Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования «Петрозаводский государственный университет»

Институт математики и информационных технологий

кафедра теорий вероятностей и анализа данных

(подпись cоискателя)

Куренчук Сергей Сергеевич

Выпускная квалификационная работа бакалавра

Разработка микросервиса для онлайн обработки данных в облаке на основе Spring Boot Framework

Направление 09.03.02 – Информационные системы и технологий

Научный руководитель:

Доктор технических наук, профессор А. А. Рогов

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись руководителя)

Петрозаводск — 2019

# Оглавление

[1. Обзор инструментов для реализаций микросервисной архитектуры. 4](#_Toc9964243)

[Docker 6](#_Toc9964244)

[Docker Swarm 7](#_Toc9964245)

[Развертывание сервисов 8](#_Toc9964246)

[Интеграция сервисов 8](#_Toc9964247)

[Discovery Service 8](#_Toc9964248)

[Key-value хранилище 9](#_Toc9964249)

[Consul 9](#_Toc9964250)

[Message broker 10](#_Toc9964251)

[RabbitMQ 11](#_Toc9964252)

[Работа с базами данных 12](#_Toc9964253)

[ORM 12](#_Toc9964254)

[JPA 12](#_Toc9964255)

[Hibernate 15](#_Toc9964256)

[Spring Framework 16](#_Toc9964257)

[Bean 17](#_Toc9964258)

[Описание Bean 17](#_Toc9964259)

[Spring Discovery Client 19](#_Toc9964260)

[Spring Data JPA 19](#_Toc9964261)

[Spring Profile 20](#_Toc9964262)

[Netflix Feign Client 21](#_Toc9964263)

[Spring Boot Framework 23](#_Toc9964264)

[2. Архитектура прототипа микросервиса 24](#_Toc9964265)

[Архитектура инфраструктуры 24](#_Toc9964266)

[Архитектура компонент прототипа 25](#_Toc9964267)

[Архитектура обработки запроса 27](#_Toc9964268)

[3. Практическая реализация 29](#_Toc9964269)

[Создание инфраструктуры на Google Cloud Platform 29](#_Toc9964270)

[Разработка микросервиса с помощью Spring 29](#_Toc9964271)

[Docker образ микросервиса 31](#_Toc9964272)

[Оценка работы прототипа 33](#_Toc9964273)

[Результаты тестирования 33](#_Toc9964274)

[Заключение 35](#_Toc9964275)

[Список литературы 36](#_Toc9964278)

**Введение**

За последнее время очень многое поменялось в сфере IT индустрии. Каждый день разрабатываются новые приложения использующие нынешние технологии для упрощения и автоматизации различных бизнес процессов. Будь то сложные математические вычисления, обработка статистический данных или рутинная работа по формированию отчетов. С новыми возможностями появляется активное их использования, а также повышаются нагрузки на конечную систему

В этой ситуации началось движение к масштабированию сервисов для распределения нагрузки. Это практика позволяет увеличить предел нагрузки. Другое дело, что не любую архитектуру приложения можно расширить. Зачастую сервисы разрабатывается как монолитное приложение, имеющие сильные связи с отдельными модулями. Что мешает масштабировать данное приложение.

Последние десять лет, при разработке серверного приложения рассматривают микросервисную архитектуру, которая не обладает этими недостатками. Микросервисная архитектура - разновидность Сервисно-ориентированной архитектуры. Приложение состоит из набора микросервисов. В отличие от монолитной архитектуры, такое решение позволяет разделить монолитный процесс и распределять нагрузку на несколько узлов. А также обеспечить границы разделяемой памяти. В сравнение с монолитным приложением, микросервис обладает следующими особенностями:

* Самостоятельность - микросервис максимально независим от других микросервисов
* Низкая связанность (Low Coupling) - изменение в микросервисе, помимо изменения интерфейса, никак не влияет на остальные компоненты системы. Поэтому такие микросервисы можно заменять на решения с идентичным интерфейсом
* Высокое зацепление (High Cohesion) - направленность микросервиса на решение определенной задачи, на ней он и сфокусирован

Существующая практика гласит, что микросервисы должны быть небольшими. Предполагая, что для разработки и поддержки потребуется маленькая команда. Учитывая, что микросервис — это самостоятельное приложение, то команда не обязана знать все нюансы системы.

Из описанного выше, можно сделать вывод о целесообразности использования микросервисной архитектуры для обработки независимых пакетов данных

Целью исследовательской работы является разработка прототипа микросервиса для обработки пользовательских изображений, с возможностью масштабирования. Для достижения цели следует выполнить следующие шаги:

1. Изучить микросервисную архитектуру.
2. Изучить работу современных инструментов для масштабирования.
3. Изучить инструменты для коммуникаций сервисов.
4. Разработать Архитектуру приложения.
5. Разработать прототип приложения.
6. Провести испытания.

В качестве инструмента разработки микросервиса был выбран современный универсальный фреймворк с открытым исходным кодом для Java-платформы Spring Boot Framework.

# 1. Обзор инструментов для реализаций микросервисной архитектуры.

Взаимодействие между сервисами чаще всего реализуют на основе HTTP, AMQP. При этом каналы связи не наделены какой-либо логикой. Она полностью перенесена на Конечные точки микросервисов. Такая архитектура приложения приводит к его децентрализациям.

Самостоятельность микросервисов позволяет для каждой задачи использовать эффективный набор инструментов. При разработке микросервисов могут применяться различные языки программирования, библиотеки, базы данных.

Такая самостоятельность приводит к децентрализаций хранения данных, что позволяет эффективно производить хранение, чтения, записи данных для конкретной задачи, но усложняет общую структуру. Но это приводит к необходимости использовать распределенные транзакций. Либо вовсе отказаться от постоянной согласованности данных

В отличие от монолитного приложения, запросы к сервисам проходят через сеть. А это может привести к задержкам, отказам. При разработке системы приходится думать о возможности отказов некоторых сервисов и о корректном поведении в этом случае других. Например, попытаться повторить запрос.

Рассмотрим положительные стороны микросервисной архитектуры относительно монолитного решения:

* Модульность.
* Ограниченный функционал.
* Модули могут быть реализованы с использованием различных языков программирования, фреймворков, связующего программного обеспечения.
* Модули могут быть распределены по различным контейнерам и серверам.
* Простота покрытия тестами.
* Очевидная зона ответственности разработчиков.
* Возможность масштабирования
* Децентрализация системы
* Общая отказоустойчивость системы
* Гибкая разработка

## Выделим отрицательные стороны использования микросервисов:

* Повышенная сложность разработки архитектуры
* Необходимость обработки ситуаций отказов при взаимодействии по сети
* Сложность взаимосвязей микросервисов
* Менее эффективное кэширование
* Дополнительная сложность в эксплуатации

Система, использующая микросервисы может содержать в себе большие компоненты. В таком случае микросервисы существую вокруг этих компонент. Эту архитектуру называют смешенной. Причин для ее существования может быть несколько:

* Работа с общими моделями системы
* Исторически сложившиеся решение при попытке перейти на микросервисную архитектуру

Для реализаций архитектуры необходимо учесть вопросы инфраструктуры, интеграций, общения сервисов между собой, работу с базой данных. В этой главе разобраны инструменты для решения этих вопрос. Ответственность за инфраструктуру в этой исследовательской работе отдана Docker Swarm. При этом существуют другие более функциональные инструменты оркестровки контейнеров и управления кластером, например, kubernetes. Docker Swarm был выбран за свою простоту и скорость развертывания. Для реализаций общения, интеграций и уменьшения связанности сервисов было выбрано две технологий: Discovery Service и Message broker. В этой главе рассмотрены их конкретные реализаций: Consul HashiCorp, RabbitMQ

Для создания микросервиса в исследовательской работе был выбран Spring Framework. В этой главе выделены и описаны инструменты для разработки микросервиса этого фреймворка.

## Docker

В современной разработке Docker - популярный инструмент создание, управления и развертывания контейнеров. В отличие от виртуальных машин, технология контейнеров используют ядро хостовой операционной системы. Что эффективнее в работе и потребляемой оперативной памяти контейнером.

Docker контейнеры рассчитаны на Linux ядро. В случае других сред происходит виртуализация Linux среды. Например, так было в операционной системе Windows. На текущий момент, после выпуска Windows 10 Redstone, произошло внедрение операционной системы Ubuntu. Это позволяет быстрее работать Docker контейнерам на этой системе.

Описание создание образа приложения можно найти в официальных документациях Docker [5]. Контейнер является инсталляцией Docker образа. В этой среде образ состоит из цепи слоев. При создании образа возможно указать родительский слой. Это позволяет наращивать на готовые образы дополнительную логику или приложения. В пример можно привести образ, содержащий JavaVM. В случае если ваше приложение использует JVM, можно родительским слоем указать образ с предустановленной JVM, перенести JAR или WAR и указать команду для запуска.

В процессе описания образа существует возможность объявить и проинициализировать переменные окружения, что позволяет гибко управлять вашим приложением. При запуске контейнера можно поменять параметры окружения. Приведу в пример объявление и использование переменной профиля Spring приложения внутри контейнера. Создавать переменную мы будем на этапе создания образа в Dockerfile, Листинг 1.

1. ENV profile manager
2. CMD ["java","-jar","./Diplom.jar","--spring.profiles.active=${profile}"]

Листинг 1 Создание и использование переменной в Dockerfile

В первой строке мы видим “ENV” - это оператор объявление переменной окружения, а “profile” и “manager” - название и значение по умолчанию соответственно. Вторая строка показывает, как описать команду запуска приложение в контейнере, а также в ней можно увидеть, как мы используем нашу переменную. Изменить значение переменной можно при запуске контейнера. Например, с помощью YML файла, пример которого мы рассмотрим позднее.

### **Docker Swarm**

Для создания кластера в Docker существует Docker Swarm, который легко позволяет развернуть необходимое количество узлов с заданными характеристиками и ресурсами. В этом режиме работы позволяется создать два типа узлов: Manager и Worker. При этом, на узлах роли менеджера также позволяется выполнять полезную нагрузку.

Но это не означает, что вы не можете выделять конкретные узлы для определенных задач. Для этого существуют метки. Любой узел, можно отметить соответствующей командой: docker node update --label-add variableName=value node-name

Этот механизм можно использовать при развертывании сервисов. В пример можно привести запуск базы данных. Естественно такой контейнер должен запускаться на узле с выделенным дисковым пространством. При развертывании сервиса, мы можем указать условия выбора узлов. В данном случае наличие определенной метки.

У Docker существует прослойка над сетью хостовой машины, что с легкостью позволяет реализовывать множество функций: зонирование, изолирование контейнеров, связку физических машин, связку определенных контейнеров и другие. Конечно, за эту легкость приходиться нести накладные расходы, что может быть критично для задач HPC и применение MPI, RDMA. На тему сравнение производительности различных инструментов контейнеризации существует исследовательская работа Корнельского университета [6] которая также указывает на эту проблему. Но в отличие от задач HPC, для Enterprise - разработки важным является масштабирование, развертывание, отказоустойчивость. Docker контейнеры позволяют организовывать это с относительной легкостью.

## Развертывание сервисов

Современным способом развертывание сервисов на кластере, основанном на Docker Swarm - использование YAML файла с описанием сетей, сервисов, параметров, зависимостей и политик развертывания. Такой файл позволяет одной командой развернуть всю необходимую архитектуру. Так как YAML файл содержит в себе много данных, то по нему можно узнать много информаций об архитектуре приложения.

## Интеграция сервисов

Микросервисная архитектура предполагает слабую связанность между компонентами системы и динамическое добавление, и удаление инсталляций компонент. Для организаций общения сервисов в такой среде необходимы специальные инструменты. В этой главе будут кратко описаны: Discovery service, Message broker, Key-value storage

## 

### Discovery Service

В микросервисной архитектуре необходимо: связывать микросервисы между собой, знать количество запущенных инсталляций микросервисов. Существует решение этой проблемы — реестр сервисов.

Реестр содержит в себе перечисление зарегистрированных сервисов из которого можно узнать адрес и количество инсталляций. Зачастую, такие реестры позволяют динамически регистрировать сервисы, проводить проверку работоспособности сервиса и имеют REST API для взаимодействия. Таким образом, с помощью реестра, есть возможность распространить информацию о сервисах другим сервисам и установить связь между ними.

В отличии DNS реестров, discovery service позволяют проводить более сложную балансировку, например, на основе известных глобальных данных, или параметров запроса пользователя [1, 222].

Если сервис меняет адрес, возможно столкновение с проблемой DNS кэширования. В случае discovery service, сервис повторит регистрацию и остальные сервисы смогут получить актуальную информацию.

### Key-value хранилище

Еще одна необходимость микросервисной архитектуры выделенное общее хранилище данных с высокой скоростью доступа. Такое хранилище представляет собой NOSQL решение на основе пар “ключ‑значение”. Где ключ - уникальный идентификатор, а значение может быть любым объектом. Такие базы данных имеют высокую пропускную способность, что важно для хранения общих данных. К таким данным можно отнести: глобальные параметры, пользовательские данные (сессия).

### Consul

Consul - программный децентрализованный регистр сервисов от компаний HashiCorp, который также решает задачу key-value хранилища.

Архитектура Consul — децентрализованная, поэтому существует две вариации запуска Consul:

1. Как сервер, т.е. эта инсталляция отвечает за регистрацию сервисов и распространение информаций между сервисами. Причем среди всех запущенных в кластере серверов Consul, автоматически выделяется лидер, а остальные серверы проводят репликацию с него. Consul является отказоустойчивым решением, поэтому в случае выхода из строя лидера, следующий лидер выбирается автоматически из оставшихся серверов.
2. Как локальный агент. Такая инсталляция подключается к удаленному серверу и это позволяет работать микросервисам с Consul как с локальным сервером. В этом решение мы избавляемся от необходимости сообщать микросервисам адрес сервера Consul.

Такая архитектура позволяет работать нескольким датацентрам и распределять нагрузку между ними

Для работы с Consul существует REST API. С его помощью возможно:

* Работать с локальной инсталляции.
* Проводить регистрацию сервисов.
* Искать сервисы.
* Работать с key-value хранилищем.
* Проводить healthcheck сервисов.

При регистрации сервисов в Consul мы передаем описание нашего сервиса: порт, адрес, id (для масштабирования сервиса), общее имя сервиса, состояние, путь для healthcheck, интервал healthcheck, TTL (Time to live).

### Message broker

Брокер сообщение - это компонента, реализующая очереди сообщений. Этот механизм является реализаций паттерна “publisher-subscribe”, позволяющий организовывать асинхронное общение между компонентами системы. При этом стороны не обязаны знать друг о друге. Исключением является структура сообщения. Все это позволяет уменьшать связанность сервисов.

Для этой задачи был разработан специальный протокол AMQP (Advanced Message Queuing Protocol). Он состоит из трех основных компонент:

1. Сообщение - единица информаций, передаваемая подписчику.
2. Точка обмена (exchange) - вершина в графе маршрутизаций сообщений, отвечает за дальнейшую передачу сообщения по выбранному каналу. Выделяют три типа:
   1. fanout - точка обмена передающая сообщение во все исходящие вершины
   2. direct - точка обмена передающая сообщение во все вершины, совпадающие с ключом маршрутизаций
   3. topic - точка обмена перед передающая сообщение во все вершины, с ключами маршрутизаций совпадающей с заданной маской ключа.
3. Очередь - конечные точки, которые содержат буфер для накопления сообщений. К ним подключаются подписчики и забирают сообщения.

За счет наличие очередей брокеры сообщений позволяют обеспечивать отказоустойчивость архитектуры. В ситуации повышенной нагрузки, очереди будут накапливать сообщения, и они будут находиться там пока подписчик сможет их забрать или, когда время ожидания пройдет заданную черту TTL. Аналогично с помощью очередей можно балансировать нагрузку между подключенными подписчиками.

### RabbitMQ

Один из самых популярных брокеров сообщений от компаний Pivotal, реализующий протокол AMQP. Это из самых производительных решений, имеющий множество интерфейсов для различных языков программирования. RabbitMQ позволяет создать отказоустойчивый кластер с репликацией данных на несколько узлов. Помимо AMQP существует поддержка HTTP, XMPP и STOMP. А еще есть множество дополнений, в том числе клиент для Consul.

## Работа с базами данных

В Java среде существуют много инструментов для работы с базами данных. На низком уровне это драйверы и коннекторы к определенным базам, которые позволяют с помощью SQL производить чтение, запись, управление транзакциями. Но это не вершина айсберга. В объектно-ориентированном мире Java существует идея ORM, спецификация JPA. Это позволяет абстрагироваться от конкретной БД.

## 

### ORM

В объектно-ориентированной среде при необходимости работы с реляционными базами данных в современной разработке использует Object-Relational Mapping для создания виртуального слоя между базой и средой. Этот слой отвечает за сохранение, чтение, отображение таблиц в объекты и установку связей. Одним из самых популярных решений для Java - Hibernate

### JPA

JPA - Java Persistence API является спецификацией получение доступа к реляционной базе данных. JPA сама по себе-это всего лишь спецификация, а не продукт. JPA-это просто набор интерфейсов, которая требует реализацию. JPA позволяет легко сохранять POJO (простые старые Java объекты), не требуя, чтобы классы реализовывали какие-либо интерфейсы или методы. JPA позволяет определять объектно-реляционные отображения объекта через стандартные аннотации или XML конфигурацию, определяющие, как класс Java сопоставляется с таблицей реляционной базы данных. JPA также определяет API Entity Manager среды выполнения для обработки запросов и транзакций объектов к базе данных. JPA определяет язык запросов объектного уровня JPQL, позволяющий выполнять запросы к объектам из базы данных.

В современной разработке для конфигураций маппинга POJO объекта используют аннотаций. Это позволяет описать все связи и свойства в самом POJO классе. Листинг 2 – это пример описания Entity для работы с таблицей Task.

1. @Entity
2. **public** **class** Task **extends** EntityParent {
3. @Column
4. **private** String fileName;
5. @Column
6. @Enumerated(EnumType.STRING)
7. **private** TaskType type;
8. @Column(name = "image")
9. **private** **byte**[] image;
10. @Transient
11. **public** **static** **final** **int** widthPrev = 240;
12. ….
13. @Column(name = "user\_id")
14. **private** Long userId;
15. @ManyToOne(fetch = FetchType.LAZY)
16. @JoinColumn(name = "user\_id", insertable = **false**, updatable = **false**)
17. **private** User user;
18. …. //getter + setter
19. }

Листинг 2 POJO класс для работы с таблицей в базе данных через Hibernate

1. @Entity - аннотация, указывающая что это объект для маппинга. С помощью параметров аннотаций можно указать таблицу БД, с которой нужно связать этот класс. В случае если параметр отсутствует, имя таблицы и класса совпадают.
2. @Column - аннотация указывающие что свойство хранится в колонке с заданным именем в параметре аннотаций. В случае если имя не указано, берется имя свойства.
3. @Transient - аннотация указывает что свойство не нужно хранить в БД. Её часто использует для статических значений и полей, которые вычисляются программно
4. @ManyToOne - одна из аннотаций для установки связей между POJO объектами. Еще существуют @OneToOne, @OneToMany, @ManyToMany. В зависимости от выбранного типа связи используют отдельные колонки или даже отдельные таблицы.

В случае если у наших сущностей есть общие поля их можно вынести в общий класс, пометить с помощью @MappedSuperclass и наследоваться от этого класса, как это было сделано в примере.

Для проведения аудита существует аннотация @EntityListeners, с помощью которой мы можем указать классы, обрабатывающие такие события как создание, обновление, удаление. В пример можно привести процесс установки времени обновления сущности, Листинг 3

1. **public** **class** EntityListener {
2. @PreUpdate
3. **public** **void** postUpdate(EntityParent o) {
4. o.setUpdateOn(DateTime.now());
5. }
6. }

Листинг 3 Добавление перехватчика на события обновление сущности Hibernate

В этом примере, с помощью аннотаций @PreUpdate происходит перехват объекта в процессе обновления.

### Hibernate

Hibernate - это популярная реализация спецификации JPA 2.1. Помимо этого Hibernate позволяет создать структуру базу данных на основе описанных классов помеченных аннотаций @Entity. Hibernate отвечает за генерацию SQL и работает с большинством современных баз данных без какой-либо привязки. А это означает что в любой момент можно сменить базу данных.

В Hibernate разработан аналог SQL - HQL, который позволяет использовать его на всех поддерживаемых баз данных. Этот язык позволяет работать с объектами и его свойствами, как с таблицами и колонками в SQL.

Для создание динамических запрос к базе данных с помощью Hibernate, в объективно-ориентированной среде есть абстракций Criteria Queries. Она позволяет генерировать HQL Select, Join, Update, Delete запросы по заданным условиям.

## Spring Framework

Spring - это набор компонент для современной разработки. Из-за IoC архитектуры, популярности, наличие стартеров у Spring Boot Framework существует множество интеграций с другими известными компонентами от других производителей. Выделим некоторые компоненты, необходимые для разработки микросервисов:

1. Spring Framework реализована поддержка AspectJ и существует собственная реализация Аспектно-ориентированного программирования(АОП). Этот инструмент позволяет добавлять общую функциональность избегая ООП парадигму. Например, для типичного кода проведение транзакций и в случае ошибок отката изменений.
2. Spring позволяет работать с большинством современных баз данных для которых существует JDBC (Java DataBase Connectivity). Для ORM используется Hibernate либо MyBatis. В отличие от Hibernate, MyBatis не реализует JPA.
3. Поверх Hibernate Spring предлагает свой дополнительные инструменты в компоненте Spring Data позволяя с легкостью создавать запросы к БД.
4. В Spring реализован фреймворк Spring Integration для отправки сообщений, работы с событийно-ориентированной архитектурой, AMQP, JMS …
5. Компонента Spring Cloud для взаимодействия между сервисами, балансировка, масштабирование. Самые популярные инструменты в этой компоненте:
   1. DiscoveryClient - абстракция для работы с реестрами сервисов.
   2. Netflix Feign Client - инструмент для работы с Api других сервисов
   3. Netflix Ribbon - инструмент для обеспечения балансировки запросов
   4. Spring Retry - предоставляет возможность автоматического повторного вызова неудачной операции обращения к другому сервису. Это полезно, когда ошибки могут возникнуть из-за обрыва соединения.

### Bean

Spring Framework - это Inversion of Control контейнер, который управляет POJO объектами. В Spring эти объекты называют бинами. Контейнер отвечает за инициализацию, управление, связывание и уничтожение бинов. Контейнер создает бины в указанной области видимости - Scope. Также существует возможность создавать пользовательские области видимости.

Bean - это объект, экземпляр которого создается, собирается и иным образом управляется контейнером Spring IoC. Создавать бины и производить в них инъекций можно с помощью:

* XML - конфигурации
* Java - конфигурации
* Аннотации

В этой исследовательской работе основной упор будет на последнем типе конфигураций. Ибо он современный и наглядный.

## 

### Описание Bean

Рассмотрим на примере как можно создать bean на основе аннотаций. Тут можно выделить два пути:

1. На основе аннотаций @Component
2. На основе аннотаций @Bean

Рассмотрим более внимательно эти подходы. В случае, когда мы описываем контроллеры, сервисы, репозиторий нам удобно использовать наследники аннотации @Component такие, как @Controller, @Service, @Repository соответственно. Это удобно, когда нет необходимости в дополнительных настройках помимо инициализаций и встраивание инъекций.

Рассмотрим на примере (Листинг 4) сервиса для взаимодействия с пользователями системы.

1. @Service
2. @Scope(value = ConfigurableBeanFactory.SCOPE\_SINGLETON)
3. @Transactional
4. **public** **class** UserService {
5. @Autowired
6. **private** UserIdentRepository userIdentRepository;
7. **public** UserIdentRepository getRepository() {
8. **return** **this**.userIdentRepository;
9. }
10. @Transactional(propagation = Propagation.REQUIRES\_NEW)
11. **public** UserIdent save(UserIdent userIdent) {
12. **return** userIdentRepository.saveAndFlush(userIdent);
13. }
14. **public** UserIdent getCurUser() {
15. Authentication authentication = SecurityContextHolder
16. .getContext()
17. .getAuthentication();
18. Object principal = authentication.getPrincipal();
19. **if** (principal **instanceof** UserIdent) {
20. **return** (UserIdent) principal;
21. }
22. **return** **null**;
23. }
24. @PostConstruct
25. **private** **void** postInit(){
26. ….
27. }
28. }

Листинг 4 Описание бина сервиса пользователей системы

1. @Service указывает что необходимо создать bean данного сервиса
2. @Scope указывает область видимости создаваемого компонента
3. @Autowired указывает поле для инъекций, аналогична аннотаций @Inject из Java EE [8, 75]
4. @PostConstruct указывает методы для выполнения после инициализаций bean

### Spring Discovery Client

Discovery Client - это абстракция, позволяющая работать реестрами сервисов. Для каждого конкретного реестра в Spring Cloud включает его реализацию. Это абстракция предназначена только для чтения. Некоторым реестрам необходимо чтобы клиент проходил регистрацию. В этом случае реализация содержит механизм регистраций и требует лишь инициализаций неких параметров. В случае использования реестра Consul необходимы следующие параметры:

* spring.cloud.consul.host
* spring.cloud.consul.port
* spring.cloud.consul.config.enabled
* spring.cloud.consul.config.prefix
* spring.cloud.consul.config.defaultContext
* spring.cloud.consul.discovery.register

Абстракция DiscoveryClient позволяет:

1. Получить список ключей инсталляций, зарегистрированных в реестре.
2. Получить описание инсталляций (хост, порт, URL, …).

### Spring Data JPA

Главная задача этого инструмента уменьшение количества строк кода для типичных задач реализаций доступа к данным различных хранилищ. В этом инструменте существует базовые интерфейсы доступа, такие как CrudRepository, PagingAndSortingRepository, Repository. Для создание собственных методов доступа их нужно лишь задекларировать в интерфейсе, наследуемом от одного из базовых и пометить интерфейс аннотаций @Repository. Название прототипа метода должно подчиняться правилам Spring Data JPA компонент и состоять из специальных ключевых слов. Для понимания приведу в пример репозиторий для доступа к данным пользователей (Листинг 5)

1. @Repository
2. **public** **interface** UserRepository **extends** JpaRepository<User , Long> {
3. Optional<User> findFirstBySub(String sub);
4. List<User> findAllByFirstname(String firstname);
5. List<User> findAllByCreateOnAfterOrderByNameDesc(DateTime createOn);
6. List<User> findAllByAgeGreaterThanAndDiedFalse(**long** age);
7. }

Листинг 5 Описание запросов к Базе Данных с помощью Spring Data JPA

Spring Data JPA позволяет не заниматься реализацией этих методов, они генерируется в процессе инициализаций бинов.

### Spring Profile

Скромный инструмент, позволяющий управлять конфигураций приложения на основе профилей. Состоит из аннотаций @Profile - параметром которой является имя профиля.Аннотаций помечать можно всевозможные бины системы, в том числе репозиторий, сервисы, контроллеры, конфигураций.

При запуске приложение указываются профили, с которым оно запускается. Можно указать несколько профилей. Это можно сделать через конфигурационные файлы или через параметры командной строки.

Если бин помечен аннотацией @Profile и имя одного из текущих профилей соответствует, тогда бин создается. Это позволяет реализовать паттерн Абстрактная фабрика на уровне конфигураций системы.

В связке с Spring Boot вы можете включить файлы конфигураций, отключенные аннотацией @SpringBootApplication. Для этого есть аннотация @Import. Благодаря этим механизмам можно гибко управлять конфигурацией приложения. Это полезно, когда вы зависите от операционной системы, от источника данных, других нюансов или, когда необходимо изменить поведение приложения. Часто бывает необходим отладочный вывод, измерение производительности в процессе разработке. В финальном продукте такого допускать нельзя. В таких ситуациях удобно использовать профили для включения и отключение функциональности.

### Netflix Feign Client

Инструмент от компаний Netflix, позволяющей аналогично декларированию контроллеров описывать запросы к api другого сервиса без необходимости реализаций этих запросов. Этот инструмент работает в паре с Discovery Client и балансировщиком нагрузки (в случае стандартных настроек это Netflix Ribbon). Листинг 6 – декларирование запроса к API сервиса «manager»

1. @FeignClient(serviceId = "manager")
2. **public** **interface** ForeignWorkerClient {
3. @RequestMapping(method = RequestMethod.GET, value = "task/{id}")
4. Task getById(@PathVariable("id") **long** id);
5. }

Листинг 6 Описание запросов к API сервиса с помощью FeignClient

1. @FeignClient - указывает что этот интерфейс должен реализовать Feign Client
2. @RequestMapping - указывает параметры запроса. Аналогично ее используют при декларировании контроллеров системы
3. Прототип метода указывает параметры запроса и ожидаемый результат. Это решение требует описать структуру ответа как POJO класс в случае если объект состоит из нескольких примитивных свойств.

Механизм инъекций зависимостей в Spring Framework

Для работы аннотаций @Autowired и @Inject используется методы и классы из пакета Reflection API. Рефлексия позволяет получить описание класса, полей, методов, аннотаций, интерфейсов. С её помощью можно вызывать методы и изменять значения полей объекта. Помимо стандартных функций, возможно работать с приватными полями и методами. Это позволяет Spring находить аннотированные поля и встраивать в них объекты, даже если это приватное поле объекта.

В процессе инициализаций приложения, созданного на Spring Framework, создается описание необходимых бинов – Bean Definition, которое содержит метаинформацию создаваемого бина. Далее это описание проходит через набор реализаций интерфейса BeanFactoryPostProcessor. Этот интерфейс позволяет изменить параметры Bean Definition. BeanFactoryPostProcessor описывается как обычный бин. Далее по обработанному Bean Definition создается объект бина с помощью реализаций BeanFactory. Если есть необходимость изменить процесс создания для бина определённого типа или интерфейса, необходимо реализовать generic интерфейс FactoryBean и в нем определить необходимый процесс создания бина.

Созданный объект проходит через набор реализаций интерфейса BeanPostProcessor, в которые передается созданный объект. Этот интерфейс позволяет взаимодействовать с объектом до и после его инициализаций. Часто его используют для создания прокси объектов [7, 138].

### Spring Boot Framework

При разработке приложения на основе Spring MVC приходиться заботиться о типичных настройках, что является рутинной работой. Spring богат функциональными возможностями. И он отлично интегрируется с популярными инструментами (AMQP, ORM, Discovery Service, …). Это приводит к большому количеству необходимых, типичных настроек.

В дочернем проекте Spring Boot существует механизм автоконфигураций. Который основан на Стартерах. Стартер - это Jar файлы, которые описывает механизм конфигураций, определенных бинов. Каждый такой архив содержит в себе каталог META-INF со специальным файлом spring.factories. Этот файл содержит в себе перечисление классов конфигураций. Об этом можно больше прочитать в документации [2]

При запуске Spring Boot приложения идет анализ зависимостей, в них идет поиск стартеров и уже на их основе происходит конфигурация соответствующих инструментов. В пример можно привести популярные стартеры “Spring Boot Data JPA Starter”, “ Spring Boot Tomcat Starter” и другие. Несмотря на указанные зависимости, вы можете частично отключить файлы конфигураций. Для этого в аннотации @SpringBootApplication есть параметр exclude в который можно передать массив с перечислением классов конфигураций, которые вам не нужны.

# 2. Архитектура прототипа микросервиса

Для исследовательской работы была выбрано задача создание микросервиса для обработки пользовательских изображений. Эта задача имеет потенциал с использованием Deep Learning для стилизаций изображений или классификаций объектов. Такие задачи требуют асинхронной обработки с возможностью горизонтального масштабирования. Для этого применяется архитектура схожая с моделью распределенных вычислений MapReduce. В архитектуре прототипа выделено две инсталляций микросервиса: manager и worker. В зону ответственности Manager инсталляций входит:

1. Прием изображений от пользователей.
2. Отображение обработанных изображений.
3. Сохранение задачи в базу данных.
4. Отправку задачи в очередь RabbitMQ на обработку.
5. Получение и сохранений обработанного изображения.
6. Идентификацию и Авторизацию пользователя.

Инсталляции worker ответственны за:

1. Прием задания из очереди RabbitMQ
2. Получение исходного изображения по REST API от менеджера
3. Обработка задачи
4. Сохранений обработанного изображения с помощью REST API менеджера
5. Подтверждение обработки сообщения RabbitMQ

Для этой архитектуры необходимы все описанные инструменты: Consul, RabbitMQ, Hibernate, Spring.

## Архитектура инфраструктуры

Кластер – это группа ЭВМ, объединенных коммуникационной средой и работающих как единое целое. Элемент группы кластера называется узлом. Узел имеет собственные настройки и управляется своей операционной системы. Однородным кластером называется кластер, состоящий из узлов, имеющих одинаковую конфигурацию.

В кластерах распределения нагрузки выделяют главные и рабочие узлы (manager и worker соответственно). Если кластер нацелен на обслуживание пользователей, тогда главные узлы отвечают за маршрутизацию задач в кластере, общее хранилище, организацию общения между сервисами. На главных узлах не выполняют обработку задач, они перераспределяют задачи на рабочие узлы. Такие кластеры нацелены на повышение производительности сервиса. На Рис1 отображена архитектура инфраструктуры разрабатываемого прототипа.

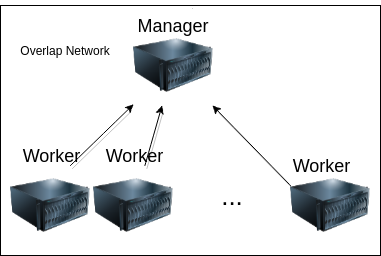


Рис. 1 Схема архитектуры инфраструктуры

В этой схеме все узлы связаны одной сетью. В этом случае мы гарантируем возможность связи всех узлов.

## Архитектура компонент прототипа

Архитектура прототипа состоит из независимых компонент, которые общаются между собой с помощью Discovery Service и Message Broker. На Рис.2 отображена архитектура прототипа, его зависимости и очередность запуска компонент в порядке очереди. Первым запускается Consul и его агенты. Остальные сервисы запускаются после и регистрируют себя в уже запущенном реестре с помощью локальных агентов Consul.

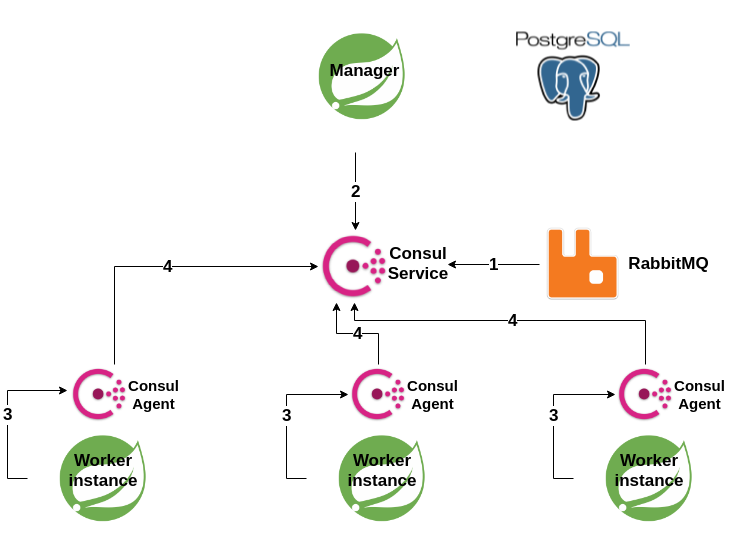


Рис. 2 Схема очередности запуска компонент разработанного прототипа

1. RabbitMQ регистрирует себя в реестре Consul.
2. Микросервис с профилем manager регистрирует себя в реестре Consul.
3. Микросервисы с профилем worker отправляют запрос на регистрацию в реестре к локальным агентам Consul.
4. Локальные агенты Consul перенаправляют запрос к серверу Consul.

В архитектуре прототипа PostgreSQL не регистрируется в реестре сервисов, из-за того, что не является общим ресурсом. При завершении работы микросервиса, происходит удаление данных о сервисе из реестра Consul с помощью REST API. После инициализаций в реестре сервисов список должен быть таким как на Рис.3.

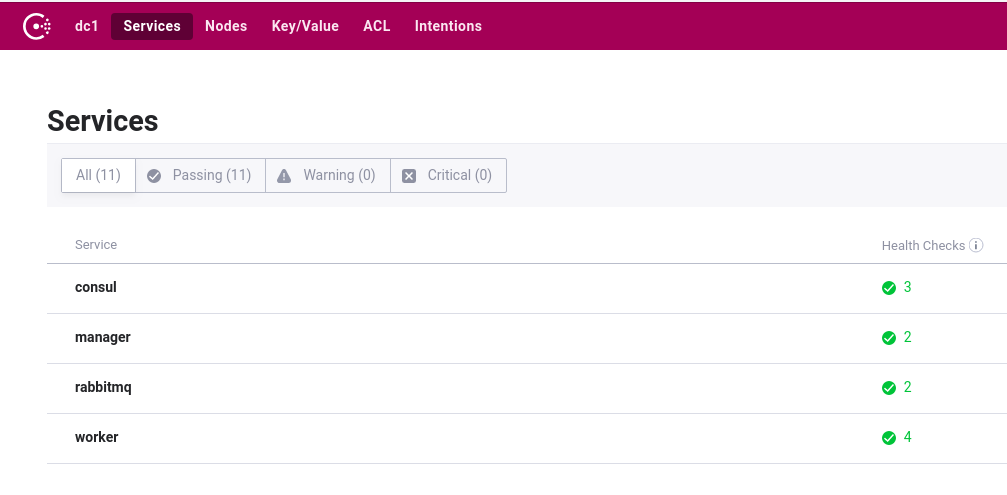


Рис. 3 список зарегистрированных сервисов в реестре Consul.

Здесь перечислены основные компоненты архитектуры, которые проходят регистрацию в реестре сервисов.

## Архитектура обработки запроса

Процесс обработки изображение в такой архитектуре сложнее, чем обычная синхронная обработка пользовательского запроса. Процесс представлен на Рис.4.

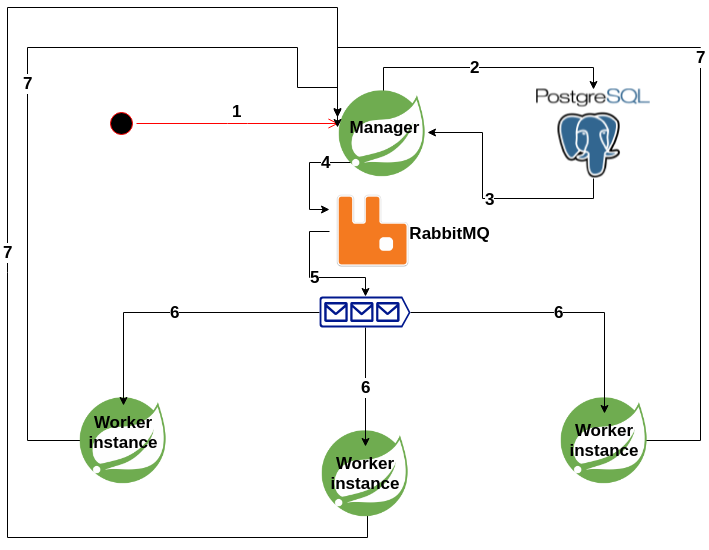


Рис. 4 Процесс маршрутизаций задачи между компонентами системы

1. Пользователь с помощью интерфейса создает задачу (Загружает изображение, устанавливает параметры обработки).
2. Задание формирует сущность содержащие описание задачи, изображение. Сущность сохраняется в базе данных.
3. Возвращается уникальный идентификатор сохраненной сущности.
4. Задача с помощью адаптера к Базе данных передается на exchange в RabbitMQ.
5. Сущность поступает в очередь, к которой подключен микросервис с профилем worker.
6. Свободная инсталляция с профилем worker забирает сообщение из очереди и обрабатывает в зависимости от параметров.
7. Обработанная задача с помощью REST API микросервиса с профилем manager сохраняется в базе данных.

# 3. Практическая реализация

## Создание инфраструктуры на Google Cloud Platform

Google предоставляет инструмент Google Cloud SDK, который позволяет работать Docker-Machine с облачными узлами как с локальными. Docker-Machine позволяет создавать и удалять виртуальные узлы, создавать кластеры и добавлять к нему виртуальные узлы, создавать сети среди узлов. Этот компонент и занимается созданием архитектур в Docker Swarm.

Через интерфейс Google Cloud Platform можно настраивать DNS, настраивать сети, управлять созданными узлами.

При развертывании разработанного прототипа сервиса используется лишь один выделенный узел с ролью менеджера, что недостаточно в промышленной эксплуатации. Такое решение было принято из-за ограниченности доступных ресурсов. Начиная от трех узлов с ролью менеджера можно надеяться на полноценное масштабирование и отказоустойчивость. Среди узлов типа “Worker” выделяется один для работы с базой данных. На данный узел предоставляется больше дискового пространства, в зависимости от размеров планируемых данных для хранения.

## 

## Разработка микросервиса с помощью Spring

Для сокращения объема кодовой базы, при разработке manager и worker инсталляций, используется механизм Spring Profile. Это позволяет разрабатывать вместо двух приложений - одно. Это может показаться странным, но у этих приложений очень много общих компонент, моделей, настроек. С помощью механизма профилей мы выделяем общие компоненты и уникальные. Если рассматривать инсталляцию manager, то для нее уникальными являются контроллеры, пользовательский интерфейс и компоненты, работающие с базой данных. Общими в этом случае являются компоненты, работающие с Discovery Service, RabbitMQ и модели. В текущем прототипе есть два профиля соответственно. При запуске Spring приложения профиль передаются через командную строку.

Благодаря Discovery Client, микросервис узнает путь до RabbitMQ. В случае профиля worker, микросервис узнает адрес инсталляций с профилем manager. Для обращения к API manager инсталляций, микросервисы с профилем worker используют Netflix Feign Client.

С помощью интеграций для RabbitMQ, в процессе запуска приложения создается необходимая архитектура и маршрутизация в брокере сообщений. В рамках этого прототипа она очень проста и состоит из двух очередей:

1. Очередь от manager с новыми заданиями. К этой очереди подключается все инсталляций worker и забирают задания из нее
2. Очередь от worker к manager с ошибками. В случае если в процессе обработки задания возникает ошибка, worker информирует об этом manager через эту очередь

Для пользовательского интерфейса используется шаблонизатор Thymeleaf. С его помощью был создан простой интерфейс позволяющий:

* Отображать уменьшенные версий обработанных изображений.
* Удалять результаты обработанных заданий.
* Создавать новые задания.

Thymeleaf позволяет встраивать динамические данные в статические ресурсы View. Thymeleaf работает с Java объектами и позволяет вызывать методы класса объекта. Этот инструмент может производить операций сравнения и перебирать коллекций. Это пригодилось для отображения уменьшенных изображений всех выполненных задач пользователя, показано на Рис.5.

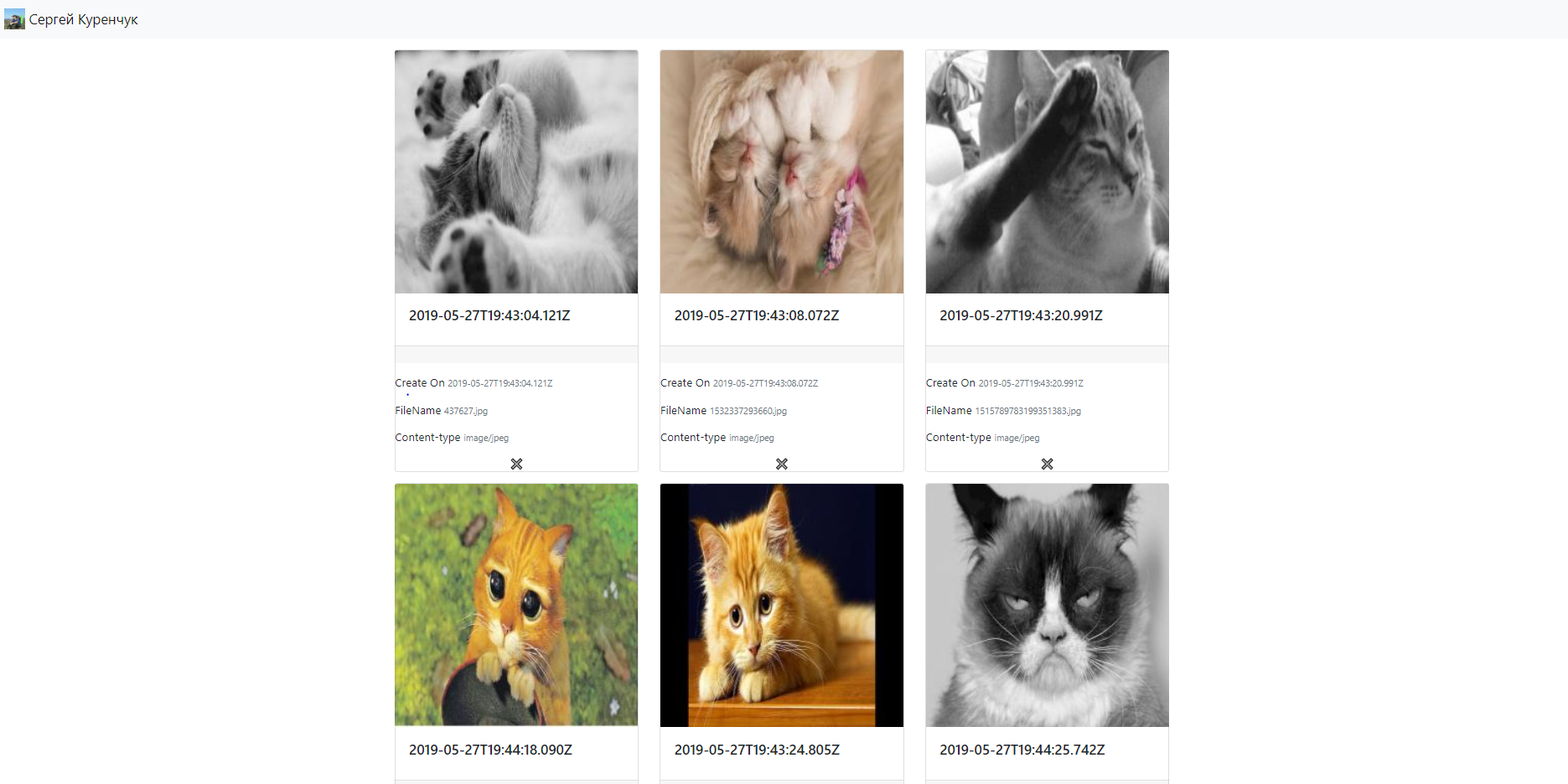


Рис. 5 Страница обработанных заявок

Было необходимо перебрать все заявки и отобразить только обработанные. В случае если произошла ошибка при обработке, было необходимо отобразить статическое превью c сообщением об ошибке. Для этого и использовались циклы, условия в Thymeleaf.

## Docker образ микросервиса

Для docker swarm инфраструктуры был разработан образ прототипа. Выбранный профиль и другие настройки микросервиса передаются с помощью переменных окружения. Для хранения, сборки и версионности используется Docker Hub. Такое решение позволяет частично автоматизировать процесс обновления. Docker Hub позволяет подключить ваш репозиторий к GIT репозиторию и пересобрать образ при заданных условиях. Например, условием может быть внесение изменений в заданную ветку.

В процессе разработке образа приложения пришлось столкнуться с некоторыми нюансами работы в Docker Swarm:

1. Очередность запуска сервисов зависит от прописанных зависимостей в блоках “depends\_on”. Если при заданных зависимостях существует не единственная комбинация, выбирается случайная из возможных. Это сделано для борьбы с ошибками зависимостей.
2. При описании сервиса, с помощью блока “environment” вы можете изменить значения переменных окружения контейнеров.
3. На текущий момент нет возможности задать адрес проверки состояния сервиса(healthcheck). Это приводит к тому, что в процессе запуска приложения, Docker запускает следующий сервис из очереди, хотя текущий сервис не успел полностью проинициализироваться. По этой причине возможна ситуация, когда зависимости сервиса могут находится не в рабочем состояний, несмотря на очередность их запуска. Достойным решением этой проблемы является остановка контейнера. В этом случае Docker будет перезапускать ваш контейнер пока зависимости не придут в рабочие состояние
4. Указание сети в блоке “networks” при описании сервиса, не добавляет сеть в зависимости сервиса. При развертывании приложение возможна попытка создание контейнера до создания сети, что приводит к краху.
5. Приятном моментом является то, что имена сервисов являются их хостами в выбранных сетях, в случае если вы не указали параметр Host у вашего сервиса
6. В случае, когда вам надо равномерно распределить сервис по узлам типа “Worker”, в сети предлагают использовать такое условие: constraints: [node.role == worker]. Что не совсем корректно. Оно не означает равномерного распределения и некоторые узлы могут остаться без нагрузки. Для решения этой проблему нужно дополнительно использовать режим “global”.

## Оценка работы прототипа

Один из способов оценивание производительности разработанного прототипа - это проведение нагрузочного тестирования. В рамках этой работы достаточно провести тестирование одного контроллера, отвечающего за загрузку изображения. А далее рассматривать с какой скоростью будут обрабатываться сообщения из очереди в RabbitMQ.

Для анализа очередей есть графический интерфейс у RabbitMQ. Из него и будут взяты данные для демонстраций. В рамках тестирования, отображена зависимость между количеством инсталляций и скорости обработки пакета изображений.

Для проведение нагрузочного тестирования используется Postman. Это ПО позволяет отправлять запросы с необходимыми данными и заданным интервалам. Для того чтобы избежать накладных расходов при использовании сети, тестирование проведено в локальной сети.

## 

## Результаты тестирования

В таблице представлены усредненные данные тестирования на физической машине с ограниченными ресурсами. Поэтому количество инсталляций ограничено. Каждый тест выполнялся в 100 итераций. На всех итерациях загружалось одно и тоже цветное изображение и применялся фильтр перевода в черно-белый формат.

Полученные данные несопоставимы с производительностью облачных платформ. Они приведены лишь для отображения работы механизма масштабирования. Рис.6 отображает график зависимости между количеством инсталляций и пропускной способностью приложения

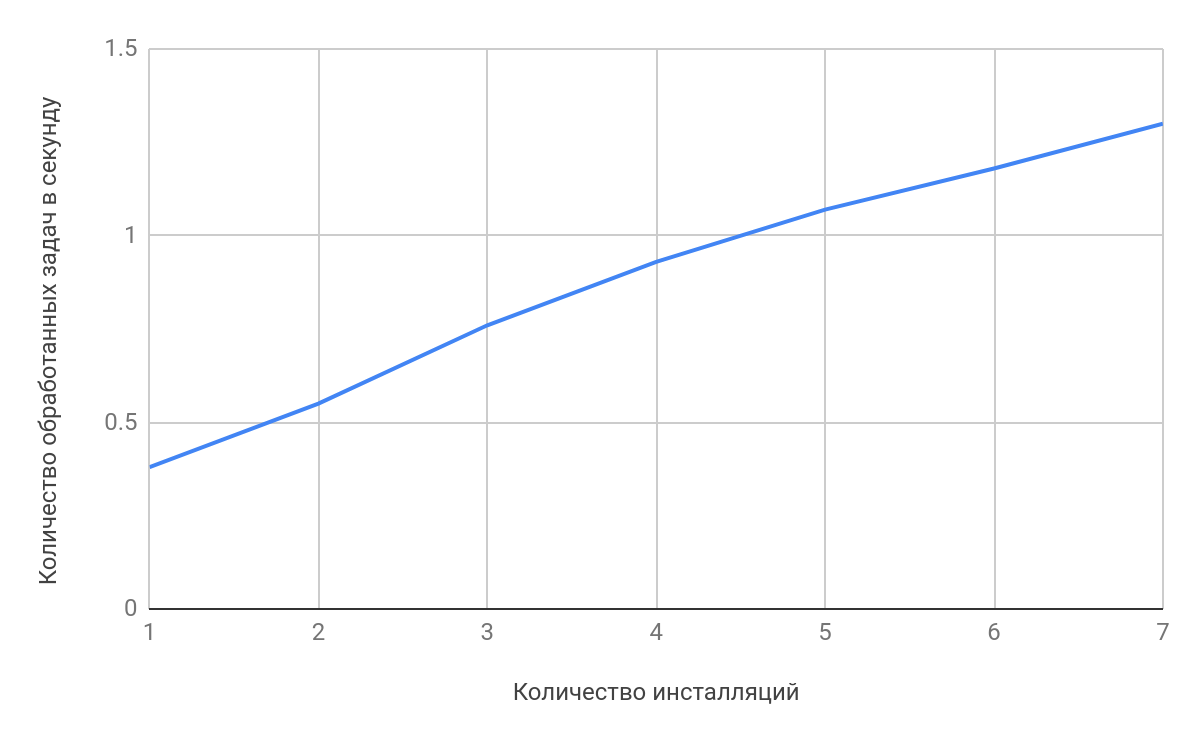


Рис. 2 Зависимость между количеством узлов и пропускной способностью сервиса

По этим данным можно сделать единственный вывод: с помощью масштабирования возможно увеличить пропускную способность приложения.

В разработанном решение ограничением пропускной способности является время, затраченное на:

1. Доставку сообщение из брокера сообщений
2. Получение изображение от manager инсталляций с помощью REST API
3. Обработку изображение
4. Сохранение изображения на инсталляции manager с помощью REST API
5. Подтверждение обработки сообщения для RabbitMQ

# Заключение

# При выполнение исследовательской работы была изучена Микросервисная архитектура, проанализированы и выбраны различные инструменты коммуникаций сервисов, и разработан прототип микросервиса для обработки изображений. Были выделены основные компоненты Spring Framework для разработки микросервисов и изучены компоненты интеграций сервисов Consul и RabbitMQ.

# Были достигнуты все поставленные цели исследовательской работы и разработанный прототип успешно запущен в облачном сервисе от Google Cloud Platform. Он имеет потенциал для обработки большого объема данных.

Разработанная архитектура может быть использована и для других независимых задач. Найти ей применение можно в обработке аудио, фото и текстовых данных. А слабая связь в этой архитектуре позволяет встраивать это решение в другие системы без огромных затрат на интеграцию.

# Список литературы

1. Java в облаке. Spring Boot, Spring Cloud, Cloud Foundry. - СПБ.: Питер, 2019. - 624 с.
2. Spring Boot Documentation // Spring Boot Reference Guide URL: https://docs.spring.io/spring-boot/docs/current (дата обращения: 28.05.2019).
3. Documentation // RabbitMQ - open source message broker URL: https://www.rabbitmq.com/documentation.html (дата обращения: 28.05.2019).

# Gavin M. Roy, James Titcumb RabbitMQ in Depth. - 1-е изд. September 2017 - 264 c.

1. Dockerfile reference // Docker Documentation URL: https://docs.docker.com/engine/reference/builder/ (дата обращения: 28.05.2019).
2. Performance Evaluation of Container-based Virtualization for High Performance Computing Environments // arXiv.org - a highly-automated electronic archiv URL: https://arxiv.org/abs/1709.10140 (дата обращения: 28.05.2019).
3. Spring. Все паттерны проектирования. – СПБ.: Питер, 2019-320 с.: ил.
4. Изучаем Java EE. Современное программирование для больших предприятий. - – СПБ.: Питер, 2018-384 с.: ил.