МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение Высшего профессионального образования  
 «Пермский государственный национальный исследовательский университет»

ОТЧЕТ

по лабораторной работе №4 «Разработка распределенного приложения для локальной сети»

|  |  |
| --- | --- |
| Работу выполнили  студенты группы ФИТ-7-8-2017  Сибиряков С.А. \_\_\_\_\_\_\_\_\_\_  (подпись)  Сюткин В.Д. \_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись)  Ошев К.В.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись)  «\_\_» \_\_\_\_\_\_\_\_\_\_\_\_2021 г. | Проверил  кандидат ф.-м. наук,  доцент кафедры МОВС,  доцент кафедры ПМИ  механико-математического  факультета  Деменев А.Г. \_\_\_\_\_\_\_\_\_\_\_  (подпись)  «\_\_» \_\_\_\_\_\_\_\_\_ 2021 г. |

Пермь 2021

Оглавление

[1. Постановка задачи 3](#_Toc68274544)

[2. Предметная область и технологии разработки 5](#_Toc68274545)

[3. Сценарии использования 7](#_Toc68274546)

[3. Процесс развёртывания 11](#_Toc68274547)

[3.1. Первичный запуск распределённого приложения 11](#_Toc68274548)

[3.2. Повторный запуск 20](#_Toc68274549)

[4. Соответствие критериям оценивания 21](#_Toc68274550)

[4.1. Проектирование с учётом особенностей предметной области 21](#_Toc68274551)

[4.2. Оптимальность архитектуры 21](#_Toc68274552)

[4.3. Параллельная работа нескольких клиентов и серверов 21](#_Toc68274553)

[4.4. Масштабируемость приложения 26](#_Toc68274554)

[4.5. Динамическое реконфигурирование 27](#_Toc68274555)

[4.6. Средства коммуникации 27](#_Toc68274556)

[4.7. Отказоустойчивость системы 27](#_Toc68274557)

[4.8. Восстановление компонента на другом узле 30](#_Toc68274558)

[4.9. Описание архитектуры каждого компонента 30](#_Toc68274559)

[4.9.1. Клиент 30](#_Toc68274560)

[4.9.2. Сервис обнаружения лиц (faces-сервис) 31](#_Toc68274561)

[4.9.3. Сервис вырезки лиц (crop-сервис) 31](#_Toc68274562)

[4.10. Структура передаваемых данных 31](#_Toc68274563)

[3.1. Описание способа передачи сообщений 32](#_Toc68274564)

[Заключение 33](#_Toc68274565)

[Библиографический список 34](#_Toc68274566)

# 1. Постановка задачи

***Цель***: изучение методов коммуникации процессов в сети, а также средств динамического конфигурирования распределенных приложений.

***Проверяемые компетенции***: способность работы с информацией из различных источников, включая сетевые ресурсы сети Интернет, для решения профессиональных задач; способность применять на практике теоретические основы и общие принципы разработки распределенных систем; уверенное знание теоретических и практических основ построения распределенных баз данных; способность использовать на практике стандарты сетевого взаимодействия компонент распределенной системы.

Задание выполняется в группе (не более трех студентов) или индивидуально. Каждый студент отчитывается по каждому пункту задания индивидуально.

Требования к выполнению работы:

* + Приложение должно обеспечивать параллельную работу нескольких клиентов и серверов. Дополнительное требование: возможность запуска нескольких серверов на одном компьютере.
  + Клиентские приложения должны автоматически находить серверы для обслуживания и выполнения заданных функций.
  + Серверы системы могут выполнять различные функции.
  + При разрыве сеанса приложения должны автоматически восстанавливать свою работоспособность.
  + Для хранения данных и доступа к ним применить ADO и/или ADO.NET (или их аналоги).
  + Приложения должны поддерживать возможность взаимодействия в различных режимах.
  + Для организации взаимодействия нужно использовать различные средства коммуникации (именованные каналы, мейлслоты, сокеты, MSMQ, .Net Remoting, web-сервисы, WCF-сервисы), сравнив их возможности.

В программе **необходимо** использовать открытые данные. Это могут быть государственные открытые данные любой страны (например, с порталов и <http://data.gov.ru/>,), иные общедоступные машиночитаемые данные.

По окончании выполнения задания каждая группа студентов должна подготовить отчет и доклад на 7-10 минут. Выступление студентов сопровождается показом презентации, отражающей основные позиции разработки.

Отчет по выполнению задания должен включать:

1. Общее описание приложения. Постановка задачи, введение в предметную область.
2. Архитектура системы. Обоснование выбора данного типа архитектуры распределенного приложения. Алгоритм работы приложения в целом.
3. Архитектура каждого из логических компонент системы (серверы, клиенты, диспетчеры). Подходы к реализации. Алгоритмы работы. Многопоточность, обоснование.
4. Методы коммуникаций компонентов системы (клиент→сервер, сервер→клиент и т.д.). Обоснование выбора этих методов коммуникации.
5. Способ передачи данных (синхронная / асинхронная, однонаправленная / двунаправленная и т.д.). Обоснование.
6. Структура передаваемых данных. Вид протоколов, обоснование выбора.
7. Отказоустойчивость системы. Как система поведет себя, если «исчезнет» один или несколько ее компонент. Что произойдет с системой, если «исчезнувший» компонент будет восстановлен на другом узле сети.
8. Работа с базой данных (если используется). Обоснование.
9. Исходный код приложений с комментариями.

Максимальное количество баллов, которые студент может получить за выполнение работы равно тридцати. Распределение баллов за выполнение работы представлено в следующей таблице:

|  |  |
| --- | --- |
| **Критерий оценивания** | **Оценка** |
| Распределенное приложение спроектировано с учетом особенностей предметной области. Выбрана наиболее подходящая модель распределенной системы. Если используется распределенная база данных, то тиражирование данных имеет подходящий для данной предметной области механизм и архитектуру. | 3 |
| Архитектура системы является оптимальной для заданных при разработке критериев. В отчете присутствует обоснование выбора данного типа архитектуры. | 3 |
| Приложение обеспечивает параллельную работу нескольких клиентов и серверов, в том числе на одном компьютере. Серверы распределенной системы выполняют различные функции. | 2 |
| Приложение является масштабируемым, позволяет добавлять новых участников взаимодействия без переписывания кода и перезапуска приложений. | 2 |
| Существует возможность динамического реконфигурирования системы. | 4 |
| Для организации взаимодействия компонент распределенной системы используется не менее четырех различных средств коммуникации. В отчете присутствует четкое обоснование выбора средств взаимодействия для каждого конкретного случая. | 4 |
| Система является отказоустойчивой. В случае если один и/или несколько компонент системы аварийно завершают свою работу. | 2 |
| Распределенное приложение продолжает работать и в случае, если после аварийного завершения некоторого компонента, он восстановлен на другом узле вычислительной сети. | 3 |
| Отчет содержит подробное описание архитектуры каждого компонента распределенного приложения. | 3 |
| В отчете описана структура передаваемых данных, формат сообщений и вид протокола, используемого для этого. | 2 |
| В отчете представлено описание способа передачи сообщений при коммуникации компонентов распределенной системы с обоснованием. | 2 |

# 2. Предметная область и технологии разработки

Выбранная для реализации предметная область – Computer Vision, то есть компьютерная обработка изображений. Была выбрана область компьютерного зрения, занимающаяся поиском человеческих лиц на изображении. Задача приложения с точки зрения пользователя – определить количество человеческих лиц на выбранном пользователем изображении, а также определённым способом показать пользователю эти лица. Выбранный способ демонстрации лиц – создание отдельных изображений с лицами, вырезанными из исходного изображения, которые будут храниться в определённой папке. Более подробно архитектура приложения будет описана далее в поздних разделах. Фотографии, используемые для тестирования приложения, взяты из Интернета в открытом доступе.

Выбранные технологии с точки зрения разработки исходного кода:

1. Язык программирования C#, использованный для реализации клиентской части приложения.

Среда разработки: Rider, приобретённая по бесплатной студенческой лицензии [1].

Был также использован UI-фреймворк Avalonia, распространяемый по лицензии MIT [2]

1. Язык программирования Python, использованный для реализации сервиса обнаружения лиц.

Среда разработки:

PyCharm Professional, приобретённая по бесплатной студенческой лицензии

и PyCharm Community, распространяемая бесплатно [3].

Основой сервиса является библиотека open-cv, распространяемая по лицензии MIT [4].

Был также использован фреймворк Flask, распространяемый свободно [5].

1. Язык программирования Go, использованный для реализации сервиса вырезки лиц.

Среда разработки: GoLand, приобретённая по бесплатной студенческой лицензии [6].

Выбранные технологии с точки зрения развёртывания:

Для ускорения и частичной автоматизации процесса реализации распределённой системы, нами были использованы следующие программные обеспечения:

1. Docker, распространяемый свободно [7]. Используется для контейнеризации приложения. Для работы с Docker использовалось его desktop-приложение.
2. Kubernetes, распространяемый по лицензии Apache Licence 2.0 [8].

Базы данных и СУБД не были использованы в виду отсутствия необходимости. Для взаимодействия между компонентами использовались протоколы HTTP [9] и protobuf [10].

# Сценарии использования

Для взаимодействия с приложением пользователь использует его клиентскую часть.

Клиенту необходимо знать адрес сервиса обнаружения лиц, к которому он будет обращаться. Если определена переменная окружения SERVER\_HOST, то приложение считает её значение и покажет на интерфейсе пользователю. В интерфейсе пользователь может менять этот адрес и задавать его самостоятельно, в случае отсутствия переменной окружения SERVER\_HOST. На Рисунке 1 изображена команда, вводимая для запуска клиента с указанием этой переменной окружения.

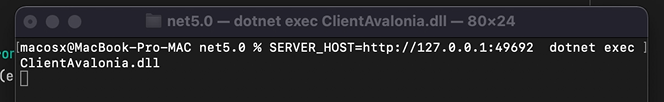


Рисунок 1 - Запуск клиента с определением переменной окружения SERVER\_HOST

Результат вызова данной команды представлен на Рисунке 2.



Рисунок 2 - Интерфейс клиента и автоматически считанное им значение переменной окружения SERVER\_HOST

Вернёмся в терминал и запустим клиентское приложение снова, в этот раз не указывая переменную окружения, как на Рисунке 3.

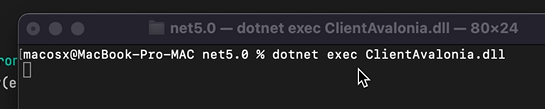


Рисунок 3 - Запуск клиента без переменной окружения SERVER\_HOST

Результат такого вызова показан на Рисунке 4.

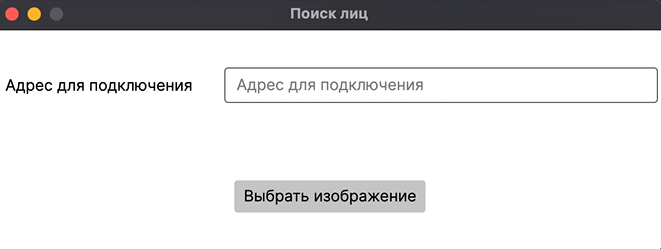


Рисунок 4 - Результат запуска клиента без переменной окружения SERVER\_HOST

Если пользователь попытается обратиться к сервису обнаружения лиц, не указывая его адрес, то произойдёт проверка «на дурака», что демонстрирует Рисунок 5.

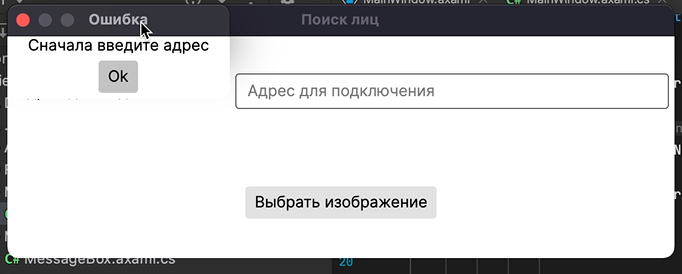


Рисунок 5 - Поведение клиента в случае отсутствия адреса сервиса обнаружения лиц

Для передачи изображения, клиенту, на котором пользователь хочет обнаружить лица, пользователь может воспользоваться двумя способами:

1. С помощью интерфейса клиентской части выбрать необходимое ему изображение; Соответствующее окно представлено на Рисунке 6.

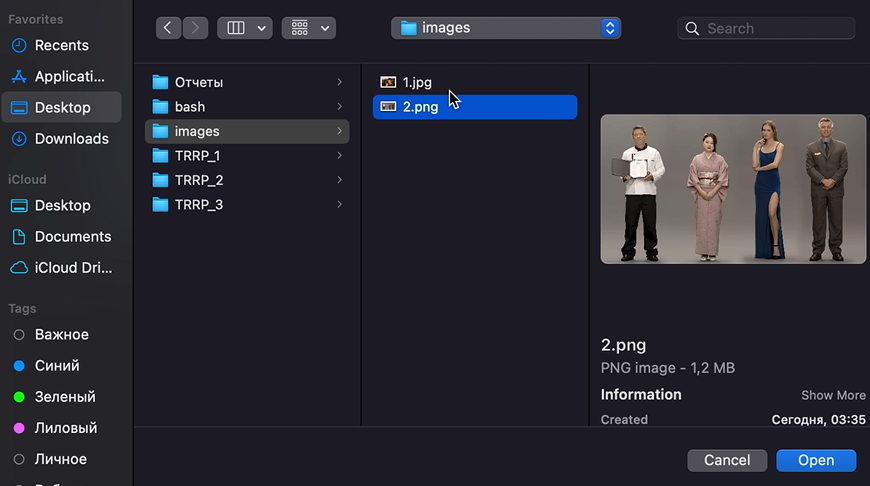


Рисунок 6 - Окно выбора необходимого пользователю изображения

1. Прописать полное имя (путь + краткое имя) изображения в качестве параметра, вызывая клиентскую часть через терминал. Данный вызов и ответ клиента на него представлены на Рисунке 7.

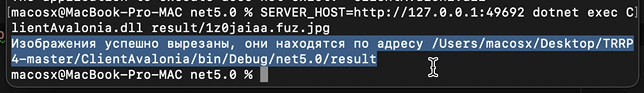


Рисунок 7 - Работа с клиентом только через терминал

В результате, если пользоваться интерфейсом, то приложение откроет нам папку с вырезанными лицами, а если пользоваться терминалом, то в эту папку необходимо перейти самостоятельно, воспользовавшись путём, которое вывело приложение, что показывает Рисунок 8.

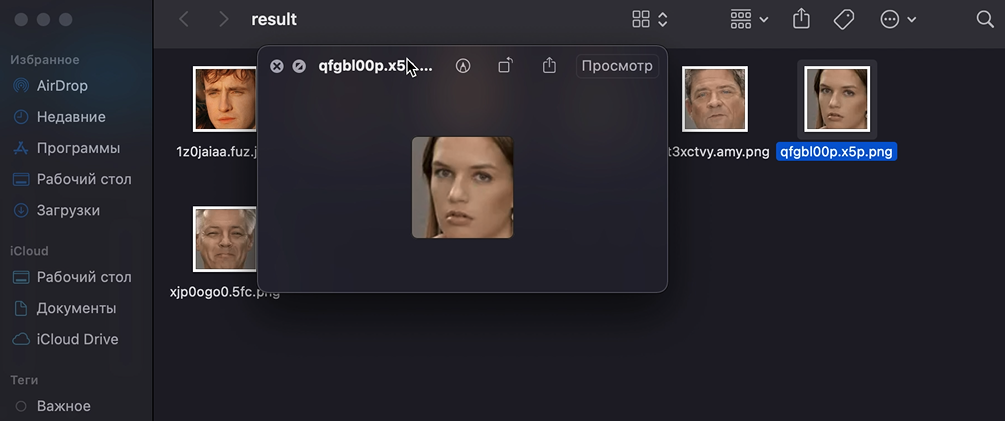


Рисунок 8 - Результат работы распределённого приложения

Дополнительно, на Рисунке 9 покажем, что произойдёт, если ввести некорректный адрес сервиса обнаружения лиц.

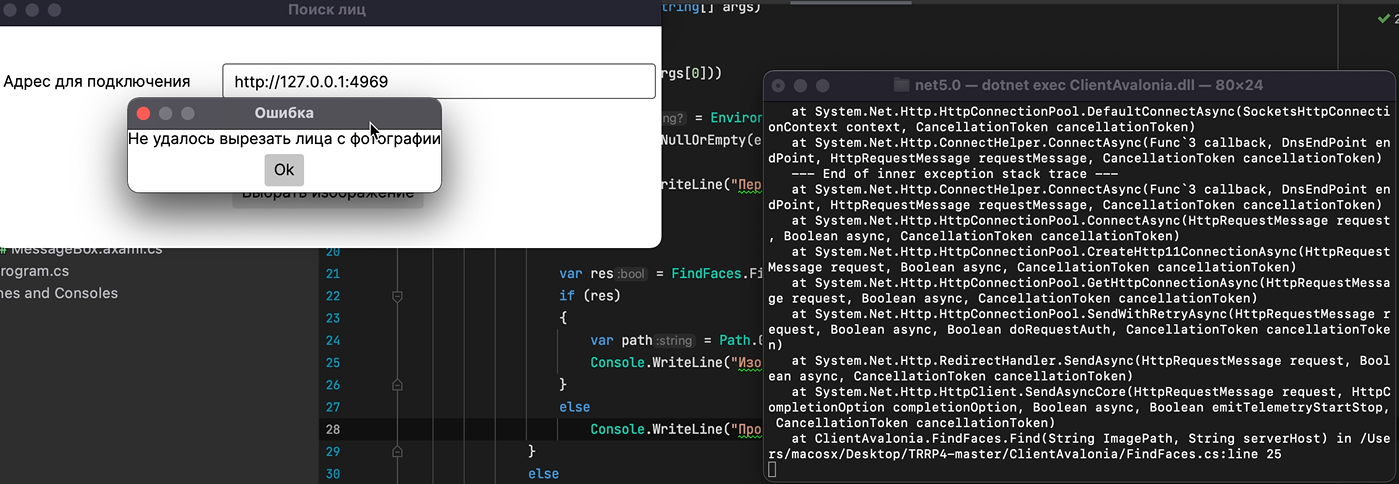


Рисунок 9 - Поведение клиента в случае указания неверного адреса сервиса обнаружения лиц

# Процесс развёртывания

Подробно опишем процесс развёртывания реализованного приложения при помощи Docker и Kubernetes.

## Первичный запуск распределённого приложения

Запускаем приложение Docker. Затем открываем терминал и с помощью команды

**minikube start**

развёртываем урезанную версию Kubernetes-кластера, называемую Minikube. Покажем это на Рисунке 10.

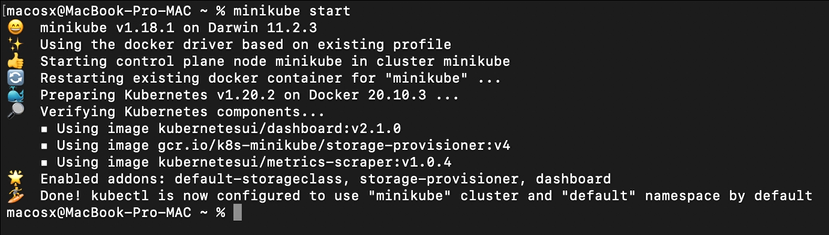


Рисунок 10 - Поведение терминала при запуске кластера

В условиях этого кластера мы и будем развёртывать наше приложение. Minikube запускает Docker-контейнер и будет использоваться как виртуальная машина (или виртуальный сервер), на котором запущен кластер Kubernetes.

Далее необходимо переключиться на Docker Daemon нашего виртуального сервиса, чтобы действия в результате выполнения команды

**docker** **<конкретная команда и её параметры>**

не происходили локальном Docker нашего компьютера, используя команду

**eval $(minikube -p minikube docker-env)**

Docker Daemon является сервером Docker, управляющим контейнерами, образами и т.п., и ожидающий запросы к API Docker. Вызов данной команды в терминале представлен на Рисунке 11.

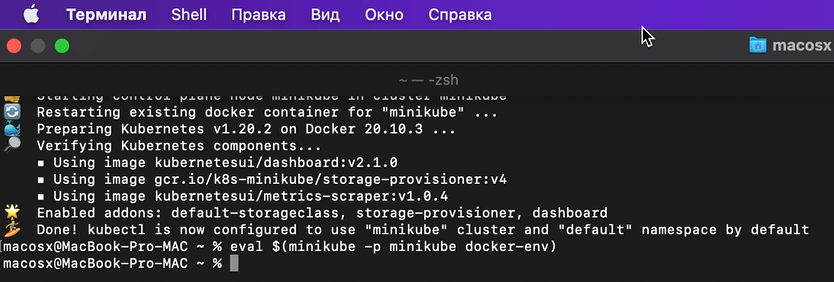


Рисунок 11 - Поведение терминала при запуске команды **eval**

Для отслеживания текущего состояния кластера в отдельном терминале выполним команду

**minikube dashboard**,

открывающую web-интерфейс Kubernetes Dashboard, показывающий состояние нашего кластера на Рисунках 12-14. Подробнее указанные в названиях рисунков сущности будут описаны позднее в этом разделе.

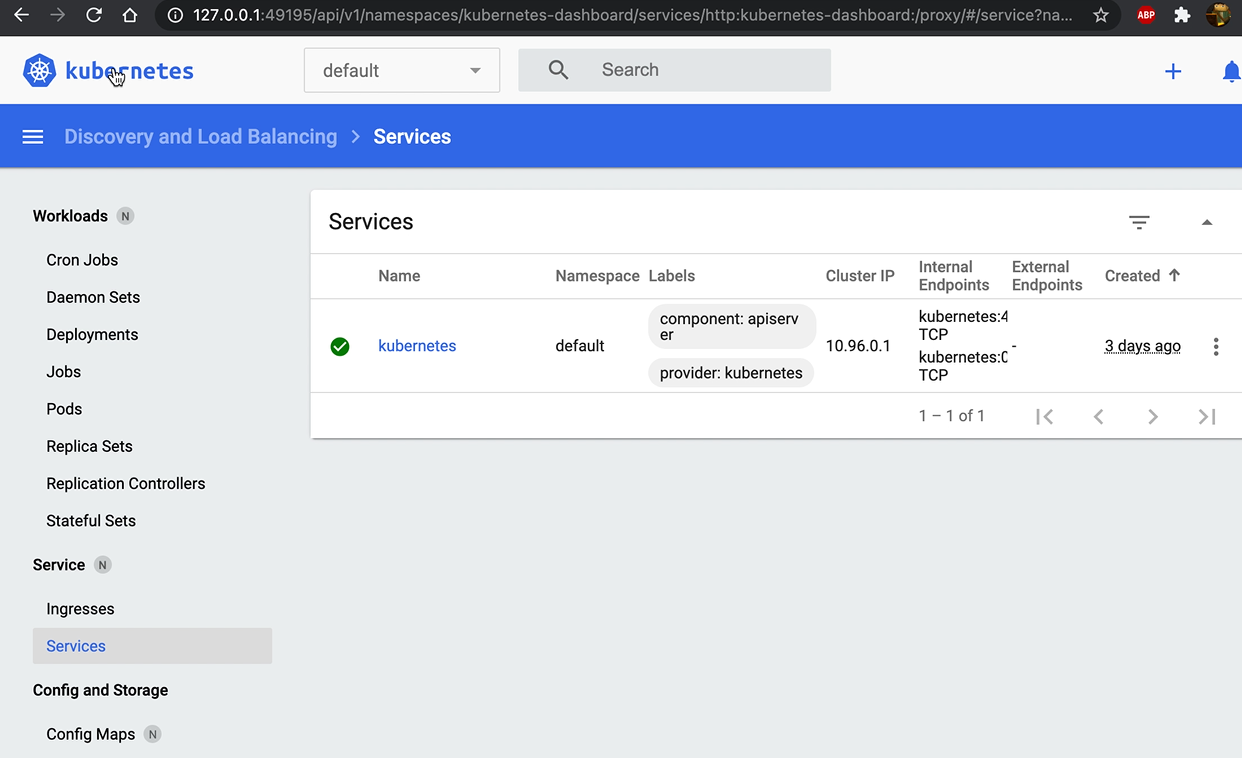


Рисунок 12 - Состояние сущности Services. "kubernetes" это базовый объект этой сущности, создаваемый для каждого кластера по-умолчанию

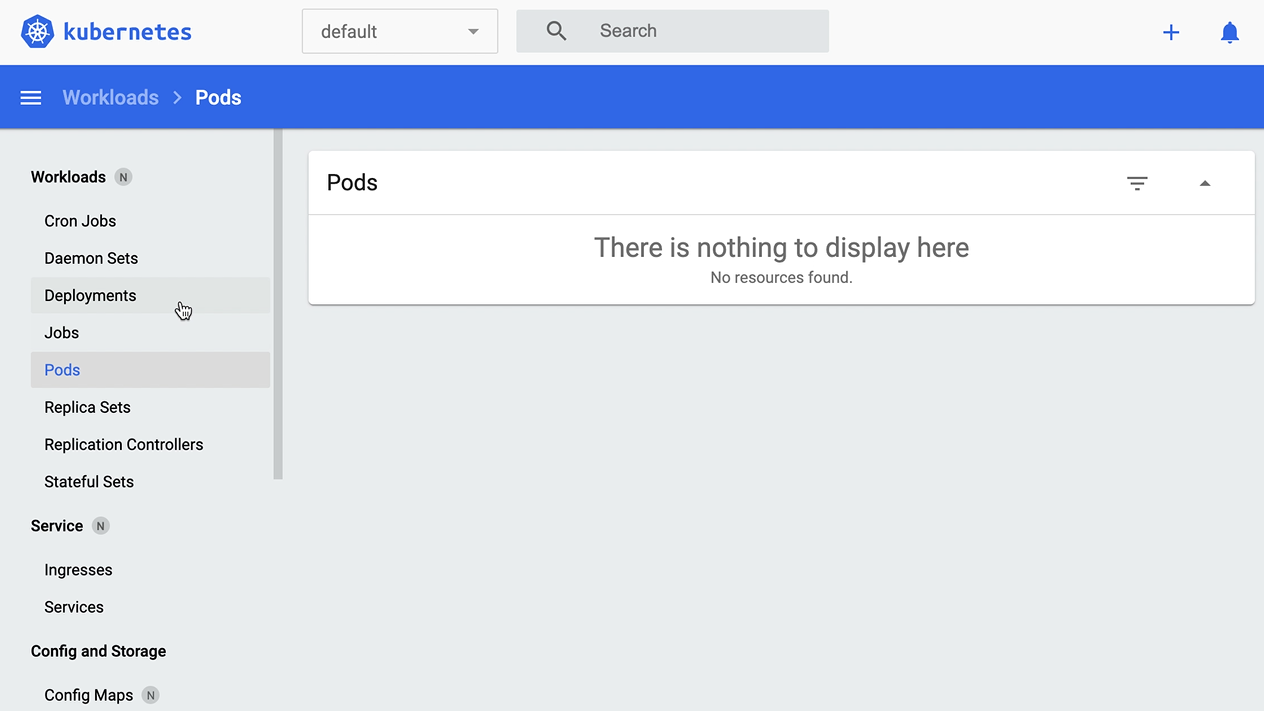


Рисунок 13 - Состояние сущности Pods, отвечающей за экземпляры приложений. Запущенные экземпляры отсутствуют

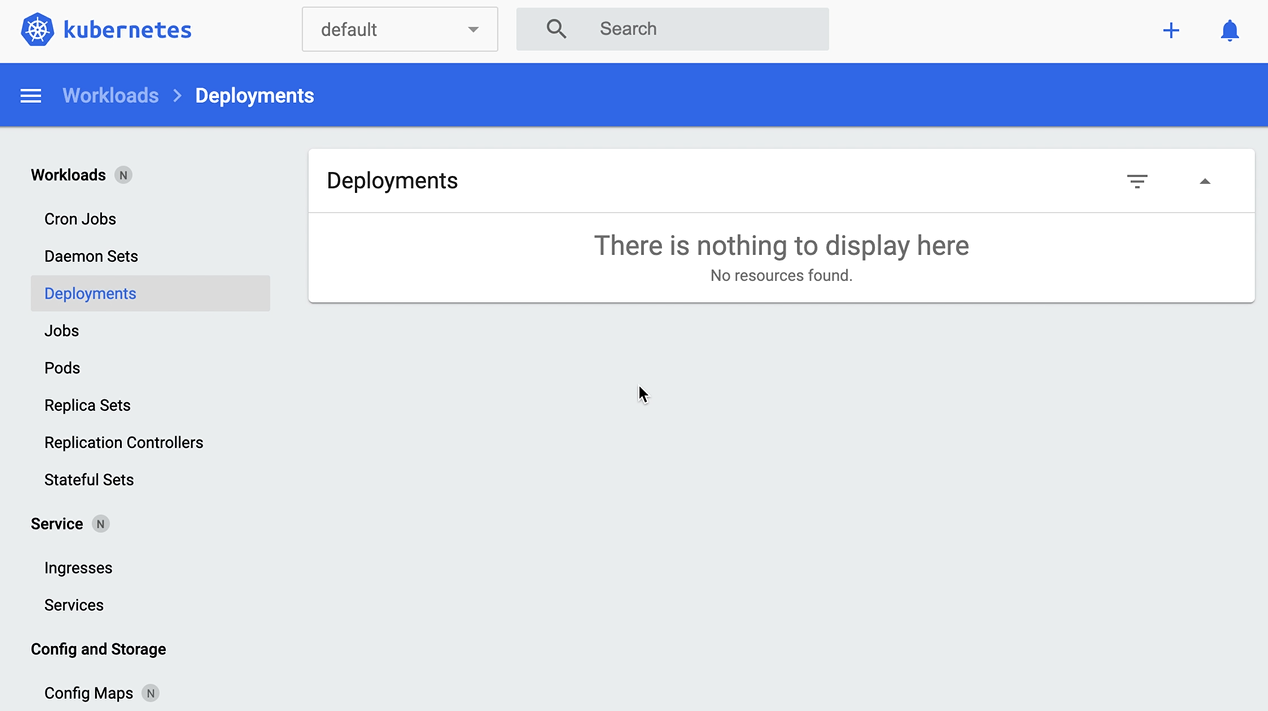


Рисунок 14 - Состояние сущности Deployments, отвечающей за развёрнутые сервисы. Такие сервисы отсутствуют

Вернёмся в предыдущий терминал. Следующим шагом является создание образов для всех сервисов. В нашем случае их два, сервис обнаружения лиц (Faces) и сервис вырезки лиц (Crop). Образ (Image) является базовой сущностью Kubernetes и представляет собой некоторую информацию о системе, включая операционную систему, установленные в неё программы, библиотеки и т.п. Образы создаются на основе определённых базовых уже существующих образов Kubernetes.

Для каждого сервиса создаём специальный файл с названием Dockerfile, в котором мы будем прописывать набор инструкций для будущего образа. Продемонстрируем содержимое Dockerfile обоих сервисов на Рисунках 15-16.

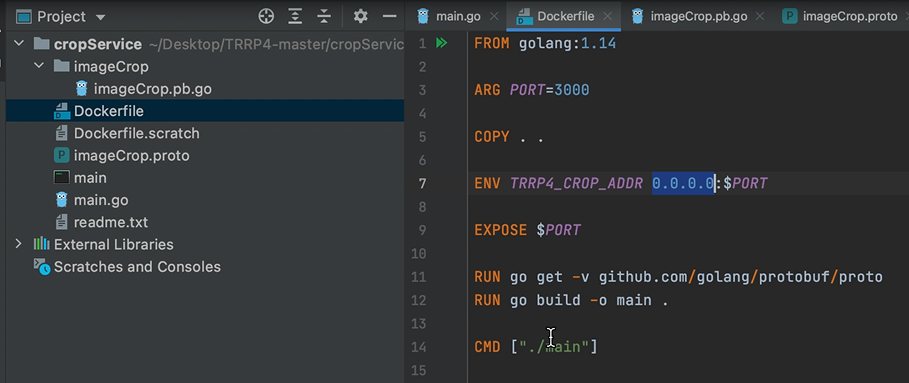


Рисунок 15 - Dockerfile сервиса вырезки лиц

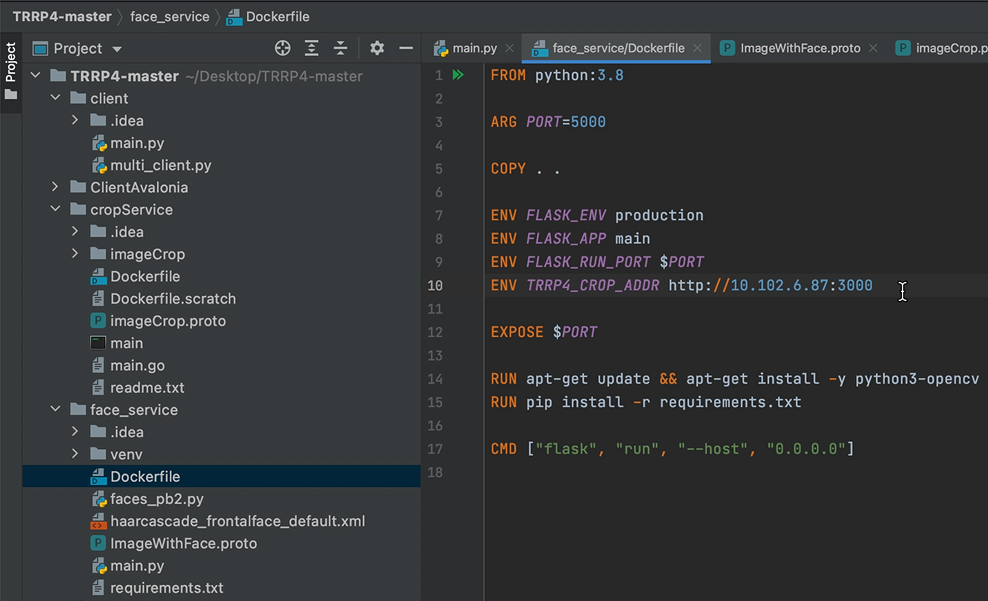


Рисунок 16 - Dockerfile сервиса обнаружения лиц

Инструкция «FROM» указывает на базовый образ, на основе которого мы создаём свой. Инструкция «ARG» показывает, что у контейнера есть аргументы, однако в этой инструкции им можно прописать значения и передавать эти аргументы контейнеру явным способом не нужно. В директиве «COPY» мы указываем, из какого пути на локальном компьютере в какой путь файловой системы контейнера мы переносим данные. В директиве «ENV» указываются переменные окружения. В директиве «EXPOSE» мы указываем порт, по которому будет доступен контейнер. В директиве «RUN» мы указываем выполняемые команды. В качестве команд для выполнения у crop-сервиса мы указываем установку зависимостей для протокола сериализации protobuf и команду компилирования исходного кода на Go. В качестве команд для выполнения у faces-сервиса мы указываем установку зависимостей для Computer Vision и необходимых для Python библиотек. Инструкция «CMD» указывает, что запускается при запуске контейнера из нашего будущего образа. Для создания образа используем команду

**docker build DOCKERFILE -t TAG**,

где в качестве DOCKERFILE мы указываем либо наш Dockerfile, либо папку, его содержащую, а в качестве TAG мы устанавливаем тэг для будущего образа с обязательным указанием его версии. Поведение терминала при вводе этой команды изображено на Рисунках 17-18.

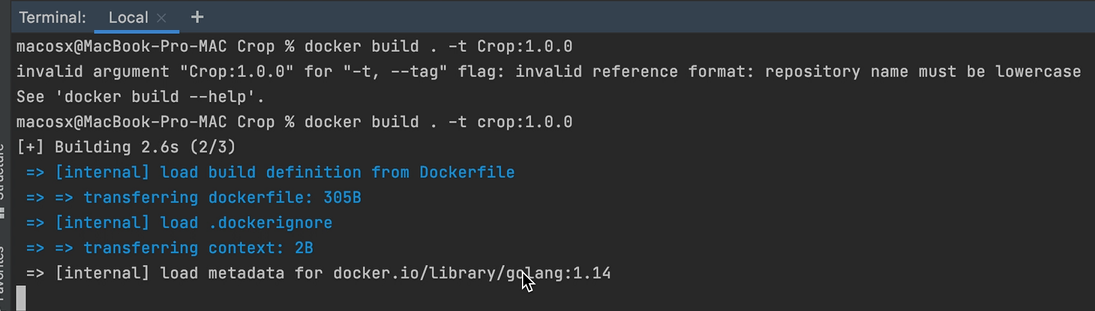


Рисунок 17 - Состояние терминала в процессе создания образа сервиса вырезки лиц



Рисунок 18 - Состояние терминала после окончания создания образа сервиса вырезки лиц

Воспользуемся утилитой kubectl, идущей вместе с Kubernetes, для развёртывания в Kubernetes ранее созданного образа с помощью команды

**kubectl create deployment NAME --image=IMAGE**,

где в качестве NAME указывается имя сервиса, а в IMAGE указывается необходимый образ, заданный через тэг ранее. Таким образом был создан Kubernetes-контейнер. Контейнер является уже развёрнутым образом. У контейнеров становятся доступны файлы логирования, просмотр файловой системы, изменение переменных окружения и т.п. Продемонстрируем введённую команду на Рисунке 19.



Рисунок 19 - Состояние терминала при развёртывании образа сервиса вырезки лиц

После выполнения этой команды создастся объект второй базовой сущности Kubernetes, называемой Deployments. Эта сущность контролирует экземпляры нашего приложения, занимается автоматическим перезапуском экземпляров в случае каких-либо нарушений. В данном случае для одного Deployment создался один соответствующий ему объект третьей базовой сущности Kubernetes, называемый Pods. Объекты Pods и являются экземплярами наших сервисов. При появлении ошибки в экземпляре, Deployment удалит соответствующий Pod экземпляра и создаст новый.

Четвёртая сущность Kubernetes это Service, которая контролирует доступ к нашим приложениям извне. Service позволяет присвоить нашему сервису IP-адрес. Для создания такой сущности воспользуемся командой

**kubectl expose deployment NAME --port=PORT --type=NodePort**,

где в качестве NAME мы указываем имя сервиса, а в PORT – порт, на котором сервис будет слушать информацию, как это показано на Рисунке 20.

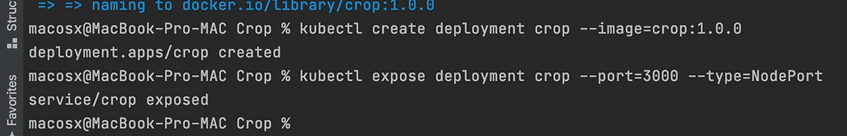


Рисунок 20 - Состояние терминала при выдаче IP-адреса сервису вырезки лиц

Далее для всех сервисов необходимо повторить эти действия, начиная с команды создания образа. При развёртывании сервисов в нашем случае необходимо соблюдать определённый порядок: сначала crop-сервис, а затем уже faces-сервис, что обусловлено тем, что faces-сервис слушает crop-сервис по определённому адресу т.к. пользуется им при своей работе. Следовательно, crop-сервис на момент развёртывания faces-сервиса уже должен быть развёрнут. Адрес crop-сервиса мы указываем в переменных окружения Dockerfile faces-сервиса, используя тот адрес, который был выдан Kubernetes. На Рисунках 21-25 представлен процесс развёртывания faces-сервиса, выполняющийся аналогичными командами.

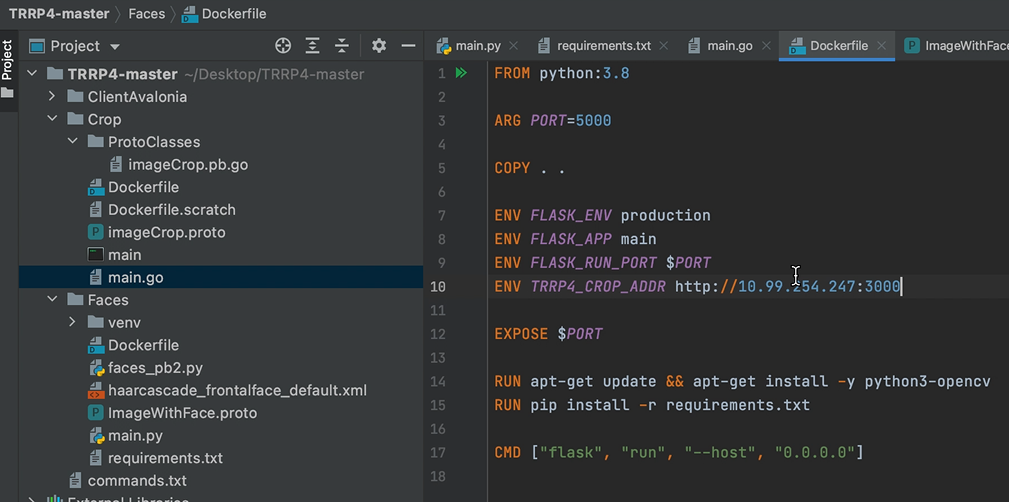


Рисунок 21 - Обновляем переменную окружения TRRP4\_CROP\_ADDR Dockerfile сервиса обнаружения лиц в соответствии с выданным ранее IP-адресом для сервиса вырезки лиц

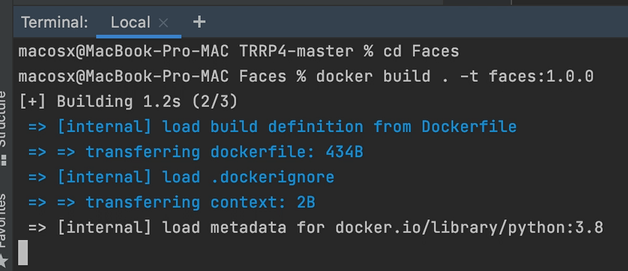


Рисунок 22 - Состояние терминала в процессе создания образа сервиса обнаружения лиц

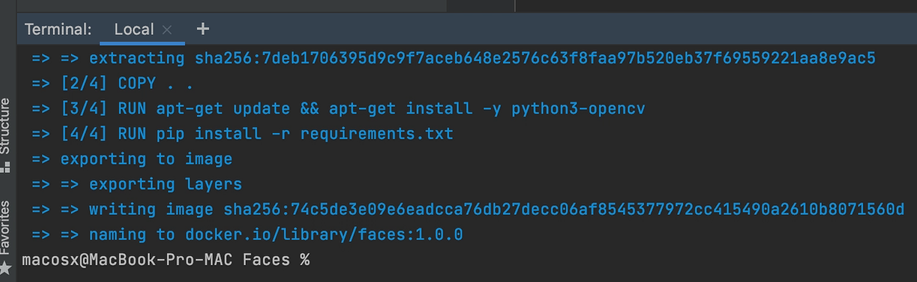


Рисунок 23 - Состояние терминала после окончания создания образа сервиса обнаружения лиц



Рисунок 24 - Состояние терминала при развёртывании образа сервиса обнаружения лиц

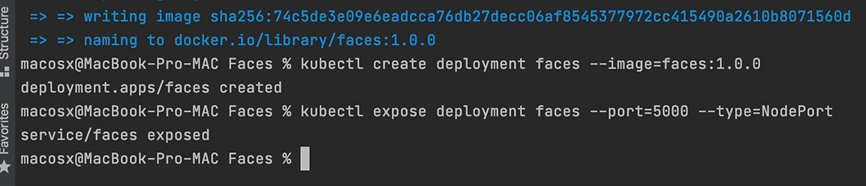


Рисунок 25 - Состояние терминала при выдаче IP-адреса сервису обнаружения лиц

Теперь посмотрим на Kubernetes Dashboard и увидим изменения в кластере. Покажем эти изменения на Рисунках 26-28. Как видно на этих рисунках, для обоих наших сервисов были созданы соответствующие им объекты сущностей Deployments, Pods и Services,

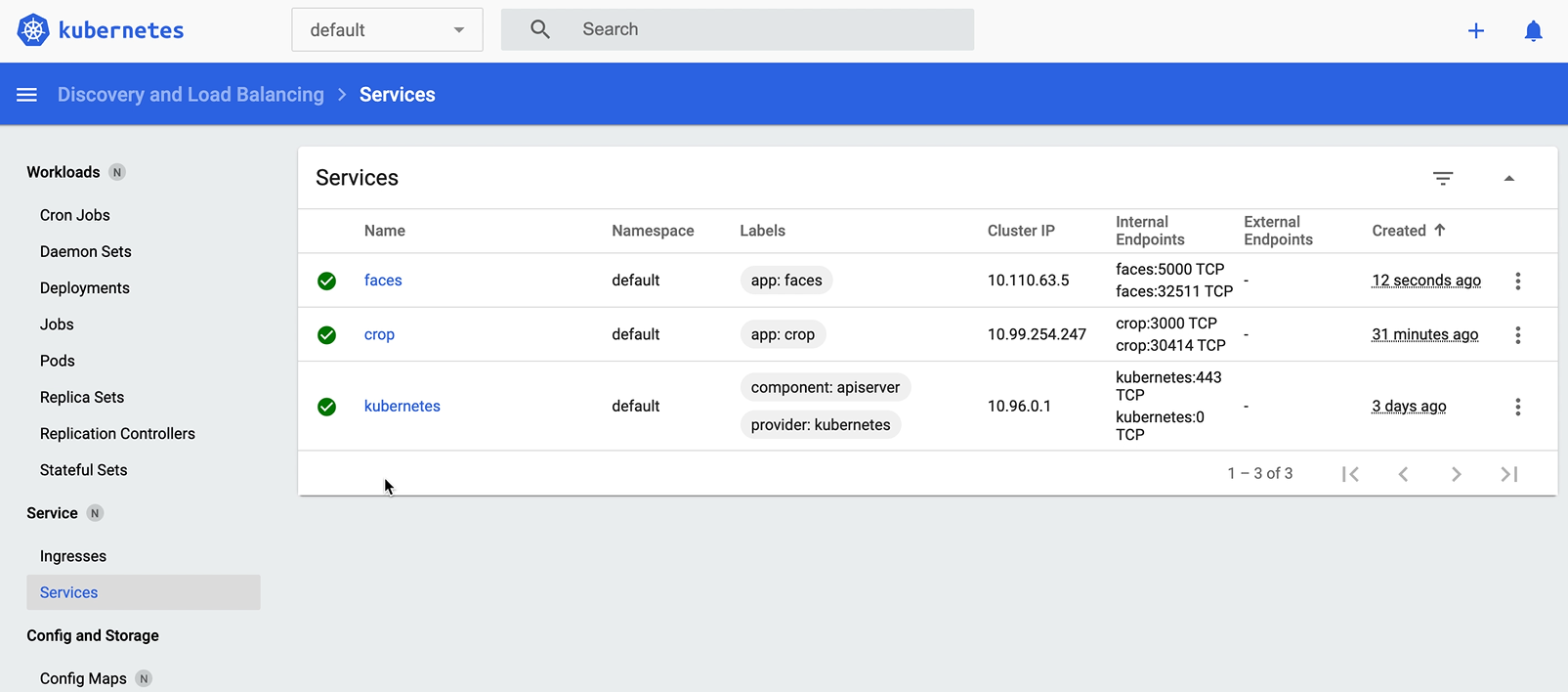


Рисунок 26 - Сервисы, появившиеся в Kubernetes

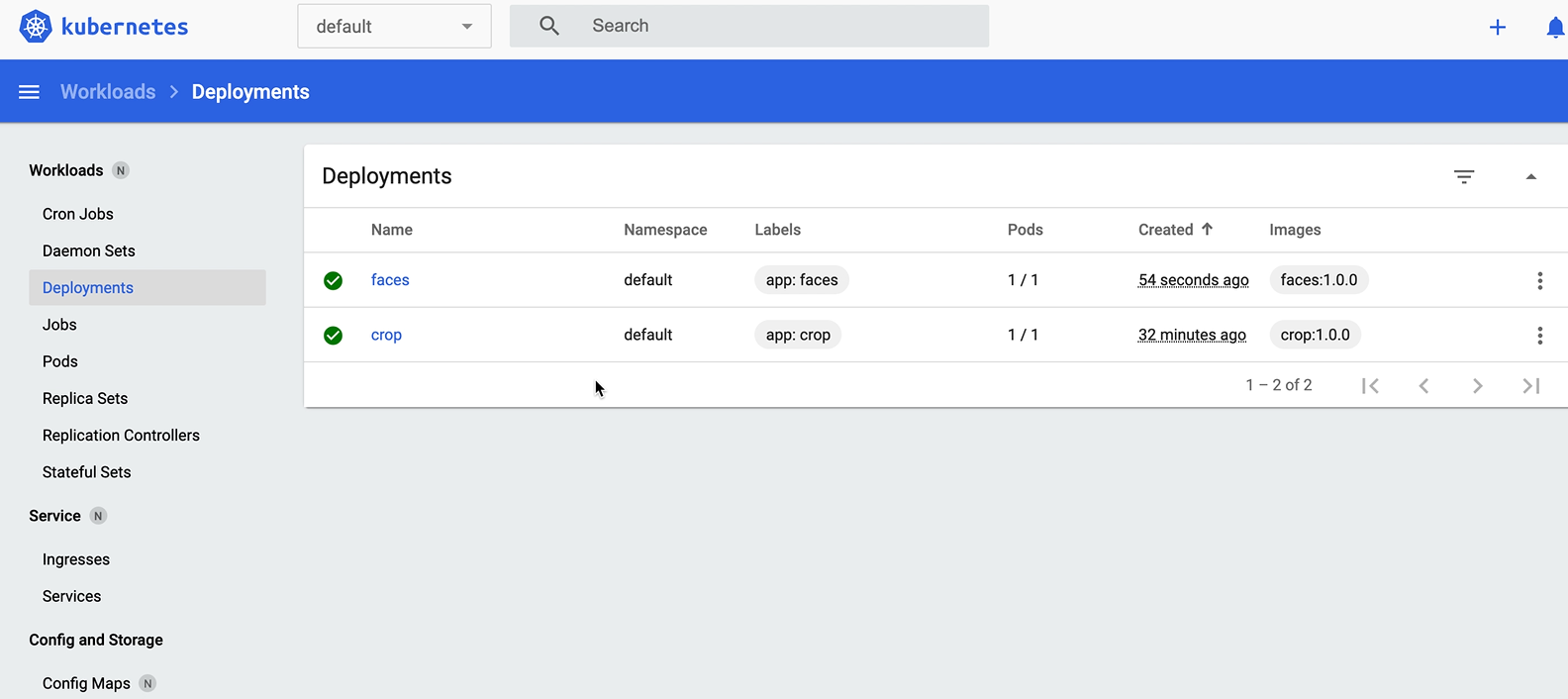


Рисунок 27 - Развёрнутые сервисы в Kubernetes

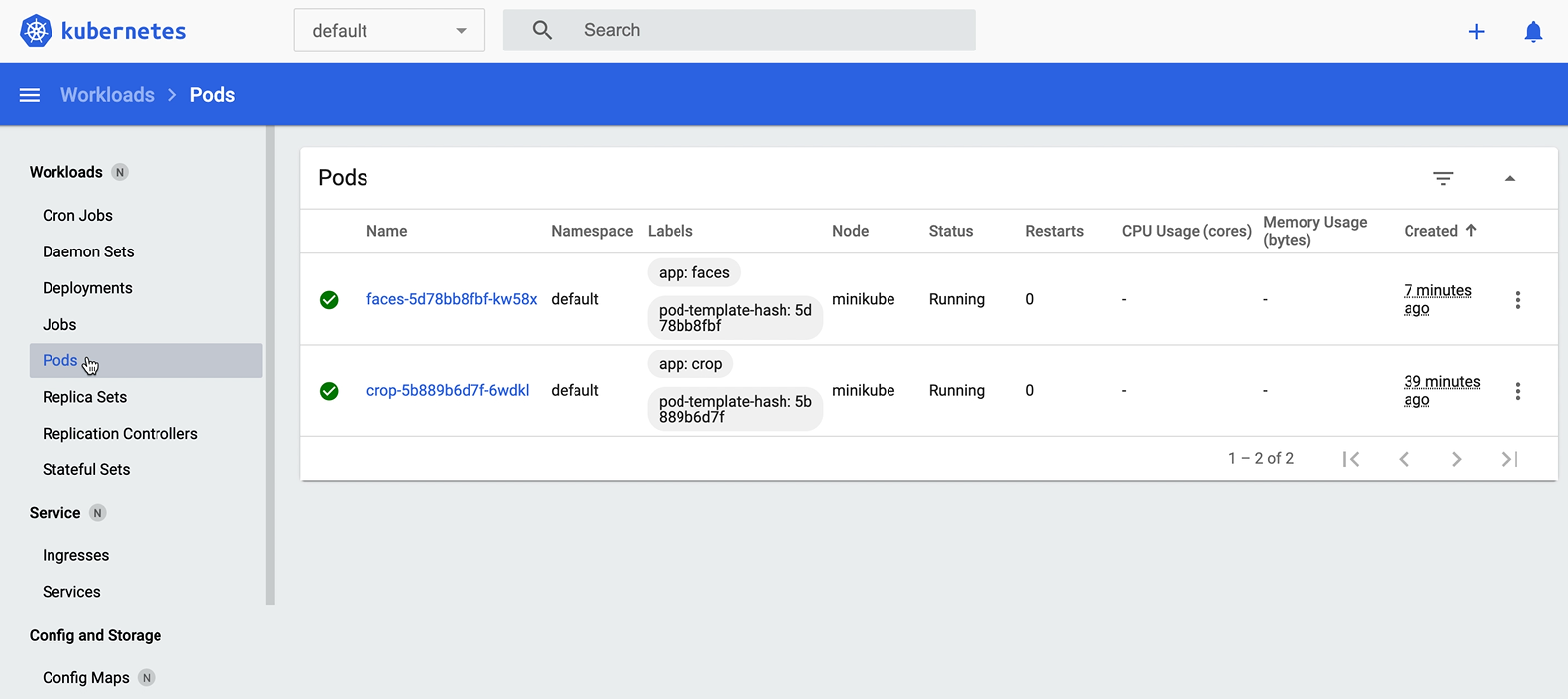


Рисунок 28 - Запущенные экземпляры сервисов в Kubernetes

После развёртывания всех сервисов остаётся выполнить последнюю команду:

**minikube service NAME**

В качестве NAME указываем основной наш сервис (Faces), Данную команду необходимо выполнить из-за ограничений mikikube, связанных с тем, что сервисы недоступны в локальной сети. Образно говоря, для «эмуляции» локальной сети используется данная команда.

На этом процесс развёртывания заканчивается и клиент получает доступ к функционалу распределённого приложения. Внешний вид сообщения, какой был выдан IP-адрес нашему приложению представлен на Рисунке 29.



Рисунок 29 - IP-адрес, к которому необходимо обращаться для использования приложения

## Повторный запуск

Для повторного запуска распределённого приложения в случае выключения minikube, например, с помощью команды

**minikube stop**

Мы увидим, что web-страницы Kubernetes стали недоступны.

Достаточно лишь выполнить команды **minikube start** и **minikube service NAME**, чтобы снова запустить распределённое приложение. Достаточно заново запустить кластер, и работа всех сервисов будет восстановлена автоматически.

При необходимости удаления развёрнутого сервиса можно воспользоваться командами удаления сущностей Deployment и Service:

**kubectl delete deployment NAME** и

**kubectl delete svc NAME**,

где в качестве NAME указывается имя сервис (а не тэг образа).

# Соответствие критериям оценивания

Некоторые критерии оценивания уже были описаны ранее, поэтому повторяющаяся информация в этом разделе будет описана только кратко.

## Проектирование с учётом особенностей предметной области

Распределённое приложение было спроектировано с учётом особенностей предметной области. Процессы обнаружения и вырезки лиц были распределены между двумя отдельными сервисами: сервисом обнаружения лиц и сервисом вырезки лиц. Задачей сервиса обнаружения лиц является лишь обнаружение человеческих лиц на переданном ему изображении в виде переменных точек левого верхнего и правого нижнего углов для каждого лица. Эти данные сервис обнаружения лиц передаёт сервису вырезки лиц, который вырезает лица из изображения по полученным координатам углов, создавая файлы изображений для пользователя.

## Оптимальность архитектуры

Распределённое приложение имеет микросервисную архитектуру, что обусловлено следующими причинами:

1. Высокая масштабируемость и расширяемость микросервисной архитектуры, необходимые для обработки изображений. Это связано с потенциальной трудоёмкостью процесса обработки изображений, что может потребовать большое количество вычислительных ресурсов;
2. Потенциальное развитие функционала приложения в виде добавления новых функций обработки приложений. Такое развитие нетрудно поддерживать именно благодаря микросервисной архитектуре. В приложении одна точка входа. Можно добавлять большое количество новых сервисов, выполняющих разнообразные функции обработки изображений, помимо имеющихся двух.

## Параллельная работа нескольких клиентов и серверов

Для реализации данного требования, а также последующих требований, использовалось программное обеспечение Kubernetes. При помощи Kubernetes можно контролировать количество экземпляров сервисов, используя команду

**kubectl scale deployment NAME --replicas=COUNT**,

где в качестве NAME мы указываем имя сервиса, а в COUNT – количество необходимых для запуска его экземпляров. Например, для обоих сервисов создадим по два экземпляра, а не по одному, как раньше. Покажем, как это делается в терминале на Рисунке 30.

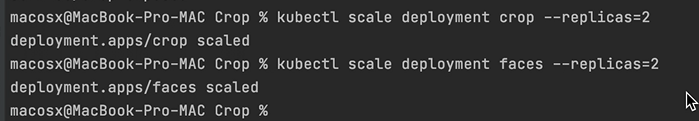


Рисунок 30 - Увеличение количества экземпляров для crop-сервиса, а затем для faces-сервиса

Откроем Kubernetes Dashboard и убедимся в том, что у нас на кластере теперь четыре Pods, соответствующие четырём запущенным экземплярам на Рисунке 31.

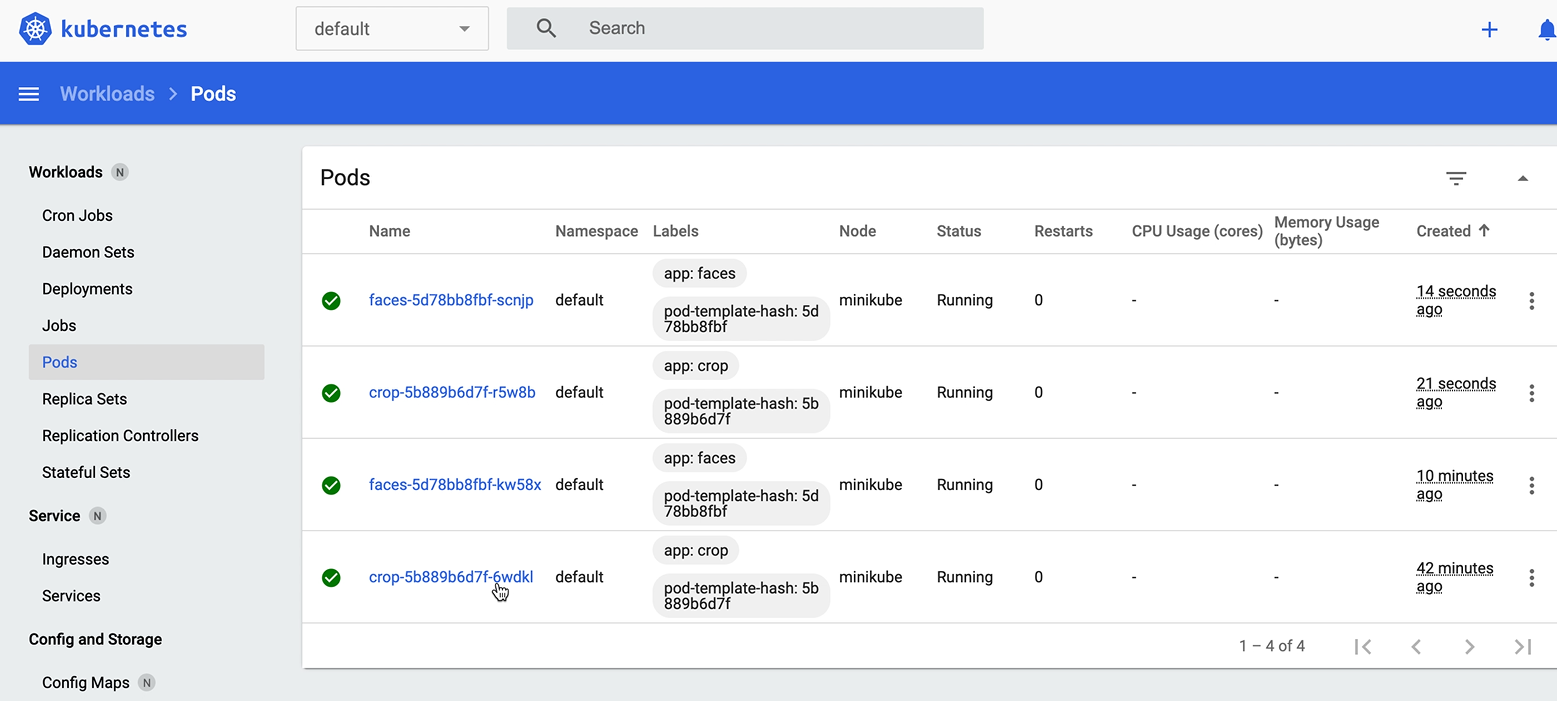


Рисунок 31 - Наличие двух сущностей Pods для каждого сервиса

Кроме того, в Deployments также увеличилось количество экземпляров класса для каждого сервиса, показываемое в столбце «Pods». Как видно на Рисунке 32, теперь их по 2 и активны тоже 2 из них. Ранее, на Рисунке 27 показывалось «1/1».

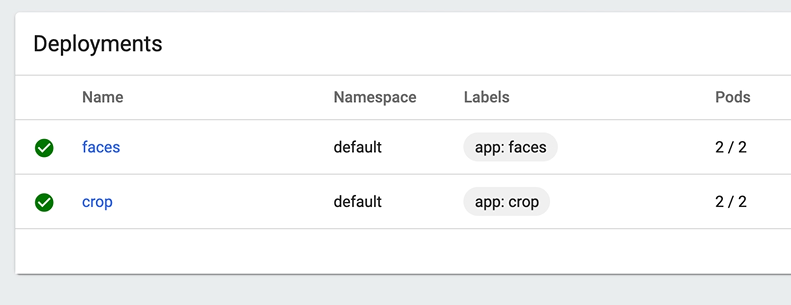


Рисунок 32 - Обновлённое состояние объектов сущностей Deployments

Kubernetes позволяет автоматически управлять конфигурацией системы. Во вкладке с объектами сущности Pods можно увидеть запущенные экземпляры приложений и их состояния, а также посмотреть файлы логирования, что показано на Рисунке 33.

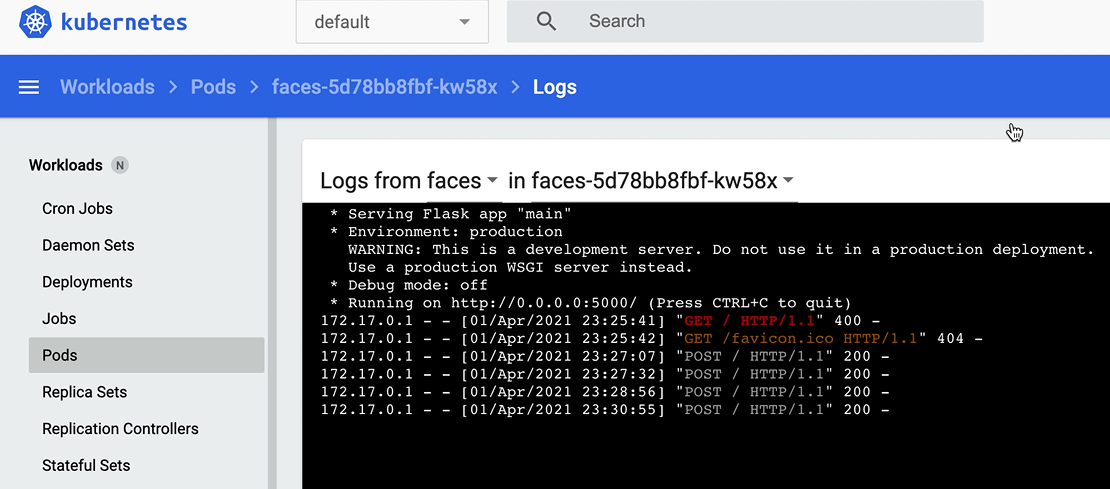


Рисунок 33 - Результат обращения к сервису ображения лиц четыре раза. Соответственно видно четыре записи с ответом 200

На Рисунке 33 был изображен результат работы приложения ещё до создания нескольких экземпляров. Теперь обратимся к сервису уже имея несколько экземпляров. Обратимся к нему через терминал несколько раз подряд, не останавливаясь, как на Рисунке 34.

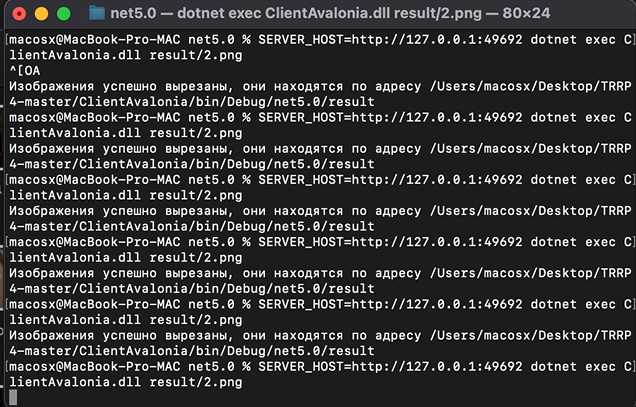


Рисунок 34 - Частый запуск клиента через терминал

Теперь откроем файлы логирования всех экземпляров и покажем их содержимое на Рисунках 35-37.



Рисунок 35 - Файл логирования нового экземпляра faces-сервиса

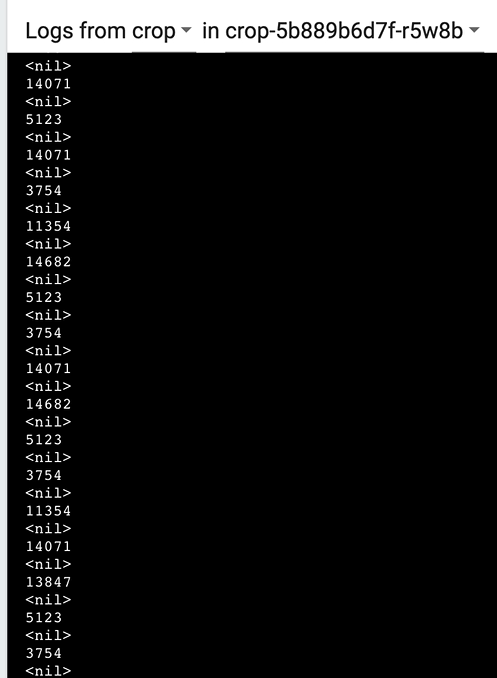


Рисунок 36 - Файл логирования нового экземпляра crop-сервиса. Выводятся размеры сгенерированных файлов с лицами в байтах

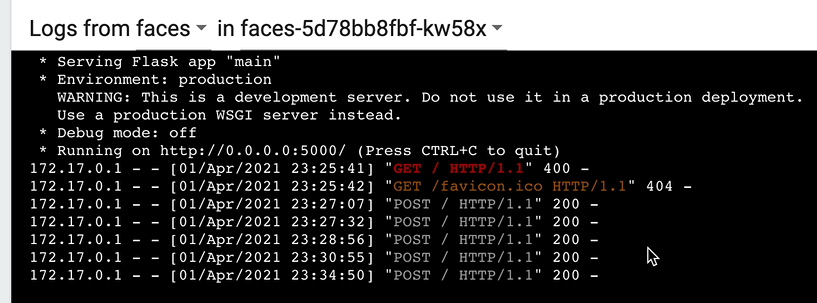


Рисунок 37 - Файл логирования старого экземпляра faces-сервиса. Появилась новая, пятая запись

## Масштабируемость приложения

Как было указано ранее, микросервисная архитектура позволяет отлично масштабировать распределённое приложение, добавляя новые и новые сервисы, а затем развёртывая их и определяя количество необходимых экземпляров в любой момент.

Гибкое управление приложением доступно при помощи web-интерфейса Kubernetes Dashboard, благодаря которому можно не вводить ранее описанные команды, а использовать элементы интерфейса. Kubernetes также предоставляет диаграммы балансировки нагрузки между экземплярами, что показано на Рисунке 38.

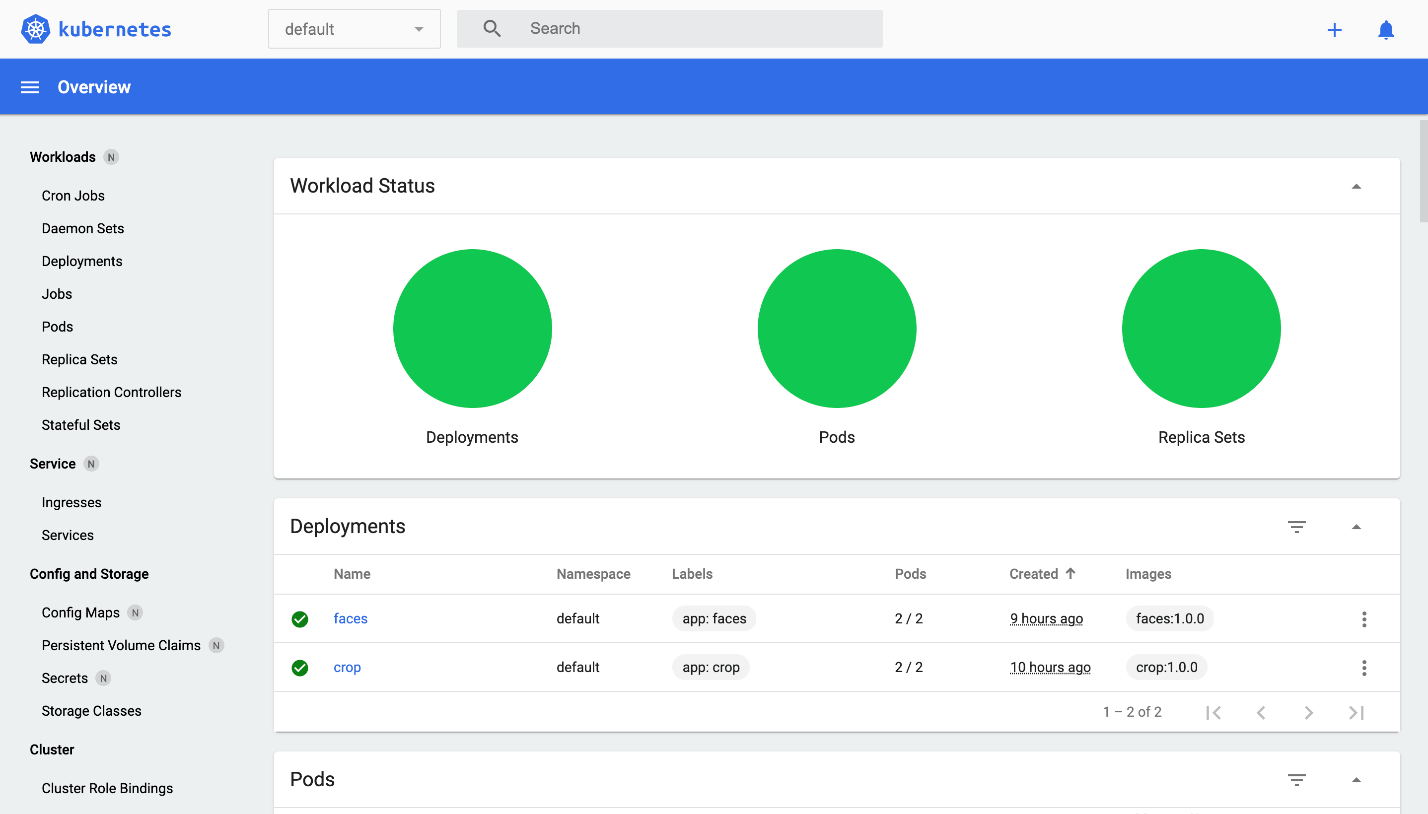


Рисунок 38 - Мониторинг состояния распределённого приложения

## Динамическое реконфигурирование

Реконфигурирование системы происходит с помощью конфигурационных JSON-файлов web-интерфейса Kubernetes Dashboard управления сущностями Deployments и Services, в которых можно менять и переменные окружения; Пример такого файла предоставлен на Рисунке 39.

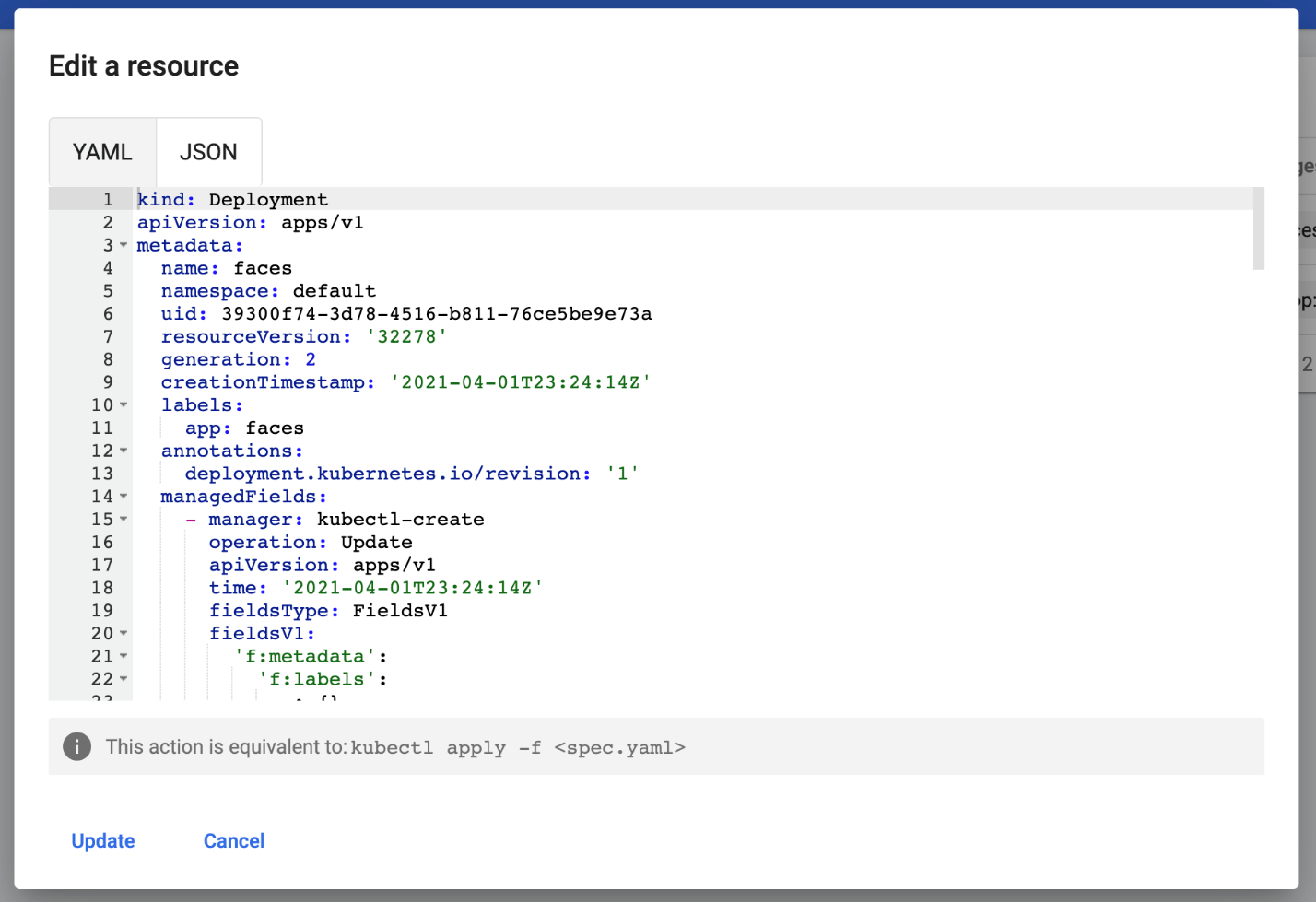


Рисунок 39 - Конфигурационный файл Deployment сущности faces-сервиса

## Средства коммуникации

Для организации взаимодействия компонентов используется только одно средство коммуникации: HTTP-запросы. Выбор данного средства обусловлен простотой его реализации и достаточностью для работоспособности системы.

## Отказоустойчивость системы

Kubernetes следит за состоянием отдельных компонентов приложения, используя сущность Deployments. В случае обнаружения какой-либо ошибки в работе любого компонента, технология перезапускает этот экземпляр, создавая для него новую сущность Pods. Для проверки удалим один из экземпляров. Это можно сделать либо с помощью команды, либо с помощью интерфейса, как на Рисунке 40.



Рисунок - Удаление одного из экземпляров

В процессе удаления web-интерфейс на какое-то время покажет и старый удалённый экземпляр, пометив его другой иконкой, и автоматически созданный новый экземпляр, заменяющий удалённый, что показано на Рисунке 41.

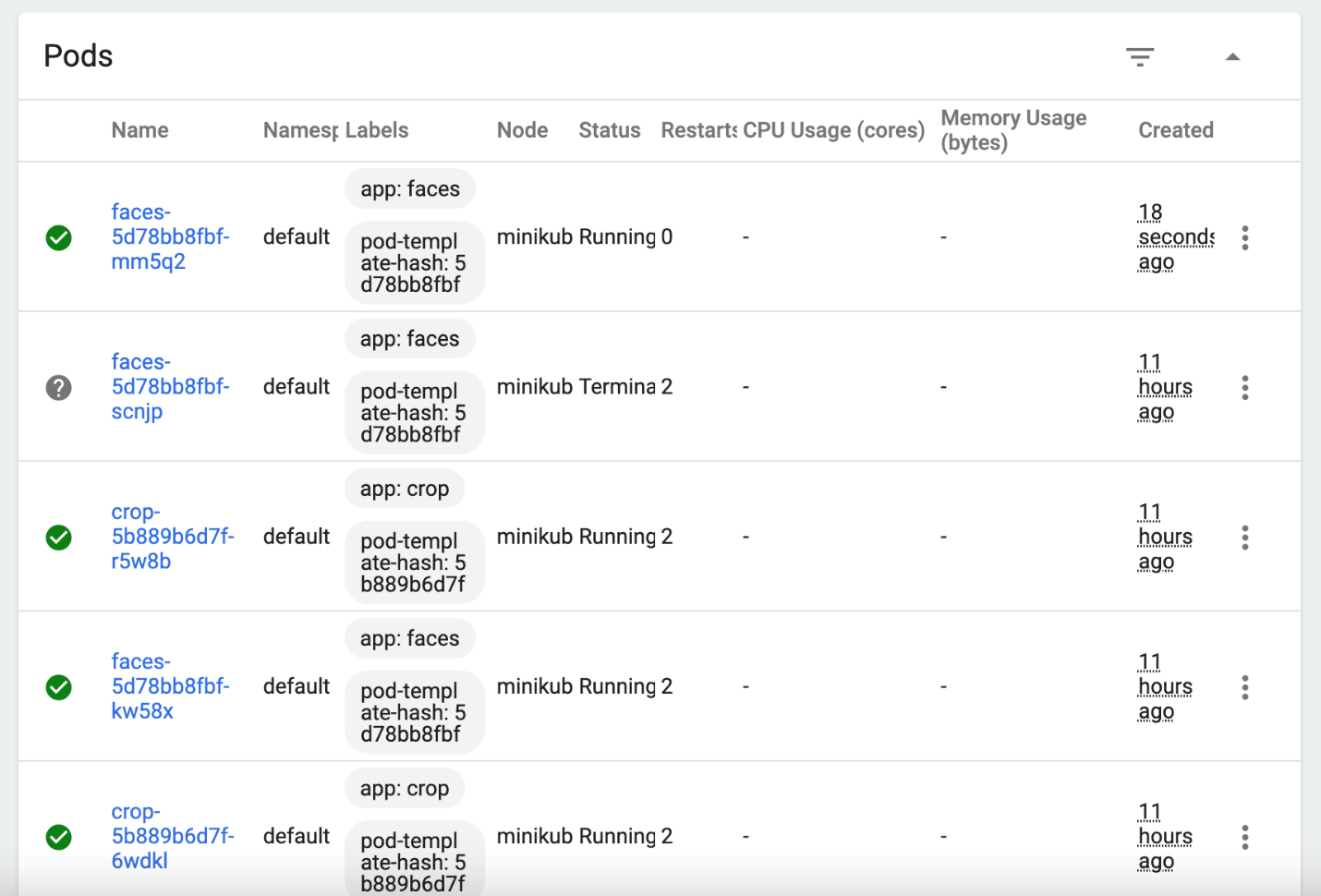


Рисунок - Состояние экземпляров в процессе удаления экземпляра и создания нового для замены

Спустя короткое время удаляемый экземпляр полностью пропадёт из списка экземпляров, как видно на Рисунке 42.



Рисунок - Удалённый экземпляр окончательно был заменён новым автоматически

## Восстановление компонента на другом узле

Новый IP-адрес вновь созданного экземпляра автоматически добавляется системой в балансировщик нагрузки. Новые запросы уже будут идти не к старому экземпляру и соответствующей ему сущности Pods, а к новым созданным экземпляру и Pod. Клиент отправляет запросы сущности Deployment, а уже она определяет, к какому конкретному Pod обращаться.

## Описание архитектуры каждого компонента

Компоненты приложения и их архетуктура:

### Клиент

Представляет собой приложение, написанное на языке программирования C# с использованием UI-фреймворка Avalonia. Исходный код для элемента управления «MessageBox» был позаимствован от пользователя kekekers на сайте Stackoverflow [11]. Данный исходный был выложен в открытый доступ для свободного пользования. Задача клиента – обратиться к сервису обнаружения лиц и взаимодействовать с пользователем.

Работа клиента определяется переменной окружения SERVER\_HOST, задаваемой пользователем, в которой указывается адрес сервиса обнаружения лиц, без которого невозможна работа клиента.

### Сервис обнаружения лиц (faces-сервис)

Представляет собой HTTP-сервер, написанный на языке программирования Python с использованием фреймворка Flask. На сервере реализован обработчик запроса к URL «**/**». Запросы к любым другим URL будут возвращать 404 ошибку HTTP протокола. Задача сервиса – найти лица на изображении и предоставить пользователю обнаруженные лица.

Работа сервиса определяется переменной окружения TRRP4\_CROP\_ADDR, задаваемой разработчиком во время создания образа, в которой указывается адрес сервиса вырезки лиц, без которого невозможна работа сервиса обнаружения лиц.

### Сервис вырезки лиц (crop-сервис)

Представляет собой сервис, написанный на языке программирования Go. Данный сервис обрабатывает только одну точку входа: «**/**». Сервис ожидает данные в сериализованном с помощью протокола protobuf виде. Задача сервиса – создать файлы на основе переданных данных, вырезая куски изображения, и передать их сервису обнаружения лиц.

## Структура передаваемых данных

Сервис вырезки лиц (Crop) получает данные от сервиса обнаружения лиц (Faces) с помощью протокола HTTP. Данные, которыми обмениваются эти сервисы, сериализованы при помощи протокола protobuf. Запрос от Faces к Crop представляет из себя структуру со следующими полями:

1. Само изображение;
2. Координаты (x, y) левого верхнего угла для вырезки;
3. Координаты (x, y) правого нижнего угла для вырезки.

Ответ от Crop к Faces представляет из себя структуру со следующими полями:

1. Обрезанное изображение;
2. Флаг, показывающий успех или неудачу вырезки, а также сообщение, конкретизирующее информацию об неудаче, если она произошла.

На Рисунке 43 представлен внешний вид proto-файла:

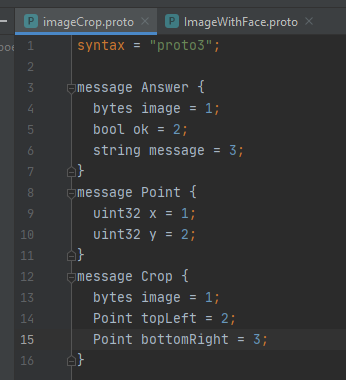


Рисунок 43 - proto-файл

## Описание способа передачи сообщений

Выбранный способ передачи сообщений между компонентами системы – протокол HTTP. Данные, сериализованные при помощи протокола protobuf, передаются внутри тела POST-запроса HTTP.

# Заключение

Таким образом, при реализации приложения четыре из шести поставленных требований были успешно выполнены:

1. Распределенное приложение спроектировано с учетом особенностей предметной области. Выбрана наиболее подходящая модель распределенной системы.
2. Архитектура системы является оптимальной для заданных при разработке критериев. В отчете присутствует обоснование выбора данного типа архитектуры.
3. Приложение обеспечивает параллельную работу нескольких клиентов и серверов, в том числе на одном компьютере. Серверы распределенной системы выполняют различные функции.
4. Приложение является масштабируемым, позволяет добавлять новых участников взаимодействия без переписывания кода и перезапуска приложений.
5. Существует возможность динамического реконфигурирования системы.
6. Для организации взаимодействия компонент распределенной системы используется только одно средство коммуникации (1 балл из 4)
7. Система является отказоустойчивой. В случае если один и/или несколько компонент системы аварийно завершают свою работу.
8. Распределенное приложение продолжает работать и в случае, если после аварийного завершения некоторого компонента, он восстановлен на другом узле вычислительной сети.
9. Отчет содержит подробное описание архитектуры каждого компонента распределенного приложения.
10. В отчете описана структура передаваемых данных, формат сообщений и вид протокола, используемого для этого.
11. В отчете представлено описание способа передачи сообщений при коммуникации компонентов распределенной системы с обоснованием.

Вывод самооценки: каждый студент претендует на 27 баллов из 30, что превышает проходное количество баллов по 100-балльной шкале.

# Библиографический список

1. Rider от JetBrains [электронный ресурс]

URL: <https://www.jetbrains.com/rider/> (дата последнего обращения: 02.04.2021)

1. Avalonia: A cross platform XAML framework for .NET [электронный ресурс]

URL: <https://avaloniaui.net/> (дата последнего обращения: 02.04.2021)

1. PyCharm: The Python IDE for Professional Developers [электронный ресурс]

URL: <https://www.jetbrains.com/pycharm/> (дата последнего обращения: 02.04.2021)

1. Pypi: opencv-python 4.5.1.48 [электронный ресурс]

URL: <https://pypi.org/project/opencv-python/> (дата последнего обращения: 02.04.2021)

1. Pypi: Flask 1.1.2 [электронный ресурс]

URL: <https://pypi.org/project/Flask/> (дата последнего обращения: 02.04.2021)

1. GoLand от JetBrains [электронный ресурс]

URL: <https://www.jetbrains.com/go/> (дата последнего обращения: 02.04.2021)

1. Docker: Empowering App Development for Developers [электронный ресурс]

URL: <https://www.docker.com/> (дата последнего обращения: 02.04.2021)

1. Kubernetes: Production-Grade Container Orchestration [электронный ресурс]

URL: <https://kubernetes.io/> (дата последнего обращения: 02.04.2021)

1. Обзор протокола HTTP [электронный ресурс]

URL: <https://developer.mozilla.org/ru/docs/Web/HTTP/Overview> (дата последнего обращения: 02.04.2021)

1. Protocol Buffers [электронный ресурс]

URL: <https://developers.google.com/protocol-buffers?hl=ru> (дата последнего обращения: 02.04.2021)

1. Stackoverflow: How to show a message box in AvaloniaUI (beta) [электронный ресурс]

URL: <https://stackoverflow.com/questions/55706291/how-to-show-a-message-box-in-avaloniaui-beta> (дата последнего обращения: 02.04.2021)