо справляется *Laravel*, являются структуры *NoSQL*, такие как *MongoDB* или *Redis*. Начать работу с *Laravel* легко благодаря его обширной документации, популярности и *Laravel Udemy*: популярные видео и учебные пособия предназначены для того, чтобы дать начинающим разработчикам *Laravel*.

Phalcon

Основаннай на *MVC PHP*-фреймворк, уникально построенная как *C*-расширение, что означает, что она работает абсолютно быстро. *Phalcon* использует очень мало ресурсов по сравнению с другими средами, что приводит к очень быстрой обработке *HTTP*/*HTTPS*-запросов, что может быть критично для разработчиков, работающих с системами, которые не требуют больших накладных расходов.

Phalcon активно развивается с 2012 года и включает в себя компоненты *ORM*, *MVC*, кэширование и автозагрузку. Его последний и первый долгосрочный выпуск поддержки включает поддержку *PHP* 7.

Phalcon предлагает разработчикам инструменты для хранения данных *Object Document Mapping* для *MongoDB*. Другие функции включают в себя механизмы шаблонов, конструкторы форм, простоту создания приложений с поддержкой интернационализации и многое другое. *Phalcon* идеально подходит как для создания высокопроизводительных *API*-интерфейсов, так и для полноценных систем.

CodeIgniter

Codeigniter является идеальной основой для быстрой разработки приложений. Это легкая и простая в использовании платформа с небольшим размером, которую можно установить, просто загрузив ее прямо на сервер. Никакой специальной командной строки или установки программного обеспечения не требуется.
Говоря о развитии фреймворка, документация *Codeigniter* понятна и легка в изучении, а сообщество обширно и очень полезно. *Code Igniter* поддерживается также академической структурой: *The British Columbia Institute of Technology*, который поможет обеспечить его дальнейшее развитие и рост.
Функционально *Codeigniter* поставляется со множеством встроенных библиотек для модульного тестирования, проверки формы, электронной почты, сеансов и многого другого. Если найти нужную библиотеку не представляется возможным, легко создать свою собственную, а затем поделиться ей с сообществом.

Symfony

Symfony - очень стабильный, хорошо документированный и модульный проект.

Cakephp

Cakephp - это идеальная среда для начинающих и для быстро развивающихся коммерческих приложений. Он поставляется с функциональностью генерации кода и скаффолдинга для ускорения процесса разработки, а также предоставляет тонны пакетов для обеспечения общей функциональности.
Он уникален тем, что имеет соглашения *MVC*, которые помогают направлять процесс разработки. Конфигурация также очень проста, поскольку устраняет

необходимость в сложных файлах конфигурации *XML* или *YAML*. Сборка выполняется быстро, а ее функции безопасности включают меры по предотвращению *XSS*, *SQL*-инъекций, *CSRF* и инструменты для проверки формы.

CakePHP находится в стадии активной разработки с хорошей документацией и множеством порталов поддержки, которые помогут начать работу. Премиум поддержка также является опцией для разработчиков, которые решили использовать *Cakephp* через *Cake Development Corporation*.

Таблица 3 - Сравнение фреймворков

Фреймворк	Плюсы	Минусы	Версия *PHP*
Laravel	Внутренняя структура и огранизация файлов. Быстрая скорость разработки приложения. *MVC* архитектура. *Unit* тестирование. Доступность документации. Высокий уровень абстракции. Собственная *ORM*. Сильная поддержка пакетов шифрования и информационной безопасности.	Неоптимизированность *ORM*.	5.5.9
Symphony	Высокая производительность. Кэширование байт-кода. Стабильность.	Отсутствует поддержка *MVC*.	5.5.9

	Доступность документации. Хорошая поддержка.		
CodeIgniter	Низкий порог вхождения. Превосходит большинство фреймворков (не *MVC*). Доступность документации. *LTS*.	На данный момент не ввели поддержку пространства имен. Труднотестируемый. Плохая инфраструктура.	5.4

Также стоит рассмотреть выбор создания своего собственного окружения для разработки приложения.

ОРГАНИЗАЦИЯ ТЕХНОЛОГИЧЕСКОГО КОНВЕЙЕРА

Организация технологического конвейера играет важную роль в сопровождении информационной системы.

Что же такое *CI* ? Непрерывная интеграция, *CI/CD* или непрерывная доставка. Это концепция, которая проходит под многими именами, но охватывает те же основные идеи. В нем изложены некоторые правила, которым нужно следовать, чтобы код, который пишут программисты, быстрее и безопаснее доходил до пользователей приложения и в конечном итоге создавал ценность.

CI: непрерывная интеграция

Часть *CI CI/CD* может быть кратко изложена следующим образом: если есть необходимость, чтобы все части того, что входит в приложение, отправлялось в одно и то же место и проходило одни и те же процессы проверки и подготовки, а также публиковалось в удобном для доступа месте.

Простейший пример непрерывной интеграции - это то, что по началу не представляется значимым: фиксация всего кода вашего приложения в одном репозитории. Хотя это может показаться легким делом, наличие единого места, где «интегрируется» весь код, являющийся основой для расширения других, более продвинутых сервисов.

После того, как весь код приложения и изменения собраны в одном месте, можно запускать некоторые процессы в этом хранилище каждый раз, когда что-то меняется. Это может включать в себя:

Запуск автоматического сканирование качества кода и создание отчета о том, насколько хорошо ваши последние изменения соответствуют хорошей практике написания кода.

Создание кода и запуск автоматизированных тестов, которые были написаны, чтобы убедиться в целостности и в том, что изменения не нарушают какую-либо функциональность.

Создание и публикация отчета о покрытии тестов.

Эти простые дополнения позволяют разработчику сосредоточиться на написании кода. Репозиторий кода предназначен для получения изменений, в то время

как автоматизированные процессы могут создавать, тестировать и сканировать код, предоставляя отчеты.

CD: непрерывное развертывание

Развертывание кода может быть сложным. Существует много свободно доступных инструментов, которые позволяют сделать это легко. Одним из популярных примеров является *Travis CI, Concourse CI*, которые напрямую интегрируется с *Github*. Также можно настроить *Concourse* для автоматического запуска задач *CI*, таких как модульные тесты, и передавать код информационной системы на платформу хостинга.

ПРОГРАММНАЯ РЕАЛИЗАЦИЯ

Проанализировав и сравнив преимущества и недостатки различным сервисных технология и инструментов разработки, был сделан выбор в пользу микросервисной архитектуры, которая позволит избавиться от проблем доставки кода, а также поможет распределить создаваемую нагрузку более правильно между серверами.

В качестве инструмента разработки для микросервиса, обрабатывающего запросы в формате *GraphQL,* был выбран язык программирования *PHP* и фреймворк, предназначенный для этой цели – *Railt.*

Для написание различных инфраструктурных сервисов, такие как: система оркестрации контейнерами *LXC*, распределитель нагрузки, прокси сервер, собственная *NoSQL* база данных, для хранения системной информации и т.д. был выбран язык программирования Golang.

Учитывая архитектуру всей системы, необходимо было продумать взаимодействие всех микросервисов друг с другом. Самое оптимальное решение - это создание процессов-демонов, которые в фоновом режиме работают с брокером очередей получают оттуда необходимую для работы информацию. В качестве брокера очередей выступил *RabbitMQ*.

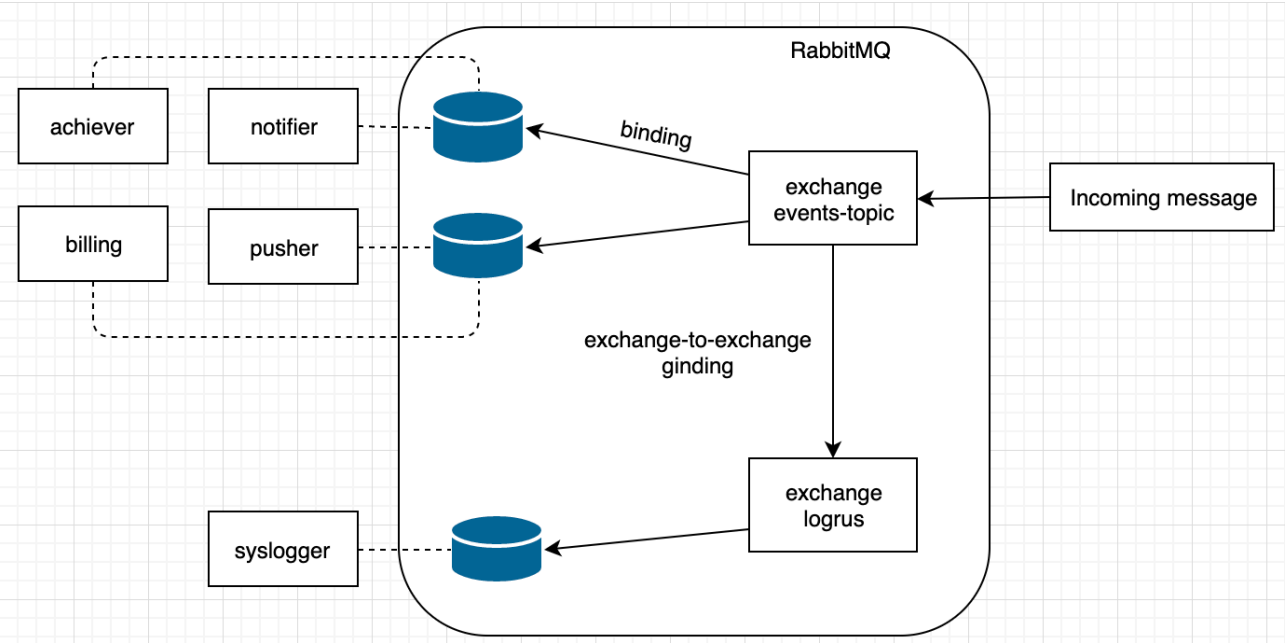


Рисунок 8 - Структура RabbitMQ

На рисунке 8 можно наблюдать устройство брокера очередей, в котором были установлены 2 *exchange,* а так же добавлены связи с очередями (т.к. название очередей не несет практической пользы, они скрыты). *Events-topic exchange* используется для взаимодействия основных сервисов информационной системы: *notifier, pusher, billing, achiever. Incoming message* – входящее сообщение от сервиса, обрабатывающего запросы *GraphQL*, которое распределяется с помощью exchange и попадет в нужную очередь для последующей обработки определенным сервисом.

Структура сервиса обработки запросов *GraphQL* очень схожа со структурами других микросервисов, которые содержат бизнес-логику, однако есть небольшие различия.

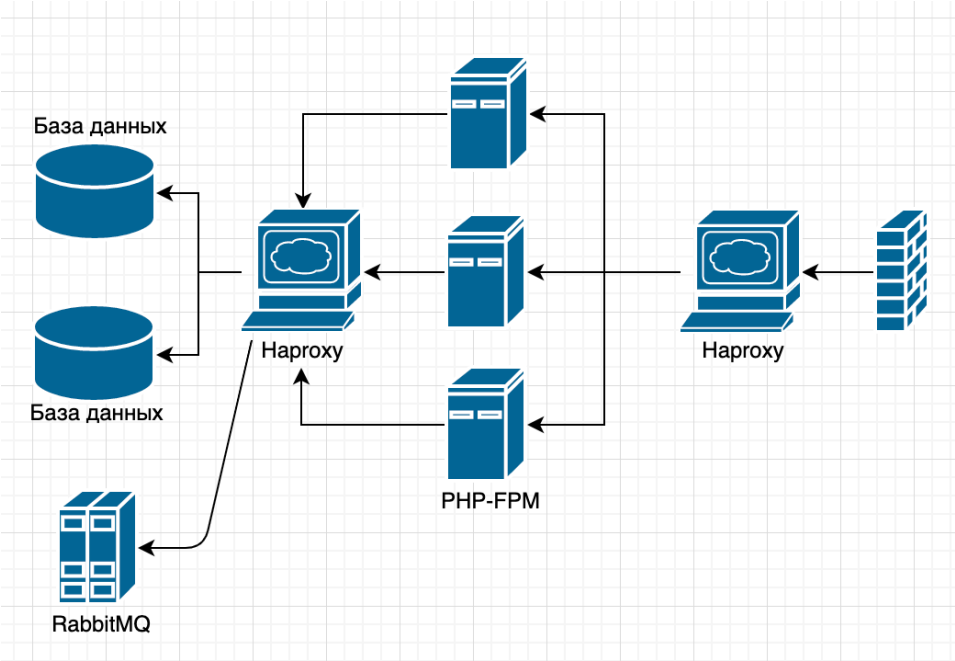


Рисунок 9 - Структура GraphQL сервиса

На рисунке 9 представлена структура микросервиса *GraphQL*. *Haproxy* – балансировщик нагрузки, который определяет на какой сервер отправить запрос. В данной структуре *haproxy* решает проблему подключения и перенаправления запроса на входе, между 3 экземплярами *PHP-FPM* и внутри самого микросервиса, между 2 базами данных *PostgreSQL* и определяет *IP*-адрес *RabbitMQ* сервера.

Как было описано ранее, структура микросервиса не сильно отличается от структуры других микросервисов, что очень хорошо видно в сравнении с сервисов отправки *PUSH*-уведомления.

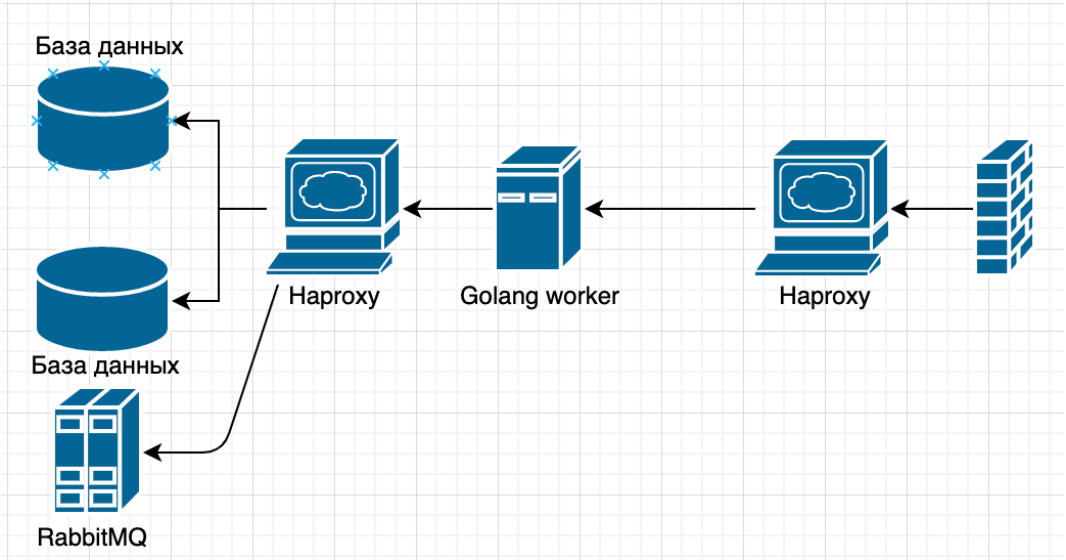


Рисунок 10 - Структура PUSH сервиса

Единственным отличием сервисов является то, как устроен процесс, в котором обрабатывается бизнес-логика. В данном случае – *Golang worker* – процесс запущенного бинарного файла, написанного на языке *Golang*. Также процессы-демоны, написанные на языке *Golang* не требуют горизонтального масштабирования, в отличии от *PHP*, т.к. они поддерживают параллельность.

Самое главной частью в приложении является сервер *GraphQL*, в котором содержится вся бизнес-логика информационной системы. Конфигурирование системы начинается с составления файла-схемы для запросов от клиентов. Схема содержит в себе несколько разделов: *query* и *mutations*. *Query* описывает все сущности, которые используются в информационной системе, а также операции поиска, которые можно будет выполнять, например, фильтры и пагинация. *Mutations* описывает функционал изменения сущностей, в том числе и удаление, описанных в разделе *Query.*

После описания всех сущностей системы и возможных с ними операций, для работы системы необходимо создать «резолверы», которые будут работать с разобранным запросом от клиента в формате GraphQL и проводить необходимые операции загрузки информации для возврата клиенту результата, который он ожидает.

Нельзя забывать о безопасности, поэтому необходимо создать систему аутентификации, чтобы доступ был только у доверенных клиентов. Для этой цели

подойдет *OAuth 2.0* – сервер аутентификации клиентов, посредством создания аутентификационных токенов.

Финальным этапом в построении *GraphQL* сервиса является тестирование. Поскольку, со временем, сервис растет, возникает проблема в ручном тестировании, т.к. объем кода и функционала увеличивается. Для этого необходимо проводить автоматическое тестирование системы, с помощью написания различных тестов.

ТЕСТИРОВАНИЕ

Тесты программного обеспечения можно разделить на две группы: функциональные тесты и нефункциональные.

Функциональные тесты делятся на:

- *Unit* тесты
- Интеграционные тесты
- Регрессионные тесты
- *Smoke* тестирование
- Тестирование интерфейсов
- *Sanity* тестирование

Нефункциональные тесты делятся на:

- Тесты на производительность
- Нагрузочные тесты
- Стресс-тесты
- Тесты на безопасность
- Тесты на совместимость
- *Recovery* тесты
- *Usability* тесты
- Тесты локализации

Альфа-тестирование

Это наиболее распространенный тип тестирования, используемый в индустрии программного обеспечения. Целью этого тестирования является выявление всех возможных проблем или дефектов перед выпуском его на рынок или для пользователя.

Альфа-тестирование проводится в конце фазы разработки программного обеспечения, но до бета-тестирования. Тем не менее, незначительные изменения могут быть сделаны в результате такого тестирования. Для этого типа тестирования может быть создана внутренняя виртуальная пользовательская среда.

Приемочные испытания

Приемочный тест выполняется клиентом и проверяет, соответствует ли конец потока системы бизнес-требованиям или нет, и соответствует ли он потребностям конечного пользователя. Клиент принимает программное обеспечение только тогда, когда все функции и возможности работают должным образом. Это последний этап тестирования, после которого программное обеспечение запускается в производство. Это также называется приемочным тестированием пользователя.

Свободное тестирование

Само название предполагает, что это тестирование проводится на специальной основе, то есть без ссылки на контрольный пример, а также без какого-либо плана или документации для такого типа тестирования. Цель этого тестирования - найти дефекты и сломать приложение, выполнив любой поток приложения или любую случайную функциональность.

Специальное тестирование является неофициальным способом поиска дефектов и может быть выполнено любым участником проекта. Трудно идентифицировать дефекты без тестового случая, но иногда возможно, что дефекты, обнаруженные во время специального тестирования, могли не быть идентифицированы с использованием существующих тестовых случаев.

Тестирование доступности

Цель тестирования доступности - определить, доступно ли программное обеспечение или приложение.

Бета-тестирование

Бета-тестирование - это формальный тип тестирования программного обеспечения, которое проводится заказчиком. Он выполняется в реальной среде перед выпуском продукта на рынок для реальных конечных пользователей.

Бета-тестирование проводится, чтобы убедиться в отсутствии серьезных сбоев в программном обеспечении или продукте, и оно удовлетворяет бизнес-требованиям с точки зрения конечного пользователя. Бета-тестирование считается успешным, когда клиент принимает программное обеспечение.

Обычно это тестирование обычно выполняется конечными пользователями или другими лицами. Это окончательное тестирование перед выпуском приложения для коммерческих целей. Обычно бета-версия выпущенного программного обеспечения или продукта ограничена определенным количеством пользователей в определенной области.

Таким образом, конечный пользователь фактически использует программное обеспечение и делится обратной связью с компанией. Затем компания предпринимает необходимые действия перед выпуском программного обеспечения по всему миру.

Внутреннее тестирование

Всякий раз, когда ввод или данные вводятся в интерфейсное приложение, они сохраняются в базе данных, и тестирование такой базы данных называется тестированием базы данных или внутренним тестированием. Существуют различные базы данных, такие как *SQL Server*, *MySQL*, *Oracle* и т. д. Тестирование базы данных включает в себя тестирование структуры таблицы, схемы, хранимой процедуры, структуры данных и так далее.

В бэкэнд-тестировании *GUI* не задействован, тестеры напрямую подключены к базе данных с соответствующим доступом, и тестеры могут легко проверить данные, выполнив несколько запросов к базе данных. Во время этого внутреннего тестирования могут быть выявлены такие проблемы, как потеря данных, взаимоблокировка, повреждение данных и эти проблемы крайне важны для исправления до того, как система перейдет в производственную среду.

Тестирование обратной совместимости

Это тип тестирования, который проверяет, работает ли недавно разработанное программное обеспечение или обновленное программное обеспечение с более старой версией среды или нет.

Тестирование обратной совместимости проверяет, работает ли новая версия программного обеспечения должным образом с форматом файла, созданным более старой версией программного обеспечения; он также хорошо работает с таблицами данных, файлами данных, структурой данных, созданной более старой версией этого программного обеспечения. Если какое-либо программное

обеспечение обновляется, оно должно хорошо работать поверх предыдущей версии этого программного обеспечения.

Тестирование граничных значений

Этот тип тестирования проверяет поведение приложения на граничном уровне.

Отраслевое тестирование

Это тип тестирования белого ящика, который проводится во время модульного тестирования. Тестирование ветвей, само название предполагает, что код тщательно тестируется путем обхода в каждой ветке.

Сравнительное тестирование

Сравнение сильных и слабых сторон продукта с его предыдущими версиями или другими аналогичными продуктами называется сравнительным тестированием.

Тестирование совместимости

Это тип тестирования, в котором проверяется, как программное обеспечение ведет себя и работает в другой среде, веб-серверах, оборудовании и сетевой среде. Тестирование на совместимость гарантирует, что программное обеспечение может работать в другой конфигурации, другой базе данных. Тестирование на совместимость выполняется командой тестирования.

Тестирование компонентов

В основном это выполняется разработчиками после завершения модульного тестирования. Компонентное тестирование включает в себя тестирование нескольких функциональных возможностей как единого кода, и его цель состоит в том, чтобы определить, существует ли какой-либо дефект после соединения этих многочисленных функциональных возможностей друг с другом.

Сквозное тестирование

Подобно системному тестированию, сквозное тестирование включает в себя тестирование всей среды приложений в ситуации, которая имитирует реальное использование, например: взаимодействие с базой данных, использование сетевых коммуникаций или взаимодействие с другим оборудованием, приложениями или системами.

Эквивалентное разбиение

Это метод тестирования и тип тестирования черного ящика. Во время этого разделения эквивалентности выбирается набор групп, и для тестирования выбираются несколько значений или чисел. Понятно, что все значения из этой группы генерируют одинаковый результат.

Целью этого тестирования является удаление избыточных тестовых случаев в конкретной группе, которая генерирует тот же результат, но не дефект.

Предположим, что приложение принимает значения в диапазоне от -10 до +10, поэтому, используя эквивалентное разбиение, значения, выбранные для тестирования, равны нулю, одному положительному значению, одному отрицательному значению. Таким образом, Эквивалентное разбиение для этого теста: -10 до -1, 0 и от 1 до 10.

Функциональное тестирование

Этот тип тестирования игнорирует внутренние детали и фокусируется только на выводе, чтобы проверить, соответствует ли он требованиям или нет. Это тестирование типа «черного ящика», ориентированное на функциональные требования приложения.

Тестирование инкрементной интеграции

Инкрементное интеграционное тестирование - это восходящий подход к тестированию, то есть непрерывное тестирование приложения при добавлении новой функциональности. Функциональные возможности и модули приложения должны быть достаточно независимыми, чтобы тестировать отдельно. Это делается программистами или тестировщиками.

Тестирование установки-удаления

Тестирование установки и удаления выполняется при полном, частичном или обновленном процессе установки / удаления в разных операционных системах в разной аппаратной или программной среде.

Интеграционное тестирование

Тестирование всех интегрированных модулей для проверки объединенной функциональности после интеграции называется интеграционным

тестированием. Модули обычно представляют собой программные модули, отдельные приложения, клиентские и серверные приложения в сети и т. д. Этот тип тестирования особенно важен для клиент-серверных и распределенных систем.

Нагрузочное тестирование

Это тип нефункционального тестирования, и целью нагрузочного тестирования является проверка того, какую нагрузку или максимальную рабочую нагрузку может выдержать система без какого-либо снижения производительности. Нагрузочное тестирование помогает определить максимальную емкость системы при определенной нагрузке и любых проблемах, которые вызывают снижение производительности программного обеспечения. Нагрузочное тестирование выполняется с использованием таких инструментов, как *JMeter*, *LoadRunner*, *WebLoad*, *Silk Performer* и т. д.

Тестирование мутаций

Мутационное тестирование - это тип тестирования белого ящика, в котором изменяется исходный код одной из программ и проверяется, могут ли существующие тестовые примеры идентифицировать эти дефекты в системе. Изменения в исходном коде программы очень минимальны, так что они не влияют на все приложение, только конкретная область, имеющая влияние, и соответствующие тестовые примеры должны быть в состоянии идентифицировать эти ошибки в системе.

Отрицательное тестирование

Тестировщики, настроенные на «отношение к разрыву» и использующие отрицательное тестирование, проверяют это в случае отказа системы или приложения. Техника отрицательного тестирования выполняется с использованием неверных данных, неверных данных или ввода. Это подтверждает, что если система выдает ошибку неправильного ввода и ведет себя как ожидалось.

Нефункциональное тестирование

Это тип тестирования, для которого в каждой организации есть отдельная команда, которая обычно называется командой по нефункциональному тестированию (*NFT*) или командой по производительности.

Нефункциональное тестирование включает в себя тестирование нефункциональных требований, таких как нагрузочное тестирование, стресс-тестирование, безопасность, объем, тестирование восстановления и т. д. Цель тестирования - убедиться, достаточно ли быстрое время отклика программного обеспечения или приложения в соответствии с бизнес-требованием.

Загрузка любой информации или системы не должна занимать много времени и должна выдерживать пиковую нагрузку.

Тестирование производительности

Этот термин часто используется взаимозаменяемо с «стресс» и «нагрузочным» тестированием. Тестирование производительности проводится для проверки соответствия системы требованиям к производительности. Для этого тестирования используются разные инструменты производительности и нагрузки.

Тестирование восстановления

Это тип тестирования, который проверяет, насколько хорошо приложение или система восстанавливается после сбоев или аварий.

Тестирование восстановления определяет, сможет ли система продолжить работу после аварии. Предположим, что приложение получает данные через сетевой кабель, и внезапно этот сетевой кабель был отключен.

Через некоторое время подключите сетевой кабель; затем система должна начать получать данные с того места, где она потеряла соединение из-за отсоединения сетевого кабеля.

Регрессионное тестирование

Тестирование приложения в целом на наличие изменений в любом модуле или функциональности называется регрессионным тестированием. Трудно охватить всю систему в регрессионном тестировании, поэтому обычно для этих типов тестирования используются инструменты автоматического тестирования.

Тестирование на основе риска

В тестировании на основе рисков функциональные возможности или требования проверяются в зависимости от их приоритета. Тестирование на основе рисков включает в себя тестирование критически важных функций, которые оказывают наибольшее влияние на бизнес и в которых вероятность сбоя очень высока.

Решение о приоритете основывается на бизнес-потребностях, поэтому, как только приоритет установлен для всех функций, сначала выполняются функции с высоким приоритетом или тестовые примеры, за которыми следуют функции со средним, а затем низким приоритетом.

Функциональность с низким приоритетом может быть протестирована или не протестирована в зависимости от доступного времени.

Тестирование на основе рисков проводится, если для тестирования всего программного обеспечения недостаточно времени, и программное обеспечение необходимо внедрить вовремя без каких-либо задержек.

Тестирование безопасности

Это тип тестирования, выполняемый специальной командой тестировщиков. Тестирование безопасности проводится для проверки того, насколько программное обеспечение, приложение или веб-сайт защищены от внутренних и внешних угроз. Это тестирование включает определение степени защиты программного обеспечения от вредоносных программ, вирусов, а также степени защищенности и надежности процессов авторизации и аутентификации.

Он также проверяет, как программное обеспечение ведет себя при хакерских атаках и вредоносных программах и как программное обеспечение поддерживается для защиты данных после такой хакерской атаки.

Smoke тестирование

Всякий раз, когда группа разработчиков предоставляет новую сборку, группа тестирования программного обеспечения проверяет сборку и гарантирует, что существенных проблем не существует.

Команда тестирования гарантирует, что сборка стабильна, и в дальнейшем проводится подробный уровень тестирования. *Smoke Testing* проверяет, что в

сборке нет дефекта пробки, что не позволит группе тестирования детально протестировать приложение.

Если тестировщики обнаружат, что основная критическая функциональность нарушена на самом начальном этапе, тогда команда тестирования может отклонить сборку и сообщить об этом команде разработчиков. Smoke тестирование проводится на детальном уровне любого функционального или регрессионного тестирования.

Статическое тестирование

Статическое тестирование - это тип тестирования, который выполняется без какого-либо кода. Исполнение выполняется по документации на этапе тестирования. Он включает в себя обзоры, пошаговое руководство и проверку результатов проекта. Статическое тестирование не выполняет код вместо синтаксиса кода, соглашения об именах проверяются.

Статическое тестирование также применимо для тестовых случаев, плана тестирования, проектной документации. Необходимо выполнить статическое тестирование группой тестирования, поскольку дефекты, выявленные в ходе этого типа тестирования, являются экономически эффективными с точки зрения проекта.

Стресс-тестирование

Это тестирование проводится, когда система подвергается нагрузке за пределы своих спецификаций, чтобы проверить, как и когда она выходит из строя. Это выполняется при большой нагрузке, например, вывод большого количества данных за пределы емкости хранилища, сложные запросы к базе данных, постоянный ввод в систему или загрузка базы данных.

Тестирование системы

В соответствии с методикой тестирования системы вся система тестируется в соответствии с требованиями. Это типовое тестирование черного ящика, основанное на общих требованиях и охватывающее все объединенные части системы.

Модульное тестирование

Тестирование отдельного программного компонента или модуля называется модульным тестированием. Обычно это делается программистом, а не тестировщиками, так как требует подробного знания внутреннего дизайна программы и кода. Также может потребоваться разработка тестовых модулей драйверов или тестовых систем.

Объемное тестирование

Массовое тестирование - это тип нефункционального тестирования, выполняемого группой тестирования производительности.

Программное обеспечение или приложение подвергаются огромному количеству данных, и при проведении объемного тестирования проверяется поведение системы и время отклика приложения, когда система сталкивается с таким большим объемом данных. Такой большой объем данных может повлиять на производительность системы и скорость обработки.

Тестирование белого ящика

Тестирование белого ящика основано на знании внутренней логики кода приложения.

Вышеупомянутые типы тестирования программного обеспечения являются лишь частью тестирования. Тем не менее, есть список из более чем 100 типов тестирования, но не все типы тестирования используются во всех типах проектов. В данной информационной системе использованы некоторые распространенные типы тестирования программного обеспечения, которые в основном используются в жизненном цикле тестирования:

- сквозное тестирование
- функциональное тестирование
- нагрузочное тестирование

Кроме того, существуют альтернативные определения или процессы, используемые в разных организациях, но основная концепция везде одинакова. Эти типы тестирования, процессы и методы их реализации постоянно меняются по мере изменения проекта, требований и области применения.

В ходе тестирования информационной системы были написаны функциональные тесты, клиентов и веб-сокет серверов.

```
package websocket
import (
	"net/url"
	"testing"
)
var hostPortNoPortTests = []struct {
	u                     *url.URL
	hostPort, hostNoPort string
}{
	{&url.URL{Scheme: "ws", Host: "example.com"}, "example.com:80", "example.com"},
	{&url.URL{Scheme: "wss", Host: "example.com"}, "example.com:443", "example.com"},
	{&url.URL{Scheme: "ws", Host: "example.com:7777"}, "example.com:7777", "example.com"},
	{&url.URL{Scheme: "wss", Host: "example.com:7777"}, "example.com:7777", "example.com"},
}
func TestHostPortNoPort(t *testing.T) {
	for _, tt := range hostPortNoPortTests {
		hostPort, hostNoPort := hostPortNoPort(tt.u)
		if hostPort != tt.hostPort {
			t.Errorf("hostPortNoPort(%v) returned hostPort %q, want %q", tt.u, hostPort, tt.hostPort)
		}
		if hostNoPort != tt.hostNoPort {
			t.Errorf("hostPortNoPort(%v) returned hostNoPort %q, want %q", tt.u, hostNoPort, tt.hostNoPort)
		}
	}
}
```

Данный тест проверяет правильность определение порта для сервера.

Также было проведено нагрузочное тестирование с помощью утилиты ab, которая позволяет указать нужное количество генерируемых запросов и количество одновременно параллельно создаваемых запросов. Запуск нагрузочного тестирования производился командой: `ab -n 10 -c 100 http://127.0.0.1/`

Отправив 100 запросов и распараллелив их на 10 поток были получены результаты (Рисунок 11):

```
Benchmarking 127.0.0.1 (be patient).....done


Server Software:        nginx/1.6.2
Server Hostname:        127.0.0.1
Server Port:            80

Document Path:          /
Document Length:        20 bytes

Concurrency Level:      10
Time taken for tests:   2.849 seconds
Complete requests:      100
Failed requests:        0
Total transferred:      18100 bytes
HTML transferred:       2000 bytes
Requests per second:    35.10 [#/sec] (mean)
Time per request:       284.877 [ms] (mean)
Time per request:       28.488 [ms] (mean, across all concurrent requests)
Transfer rate:          6.20 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.1      0       0
Processing:    63  271  60.7    274     381
Waiting:       63  271  60.7    273     381
Total:         63  271  60.7    274     381

Percentage of the requests served within a certain time (ms)
  50%    274
  66%    298
  75%    311
  80%    321
  90%    349
  95%    357
  98%    376
  99%    381
 100%    381 (longest request)
```

Рисунок 11 - Результаты нагрузочного тестирования API

ЗАКЛЮЧЕНИЕ

Исходя из установленной в начале работы цели, была спроектирована и реализована серверная часть информационной системы, а так же был проведен сравнительный анализ существующих аналогов, технических возможностей и характеристик различных микросервисов и серверных архитектур.

Выбранные технические сервисы помогли создать высокоустойчивую систему, способную выдержать большие нагрузки на сервер, и создали возможность масштабирования серверной части. Также были исследованы системы автоматической доставки и развертывания кода, что помогла сократить время на выкладку новых версий.

СПИСОК ЛИТЕРАТУРЫ

1. B. Fitzpatrick, "Distributed caching with memcached", *Linux journal*, vol. 2004, no. 124, pp. 5, 2004.
2. K. Banker, MongoDB in action, Manning Publications Co., 2011.
3. Oleksii Kononenko, Olga Baysal, Reid Holmes, Michael W Godfrey, "Mining modern repositories with elasticsearch", *Proceedings of the 11th Working Conference on Mining Software Repositories*, pp. 328-331, 2014.
4. Jun Bai, "Feasibility analysis of big log data real time search based on hbase and elasticsearch", *Natural Computation (ICNC) 2013 Ninth International Conference on*, pp. 1166-1170, 2013.
5. Clinton Gormley, Zachary Tong, Elasticsearch: The Definitive Guide, O'Reilly Media, Inc., 2015.
6. S. Newman, Building Microservices, O'Reilly Media Inc., 2015.
7. L. Bass, I. Weber, L. Zhu, DevOps: A Software Architect's Perspective, Addison-Wesley Professional, 2015.
8. *Digital Performance & Application Performance Monitoring | Dynatrace.*
9. E. Wolff, Microservices: Flexible Software Architecture, Addison-Wesley Professional, 2016.
10. Richardson Chris, "Microservice architecture pattern", *microservices.io*.
11. Рекс Блэк «Ключевые процессы тестирования», 2014
12. Калбертсон Роберт, Браун Крис, Кобб Гэри. Быстрое тестирование. — М.: «Вильямс», 2002. — 374 с.
13. Синицын С. В., Налютин Н. Ю. Верификация программного обеспечения. — М.: БИНОМ, 2008. — 368 с
14. Порселло Ева, Бэнкс Алекс, *GraphQL*, язык запросов для современного веб-приложения, *O'Reilly Media Inc.*, 2019
15. *IEEE Standard Test Access Port and Boundary-Scan Architecture,* 1990.
16. *T. P. Parker, C. W. Webb, "A Study of Failures Identified During Board Level Environmental Stress Testing", IEEE Transactions on Components Hybrids and Manufacturing Technology December* 1992.

17. *H. A. Chan, P. J. Englert, M. A. Oien, S. R. Rajaram, "Environmental Stress Testing", AT&T Technical Journal,* 1994.