# Изучение инструмента

## Введение

Kubernetes — это портативная расширяемая платформа с открытым исходным кодом для управления контейнеризованными рабочими нагрузками и сервисами, которая облегчает как декларативную настройку, так и автоматизацию. Kubernetes это не просто ПО, а целая экосистема, в которой K8s (kubernetes) является необходимым минимумом, остальные модули устанавливаются и настраиваются по необходимости.

K8s работает с контейнерами, контейнеры похожи на виртуальные машины, но у них есть свойства изоляции для совместного использования операционной системы (ОС) между приложениями, поэтому контейнеры считаются легкими.

Kubernetes поддерживает несколько сред для запуска контейнеров: Docker, containerd, CRI-O, и любая реализация Kubernetes CRI (Container Runtime Interface).

## Компоненты

Вся работа происходит в кластере, в котором находятся рабочие машины (физические или виртуальные) – узлы, в рабочих узлах размещены поды, являющиеся компонентами приложения.

### Поды

Под (Pod) — это абстрактный объект Kubernetes, представляющий собой группу из одного или нескольких контейнеров приложений, включающую общие используемые хранилище (тома), IP-адрес и информацию по их запуску.

Под может включать в себя различные контейнеры приложений (например, веб-приложение и базу данных, которая необходима для работы приложения), но все контейнеры в поде имеют одни и те же IP-адрес и порт.

Каждый Pod-объект связан с узлом, на котором он размещён, и остаётся там до окончания работы (согласно стратегии перезапуска) либо удаления.

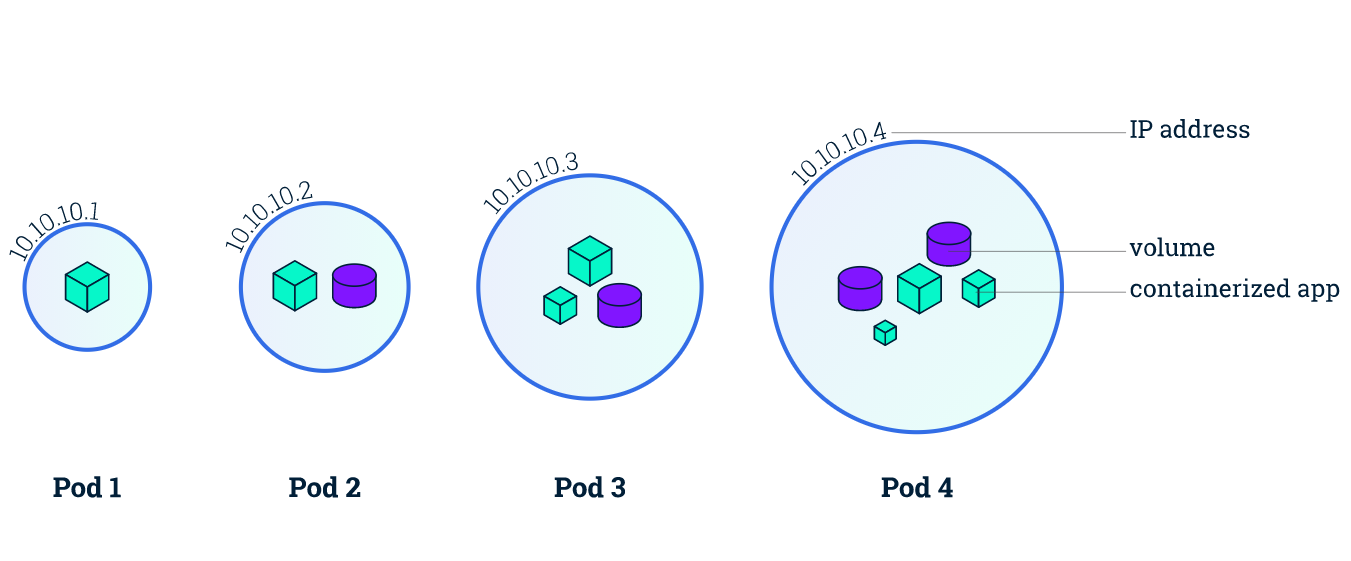


Рисунок 1 – Схема Подов

### Узлы

Узел (Node) - это рабочая машина в Kubernetes, которая в зависимости от кластера может быть либо виртуальной, либо физической.

Среди узлов существует один или несколько (для гарантии отказоустойчивости) ведущих (master) машин, которые отвечают за управление другими (worker) узлами.

Узел может содержать несколько подов, которые мастер Kubernetes автоматически размещает на разные узлы кластера в зависимости от доступных ресурсов на каждом узле (автоматическое планирование).

В каждом узле Kubernetes как минимум работает:

* Kubelet — процесс, отвечающий за взаимодействие между мастером Kubernetes и узлом; он управляет подами и запущенными контейнерами на рабочей машине.
* Среда выполнения контейнера (например, Docker), отвечающая за получение (загрузку) образа контейнера из реестра, распаковку контейнера и запуск приложения.
* kube-proxy – сетевой прокси, который конфигурирует правила сети на узлах. При помощи них разрешаются сетевые подключения к подам изнутри и снаружи кластера.

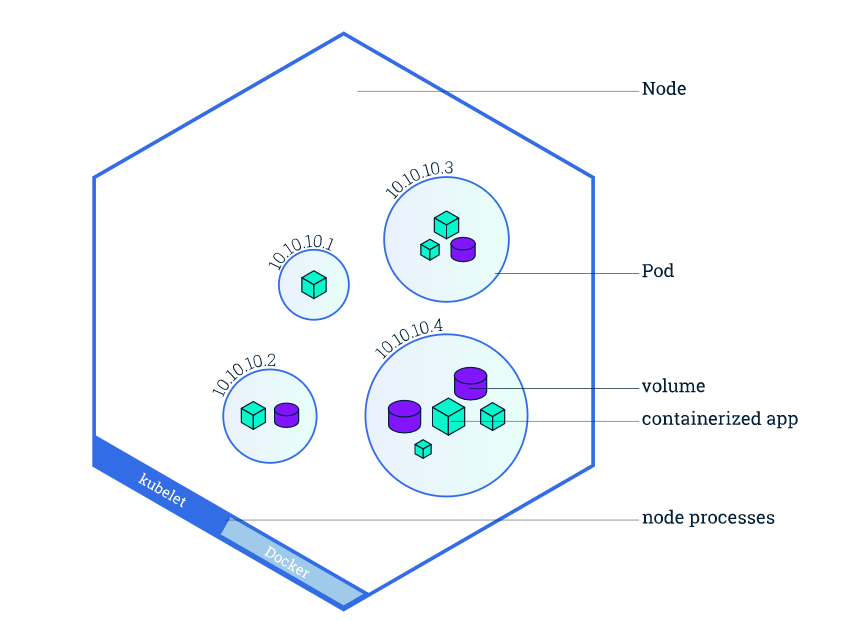


Рисунок 2 – Схема Узла

## Управление компонентами

Управление происходит посредством нескольких компонентов, объединенных под названием – Kubernetes Control Plane (компоненты панели управления).

Компоненты панели управления могут быть запущены на любой машине в кластере. Однако для простоты сценарии настройки обычно запускают все компоненты панели управления на одном компьютере (master) и в то же время не позволяют запускать пользовательские контейнеры на этом компьютере.

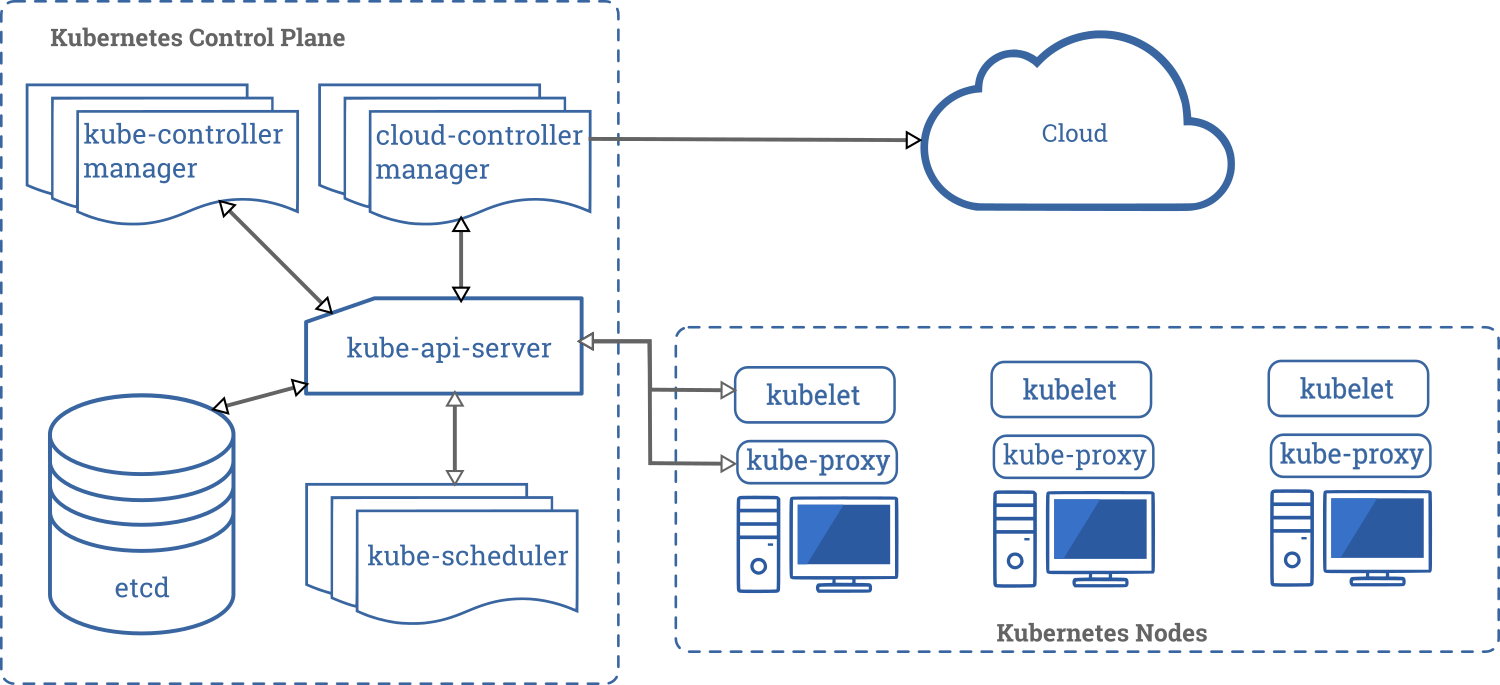


Рисунок 3 – Схема кластера K8s

### kube-apiserver

Сервер API Kubernetes проверяет и настраивает данные для объектов API, которые включают модули, службы, контроллеры репликации и другие. Сервер API обслуживает операции REST и предоставляет интерфейс для общего состояния кластера, через который взаимодействуют все остальные компоненты.

Является клиентской частью панели управления Kubernetes.

### etcd

etcd является распределённый и высоконадёжный хранилищем данных в формате "ключ-значение", которое используется как основное хранилище всех данных кластера в Kubernetes.

### kube-scheduler

Компонент плоскости управления, который отслеживает созданные поды без привязанного узла и выбирает узел, на котором они должны работать.

При планировании развёртывания подов на узлах учитываются множество факторов, включая требования к ресурсам, ограничения, связанные с аппаратными/программными политиками, принадлежности (affinity) и непринадлежности (anti-affinity) узлов/подов, местонахождения данных, предельных сроков.

### kube-controller-manager

Компонент Control Plane запускает процессы контроллера, который отслеживает общее состояние кластера через API-сервер и вносит изменения, направленные на достижения желаемого состояния.

Эти контроллеры включают:

* Контроллер узла (Node Controller): уведомляет и реагирует на сбои узла.
* Контроллер репликации (Replication Controller): поддерживает правильное количество подов для каждого объекта контроллера репликации в системе.
* Контроллер конечных точек (Endpoints Controller): заполняет объект конечных точек (Endpoints), то есть связывает сервисы (Services) и поды (Pods).
* Контроллеры учетных записей и токенов (Account & Token Controllers): создают стандартные учетные записи и токены доступа API для новых пространств имен.

### cloud-controller-manager

cloud-controller-manager запускает контроллеры, которые взаимодействуют с основными облачными провайдерами.

С помощью cloud-controller-manager код как облачных провайдеров, так и самого Kubernetes может разрабатываться независимо друг от друга.

Следующие контроллеры зависят от облачных провайдеров:

* Контроллер узла (Node Controller): проверяет облачный провайдер, чтобы определить, был ли удален узел в облаке после того, как он перестал работать.
* Контроллер маршрутов (Route Controller): настраивает маршруты в основной инфраструктуре облака.
* Контроллер сервисов (Service Controller): создаёт, обновляет и удаляет балансировщики нагрузки облачного провайдера.
* Контроллер тома (Volume Controller): создаёт, присоединяет и монтирует тома, а также взаимодействует с облачным провайдером для оркестрации томов

## Объекты Kubernetes

Объекты Kubernetes — сущности, которые хранятся в Kubernetes. Kubernetes использует их для представления состояния кластера. В частности, они описывают следующую информацию:

* Какие контейнеризированные приложения запущены (и на каких узлах).
* Доступные ресурсы для этих приложений.
* Стратегии управления приложения, которые относятся, например, к перезапуску, обновлению или отказоустойчивости.

Для работы с объектами Kubernetes – будь то создание, изменение или удаление — нужно использовать API Kubernetes, например, kubectl.

Почти у каждого объекта есть два вложенных поля-объекта, которые управляют конфигурацией объекта:

* spec, указывается требуемое состояние
* status, описывает текущее состояние, которое создается и обновляется самим K8s.

### Описание объекта (spec)

При создании объекта в Kubernetes нужно передать спецификацию объекта, которая содержит требуемое состояние, а также основную информацию об объекте (например, его имя). При работе с API или kubectl необходимо, чтобы вся информация была в JSON-формате (json, yaml).

Пример из документации:

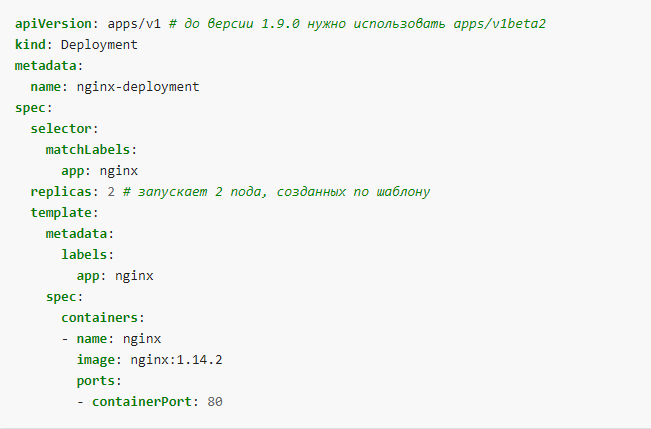


Рисунок 4 – Пример .yaml файла

Один из способов создания объекта Deployment с помощью .yaml-файла, показанного выше — использовать команду **kubectl apply**, которая принимает в качестве аргумента файл в формате .yaml.

Например: **kubectl apply -f deployment.yaml --record**

Существует несколько обязательных полей при создании объекта:

* apiVersion – используемая версия API для создания объекта
* kind – тип объекта
* metadata – данные, позволяющие идентифицировать объект (name, UID и namespace)
* spec – требуемое состояние объекта

Конкретный формат поля-объекта spec зависит от типа объекта Kubernetes и содержит вложенные поля, предназначенные только для используемого объекта, найти их можно в [справочнике API Kubernetes](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.21/).

### Основные объекты

Ниже приведены основные объекты (абстракции), используемые при работе с Kubernetes:

* Pod – минимальный элемент, в котором запускаются контейнеры;
* ReplicaSet – родительский элемент Pod’а, который отвечает за количество запускаемых подов;
* Deployment – родительский элемент ReplicaSet’а, который отвечает за обновление подов;
* ConfigMap – необходим для передачи настроек, например, добавить default.conf для nginx;
* Secret – почто то же, что и ConfigMap, но предназначены для конфиденциальных данных (логины, токены, пароли и т.д.);
* Service – распределяет трафик на поды;
* Ingress – описывает правила доступа к приложению из внешней сети (интернет);
* PV – постоянное хранилище, хранит данные, которые не теряются, когда умирает под;
* DaemonSet – запуска по одному поду на каждом узле, полезен для мониторинга (работы с агентами);
* StatefulSet – используется для запуска приложений с сохранение состояния (базы данных);
* Job – выполняется к кластере только один раз, предназначен для выполнение единоразовых задач (настройка окружения);
* CronJob – создает Job по расписанию.

### Виды управления объектами

В инструменте командной строки kubectl есть несколько разных способов создания и управления объектами Kubernetes.



Рисунок 5 – Способы управления

Императивные команды – выполняемые операции указываются в качестве аргументов или флагов:

**kubectl run nginx --image nginx**

Императивная конфигурация объекта – устанавливается действие (создание, замена и т.д.), флаги и как минимум одно имя файла с определением объекта:

**kubectl create -f nginx.yaml**

Декларативная конфигурация объекта – операции (создание, обновления, удаление) автоматически для каждого объекта определяются kubectl.

Обработать все конфигурационные файлы объектов в директории configs и создать либо частично обновить активные объекты:

**kubectl diff -f configs/**

**kubectl apply -f configs/**

### Пространство имен

Kubernetes поддерживает несколько виртуальных кластеров в одном физическом кластере. Такие виртуальные кластеры называются пространствами имён.

Используется это для разделения ресурсов между большим количеством пользователей (например, для 10 пользователей в этом нет необходимости, но для 100 это уже очень полезная функция) с помощью квоты ресурсов.

Квота ресурсов, определяемая ResourceQuota объектом, обеспечивает ограничения, которые ограничивают совокупное потребление ресурсов для каждого пространства имен, она (квота) создается и настраивается администратором.

Можно посмотреть список, существующих пространств имен:

**kubectl get namespace**

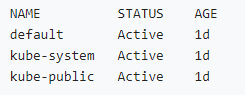


Рисунок 6 – примерный вывод команды get namespace

Пространства на рисунке являются стандартными:

* default – для объектов без какого-либо другого пространства имен
* kube-system – для объектов, созданных Kubernetes
* kube-public – доступно для чтения всем пользователям (даже для неаутентифицированных пользователей)

Для определения пространства имен для одной команды необходимо использовать флаг:

**--namespace=<namespace-name>**

Для определения пространства имен для всех последующих команд необходимо воспользоваться установкой контекста:

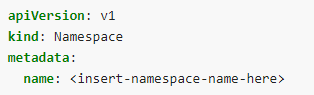
**kubectl config set-context --current --namespace=<namespace-name>**

Посмотреть какие ресурсы K8s находятся или не находятся в пространстве имен:

**kubectl api-resources --namespaced=<true or false>**

Для создание нового пространства имен есть два способа:

* Создать .yaml-файл с параметром kind: Namespace и выполнить команду **kubectl create -f ./my-namespace.yaml**:



* Ввести команду **kubectl create namespace <insert-namespace-name-here>** (для удаления заменить create на delete)

### Метки и селекторы

Метки (labels) — это пары ключ-значение, которые добавляются к объектам, например, поды. Метки предназначены для идентификации атрибутов объектов, которые имеют значимость и важны для пользователей, но при этом не относятся напрямую к основной системе:



Рисунок 7 – Пример создание объекта с метками

Метки позволяют пользователям гибко сопоставить их организационные структуры с системными объектами, не требуя от клиентов хранить эти соответствия.

В отличие от имен и идентификаторов, метки не гарантируют уникальность, поэтому с помощью меток клиент/пользователь может идентифицировать набор объектов при помощи селекторов. Селектор меток — основное средство группировки в Kubernetes.

В настоящее время API поддерживает два типа селекторов:

* на равенстве - позволяют отфильтровать объекты по ключам и значениям меток (=, !=)



Рисунок 8 – пример селекторов на равенстве

* на наборе - фильтрует ключи в соответствии с набором значений (in, notin, exists)



Рисунок 9 – пример селекторов на наборе

С помощью условия равенства в объектах Pod можно указать, какие нужно выбрать ресурсы, в данной случае выбирать для пода узлы с меткой accelerator: nvidia-tesla-p100:

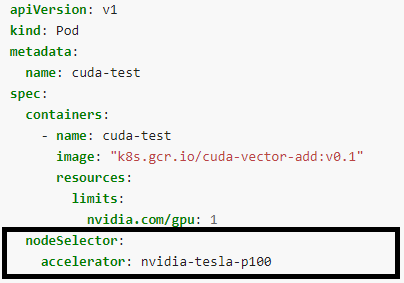


Рисунок 10 – использование меток при конфигурации пода

Селектор меток может состоять из нескольких условий, разделенных запятыми (воспринимается, как логическое И).

Примеры использования с API (kubectl):

**kubectl get pods -l environment=production,tier=frontend**

**kubectl get pods -l 'environment in (production),tier in (frontend)'**

Также селекторы могут работать не только с метками, но и с полями ресурсов, для этого необходимо использовать селектор полей (--field-selector)

Следующая команда kubectl выбирает все Pod-объекты, в которых значение поля status.phase равно Running:

**kubectl get pods --field-selector status.phase=Running**

Доступные селекторы полей зависят от типа ресурса Kubernetes. У всех типов ресурсов есть поля metadata.name и metadata.namespace

Для работы с этим типом селектором используются селекторы на равенство (=, !=)

### Аннотации

Метки можно использовать для выбора объектов и для поиска коллекций объектов, которые соответствуют определенным условиям. В отличие от них аннотации не используются для идентификации и выбора объектов, но их вид точно такой же, как и у меток – ключ: значение.

Используются они для информационных целей, например:

* информация о сборке, выпуске или образе;
* ссылки на репозитории логирования, мониторинга, аналитики или аудита;
* информация о клиентской библиотеке или инструменте, которая может использоваться при отладке;
* номера телефонов или пейджеров ответственных лиц или записи в справочнике, в которых можно найти нужную информацию, например, сайт группы;
* инструкции от конечных пользователей по исправлению работы или использования нестандартной функциональности.

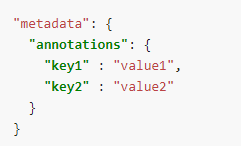


Рисунок 11 – пример аннотаций

## Безопасность и аудит событий

### Аутентификация

В Kubernetes существует два типа пользователей:

* Service Accounts – внутренние пользователи
* Users – внешние пользователи

Основное отличие их друг от друга в том, что Service Accounts существуют в Kubernetes API, которые привязаны к пространству имен и набору авторизационных данных, хранящиеся в объектах типа Secret, основная задача таких пользователей – управления правами доступа внутренних (API) процессов внутри кластера. Как можно понять users не имеют записей в Kubernetes API и предназначены для людей или процессов вне кластера.

Каждое взаимодействие с Kubernetes API привязано к одному из двух типов (Service Accounts, Users), либо считается анонимным.

Аутентификационные данные пользователя включают в себя:

* Username — имя пользователя;
* UID — машинно-читаемая строка идентификации пользователя;
* Groups — список групп, к которым принадлежит пользователь;
* Extra — дополнительные поля, которые могут быть использованы механизмом авторизации.

По умолчанию в кластере используется несколько схем авторизации:

* service account tokens – для Service Accounts;
* X509 – для Users.

Kubernetes поддерживает не только эти схемы, но и другие (Bearer-токены, аутентифицирующий прокси, HTTP Basic Auth)

### Авторизация

Аутентифицированная учетная запись по умолчанию не имеет прав на действия в кластере. Для предоставления разрешений в Kubernetes реализован механизм авторизации.

Актуальный способ разделения прав доступа к кластеру называется RBAC (Role-based access control). Он был объявлен стабильным с версии Kubernetes 1.8. RBAC реализует модель прав, в которой запрещено всё, что не разрешено явно.

Для управления доступом в Kubernetes через RBAC используются следующие сущности API:

* Role и ClusterRole — роли, которые служат для описания прав доступа:
* Role позволяет описать права в рамках пространства имён;
* ClusterRole — в рамках кластера, в том числе к кластер-специфичным объектам типа узлов, non-resources urls (т.е. не связанных с ресурсами Kubernetes — например, /version, /logs, /api\*);
* RoleBinding и ClusterRoleBinding — служит для привязки Role и ClusterRole к пользователю, группе пользователей или ServiceAccount.

Для управления Роли описывают права при помощи наборов правил, содержащих:

* группы API;
* ресурсы (resources: pod, namespace, deployment и т.п.);
* глаголы (verbs: set, update и т.п.);
* имена ресурсов (resourceNames) — для случая, когда нужно предоставить доступ к какому-то определённому ресурсу, а не ко всем ресурсам этого типа

### Аудит событий

В Kubernetes все операции проходят через api-server, что можно использовать для контроля за любой активностью в кластере. Для этого используют аудит системы, которая позволяет логировать все обращения к K8s API.

Каждый запрос на каждом этапе его выполнения генерирует событие аудита, которое затем предварительно обрабатывается в соответствии с определенной политикой и записывается в серверную часть, эти этапы:

* RequestReceived – Этап, когда обработчик аудита получает запрос, но до того, как он будет делегирован вниз по цепочке обработчиков;
* ResponseStarted- После отправки заголовков ответа, но до отправки тела ответа. Этот этап генерируется только для длительных запросов;
* ResponseComplete - Тело ответа отправлено, больше байтов отправляться не будет;
* Panic - События, возникающие при возникновении паники.

Политика аудита определяет правила о том, какие события следует регистрировать и какие данные они должны включать, когда событие обрабатывается, оно сравнивается с каждым элементом списка правил. Первое правило устанавливает уровень события:

* None – не регистрировать события, соответствующие этому правилу;
* Metadata – записывать метаданные запроса (запрашивающий пользователь, временная метка, ресурс, команда и т.д.);
* Request – записывает метаданные и тело запроса, но не тело ответа;
* RequestResponse – записывает метаданные, тело запроса и тело ответа.

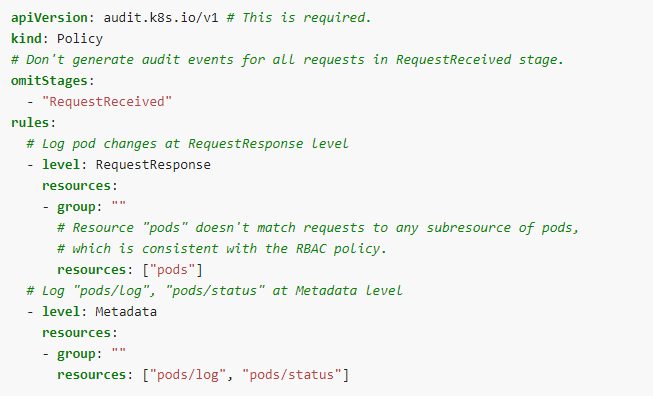


Рисунок 12 – пример файла политик

Для того, чтобы включить аудит, который будет записывать события аудита в файл, необходимо kube-apiserver передать несколько флагов:

* --audit-policy-file=<path to .yaml policy file> - указывает какую политику аудита использовать;
* --audit-log-path=<path to log file> - указывает куда отправлять данные;
* --audit-log-format=<type of logs (json)> - означает в каком формате отправлять данные.

Также необходимо, чтобы путь до всех файлов (политик и логов) был доступен для чтения процессом apiserver, из-за этого необходимо монтировать их (пути) внутри контейнера, для этого нужно добавить несколько строк в его (apiserver) конфиг, а именно смонтировать тома и настроить hostPath:

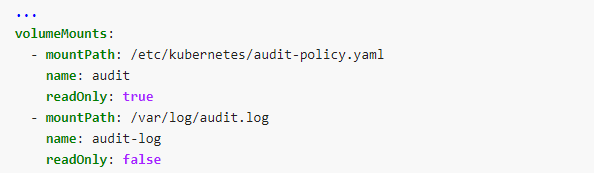


Рисунок 13 – подключение двух томов (политика и логи аудита)

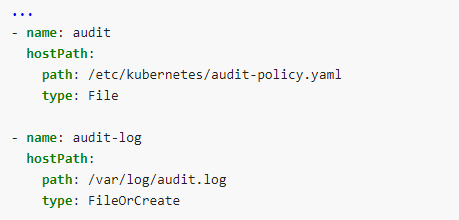


Рисунок 14 – настройка hostPath для политики и логов

Помимо трех основных флагов, имеется очень много дополнительных флагов, например, для ротации логов (--audit-log-maxbackup=10, --audit-log-maxsize=100, --audit-log-maxage=7).

Второй способ ведения записей событий аудита - организация отправки логов в удаленный веб-API, который является формой API Kubernetes. Это можно настроить, используя следующие флаги:

* --audit-webhook-config-file указывает путь к файлу с конфигурацией webhook;
* --audit-webhook-initial-backoff - указывает время ожидания после первого неудачного запроса перед повторной попыткой.

# Практика

Рабочий стенд будет представлять из себя установленный minikube. Minikube - это инструмент, позволяющий легко запускать Kubernetes на локальной машине, принцип работы сводится к установке и запуску кластера K8s, внутри которого находится всего один узел (виртуальная машина). Хотя minikube очень прост, но для обучения он подходит как нельзя лучше.

Практика будет заключаться в выполнении нескольких задач:

* подготовка стенда;
* развертывание приложения в кластере;
* обновление развернутого приложения.

Стоит отметить, что все названия с форматом .exe ведут на скачивание этих .exe файлов.

## Подготовка стенда

Для корректной работы кластера необходимо установить kubectl, minikube, Docker добавить их в Path, также установить любой гипервизор (VM, KVM)

### Установка Docker и создание образа

Docker — программное обеспечение для автоматизации развёртывания и управления приложениями в средах с поддержкой контейнеризации, контейнеризатор приложений.

Для установки необходимо установить Docker Desktop, а именно скачать [Docker Desktop Installer.exe](https://desktop.docker.com/win/stable/amd64/Docker%20Desktop%20Installer.exe?utm_source=docker&utm_medium=webreferral&utm_campaign=dd-smartbutton&utm_location=header) и следовать всем инструкциям, также на машине должен быть полностью включен Hyper-V.

Теперь, когда Docker установлен, можно переходить к созданию образа для приложения. Это приложения будет представлять их себя очень простое веб-приложение, написанное на Flask. Приложение будет состоять из двух файлов:

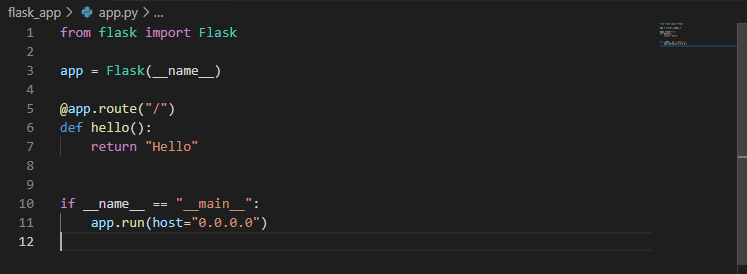
* main.py – python-файл, в котором находится весь код приложения: 

Рисунок 15 – Код веб-приложения

* requirements.txt – текстовый файл, в котором находятся необходимые модули для работы приложения:

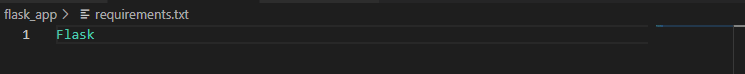


Рисунок 16 – Файл requirements.txt

Для создания образа необходимо создать файл под название Dockerfile, в котором будет находится конфигурационная информация:

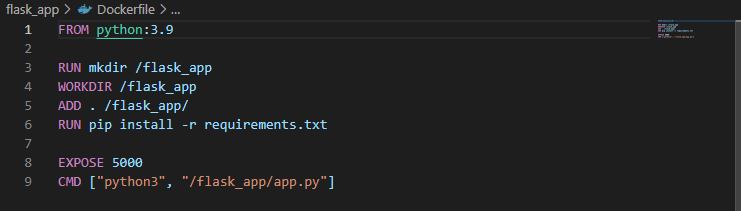


Рисунок 17 – Dockerfile

Создаем образ с помощью команды: **docker build -f Dockerfile -t hello:latest .**

Проверяем успешность создание образа и запускаем контейнер:



Рисунок 18 – Проверка наличия контейнера

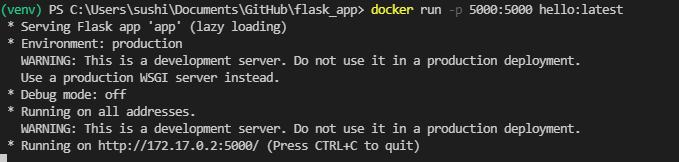


Рисунок 19 – Запуск контейнера

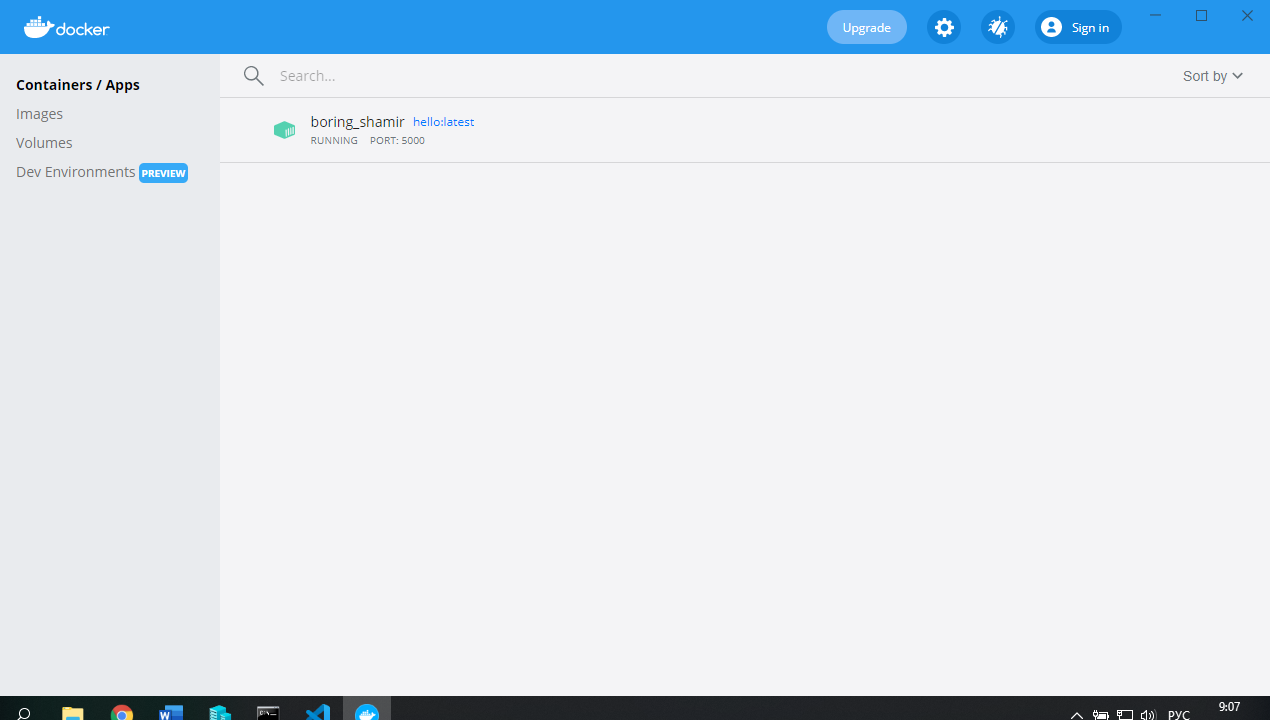


Рисунок 20 – Контейнер в Docker Desktop

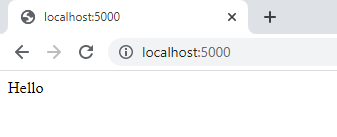


Рисунок 21 – Проверка работоспособности контейнера

### Установка minikube и добавление в Path

Шаги для подготовки стенда:

1. Скачать minikube ([minikube-installer.exe](https://github.com/kubernetes/minikube/releases/latest/download/minikube-installer.exe)).
2. Запустить .exe файл и установить minikube.
3. Скачать kubectl ([kubectl.exe](https://storage.googleapis.com/kubernetes-release/release/v1.21.0/bin/windows/amd64/kubectl.exe))
4. Закинуть kubectl в папку с minikube, для более удобного добавления в Path
5. Зайти в свойства компьютера\дополнительные настройки\переменные среды, найти Path и добавить путь до kubectl, для minikube он ставится автоматически:

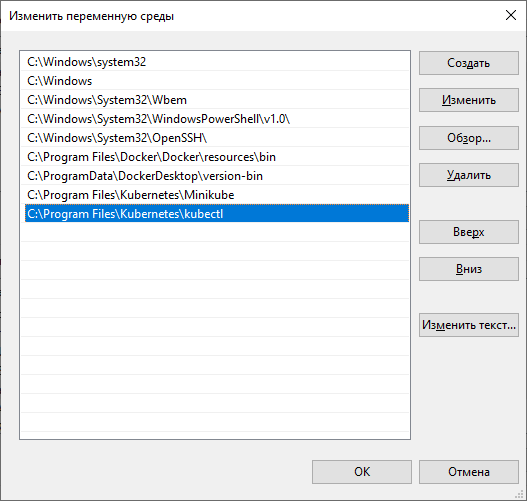


Рисунок 22 - Path

### Запуск и проверка кластера

Для проверки необходимо зайти в cmd от имени Администратора и выполнить несколько команд:

1. Запустить Диспетчер Hyper-V
2. minikube start --vm-driver=hyperv– запустить кластер при помощи hyperv, при первом запуске необходимо подождать его установку
3. После запуска в интерфейсе Hyper-V должна появиться виртуальная машина minikube:

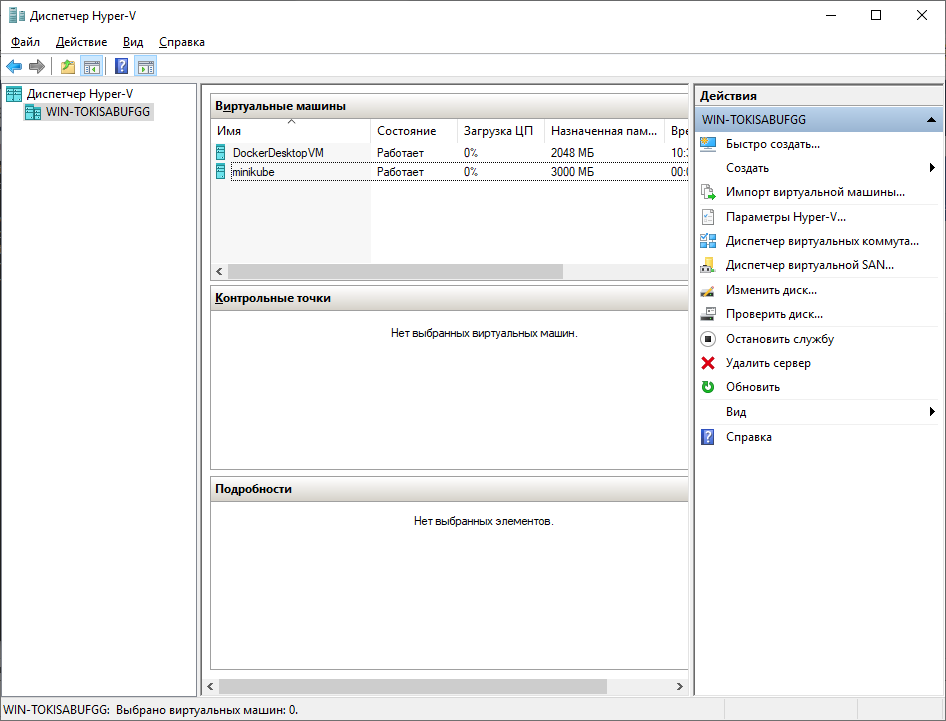


Рисунок 23 – Диспетчер Hyper-V с minikube

1. Ввести в терминал команду minikube status для проверки состояния кластера, если кластер запустился правильно, то будет вывод будет иметь такой вид:

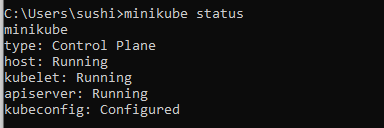


Рисунок 24 – Вывод при правильной работе кластера

## Развертывание приложения в Kubernetes

### Перенос образа на minikube

Для того, чтобы docker внутри minikube смог использовать созданный образ, его (образ) необходимо перенести в локальный реестр minikube:

1. Сохраняем образ как архив (.tar):



Рисунок 25 – Сохранение образа

1. Переносим архив в minikube с помощью scp:

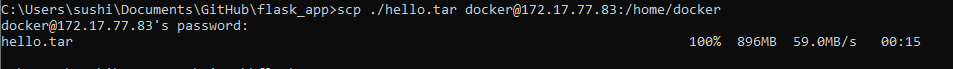


Рисунок 26 – Перенос архива

1. Заходим в minikube и загружаем архив в локальный реестр:

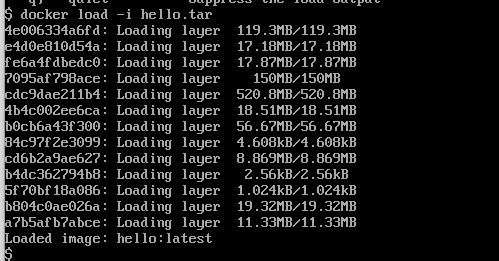


Рисунок 27 – Загрузка в локальный реестр

Теперь нашему кластеру доступен образ, созданный ранее, проверим, используя команду docker image ls, теперь помимо K8s образов на машине присутствует hello:latest образ:

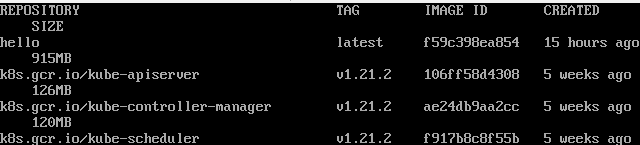


Рисунок 28 – Проверка наличия образа

### Конфигурирование и создание подов

Далее необходимо создать .yaml-файл с конфигурацией нашего приложения:

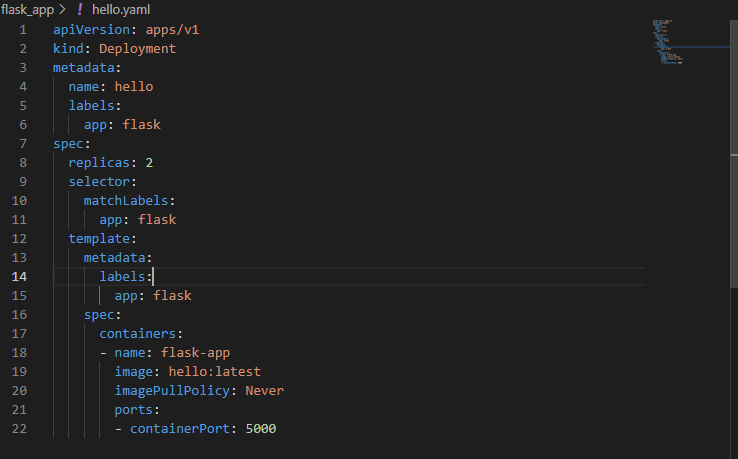


Рисунок 29 – Конфигурационный .yaml-файл

Примерное значения этого файла для kubectl – создать объект типа Deployment, используя apps/v1, с именем hello, в количестве 2 штук, используя локальный образ hello:latest (imagePullPolicy: Never запрещает пользоваться внешними источниками образов, например, Docker hub), также открыть 5000 порт для внешний соединений.

Запускаем и проверяем работоспособность подов:



Рисунок 30 – Запуск подов, используя конфигурационный файл

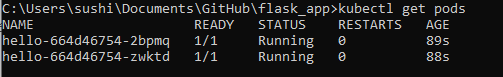


Рисунок 31 – Проверка подов

Как видно из рисунка выше, работают два пода с именем hello. Но для того, что мы могли протестировать работу веб-приложения необходимо создать ещё один под, а именно тип Service, который предоставляет доступ к поду для внешних источников:



Рисунок 32 – Создание Service пода

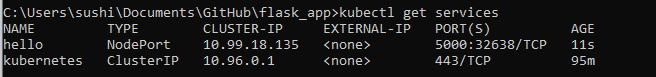


Рисунок 33 – Проверка работоспособности Service

Теперь необходимо узнать url приложения и проверить его работоспособность:



Рисунок 34 – Получения url

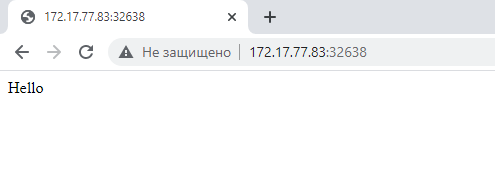


Рисунок 35 – Проверка работоспособности пода

## Обновление приложения

Для обновления созданного Deployment’а hello существует два правильных способа:

* Изменить конфигурационный файл .yaml;
* Использовать set image.

### Создание нового образа

Для демонстрации обновления необходимо создать новый образ нашего веб-приложения, теперь вместо “Hello” оно будет выводить “Privet”, затем создаем образ и загружаем его в локальный реестр minikube:

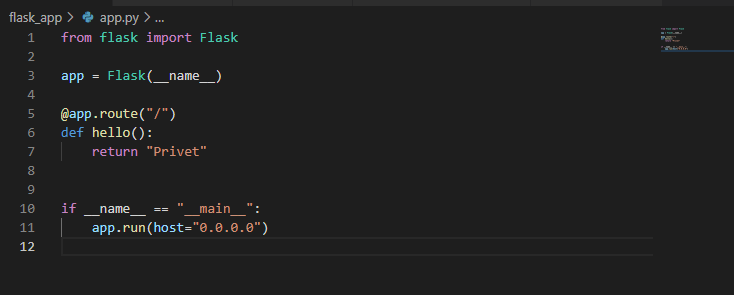


Рисунок 36 – Код, обновленного приложения

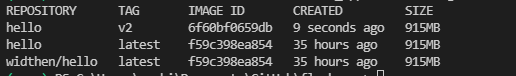


Рисунок 37 – Список с новым образом

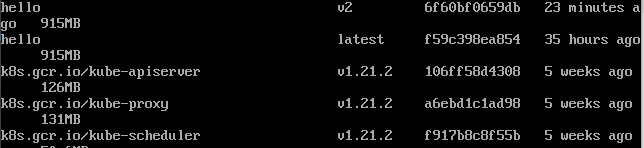


Рисунок 38 – Список с новым образом на minikube

### Изменение конфигурационного файла

Пошагово обновим приложение с помощью изменения файла:

1. Изменяем файл hello.yaml:

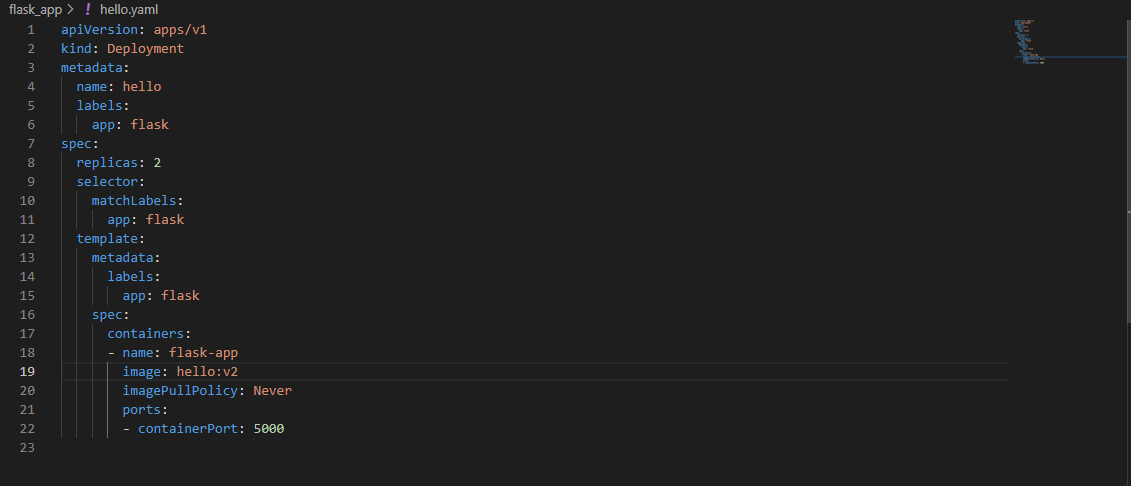


Рисунок 39 – Обновленный конфигурационный файл

1. Завершаем текущие поды:

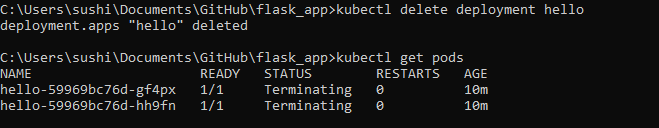


Рисунок 40 – Завершение подов

1. Запускаем новые поды с новым конфигурационным файлом:

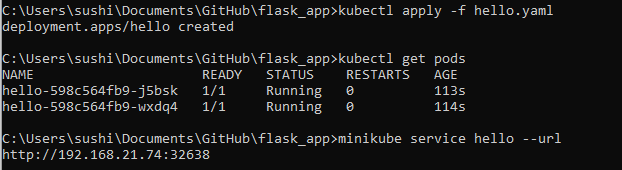


Рисунок 41 – Запуск новых подов

1. Проверяем, переходя по полученному адресу:

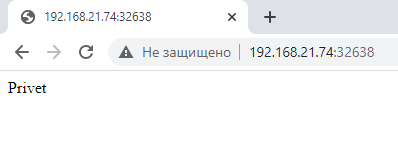


Рисунок 42 – Проверка обновления приложения

Оценка способа, плюсы:

* Можно изменять все параметры объекта.

Минусы:

* Существует временное окно, когда приложении не доступно;
* Только внешняя система управления версиями (git).

### Использование set image

Пошагово обновим приложение с помощью set image:

1. Приведем hello.yaml-файл в первоначальный вид и запустим поды на его основе:



Рисунок 43 – Первоначальный вид hello.yaml

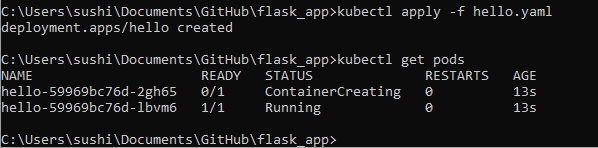


Рисунок 44 – Создание подов на основе первоначального файла

1. С помощью команды set image меняет образ для текущих подов:

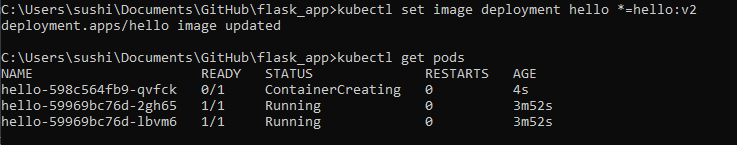


Рисунок 45 – Обновление с помощью set image

1. Проверим обновление:

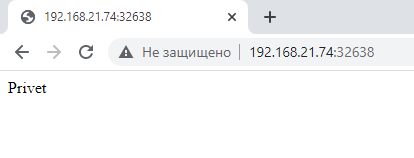


Рисунок 46 – Проверка обновления

Как можно заметить изменились значения ReplicaSet и Pod, потому что создались новые объекты взамен старых, но главная особенность такого метода — это плавающее обновления, поды обновляются не все сразу, а по одному, то есть пока один под обновляется другой работает, что обеспечивает бесперебойность работы приложения.

Также если посмотреть список ReplicaSet’ов, то можно увидеть неизвестный ReplicaSet с 0 подов под его контролем:

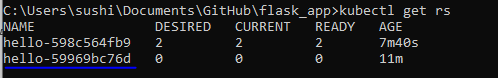


Рисунок 47 – Вызов списка ReplicaSet’ов

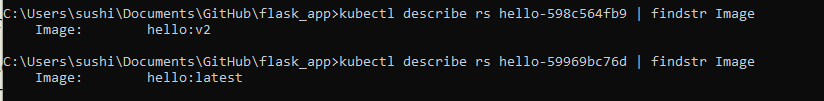


Рисунок 48 – Сравнение образов двух ReplicaSet’ов

Как видно из рисунка выше этот ReplicaSet используем прошлый образ и с его (ReplicaSet) помощью можно откатить обновление (по умолчанию может быть до 10 версий), стоит обратить внимание на значение ReplicaSet:

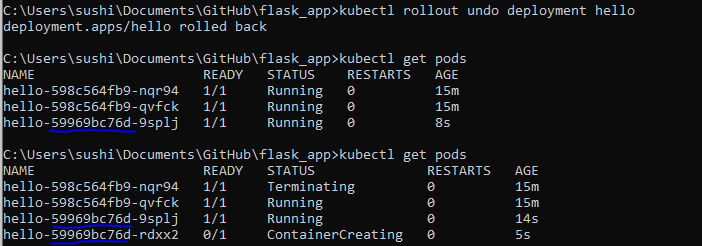


Рисунок 49 – Откат обновления

Точно такое значение, что и у нулевого, проверим, что будет при переходе на сайт:

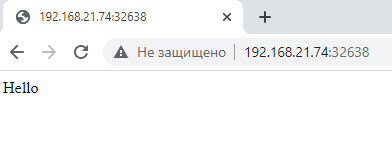


Рисунок 50 – Проверка отката обновления

Теперь выводится старая надпись, что говорит об успешном откате изменения.

Оценка способа, плюсы:

* Встроенный контроль версий;
* Бесперебойная работа приложения.

Минусы:

* Можно менять только образ, для других изменений используются другие команды, например, для масштабирования – **kubectl scale**