



# Kubernetes. Введение

Докладчик: Илья Крылов

Дата: май 2020

# Что такое Kubernetes?

Kubernetes (далее k8s) - open-source проект для управления кластером контейнеров Linux как единой системой.

Kubernetes управляет и запускает контейнеры Docker на большом количестве хостов, а также обеспечивает совместное размещение и репликацию большого количества контейнеров.

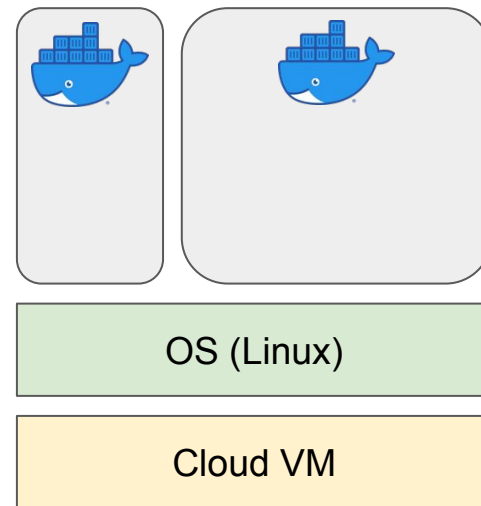
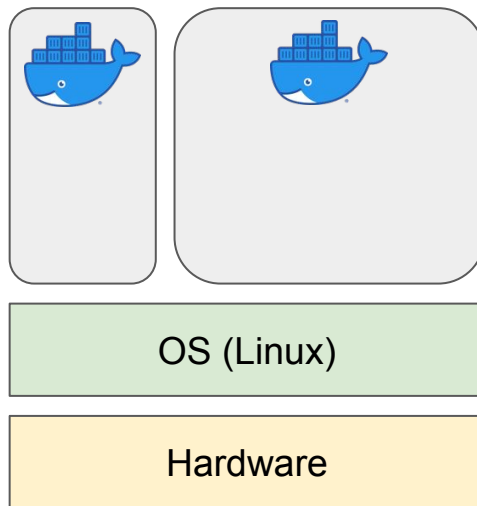
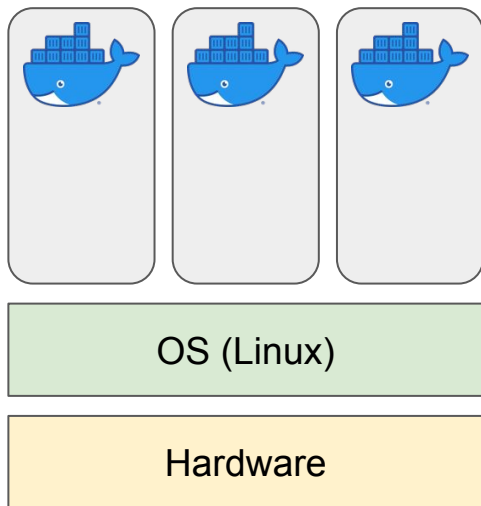
Проект был начат Google и теперь поддерживается многими компаниями, среди которых Microsoft, RedHat, IBM и Docker.

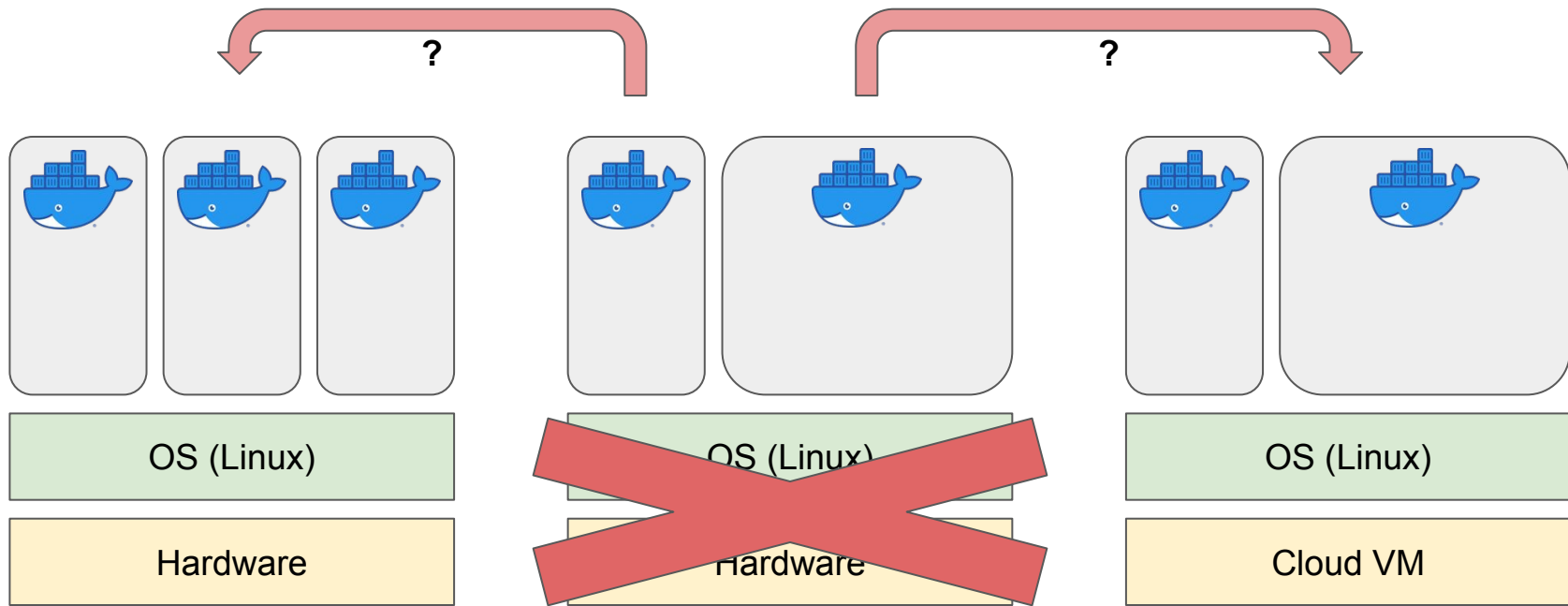
# Какую проблему решает k8s?

Две основные проблемы:

1. Масштабирование и запуск контейнеров на большом кол-ве хостов.
2. Балансировка контейнеров между хостами.

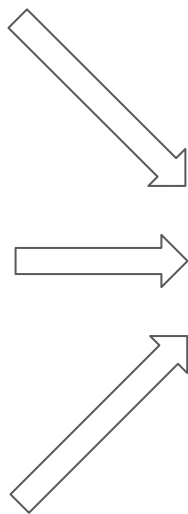
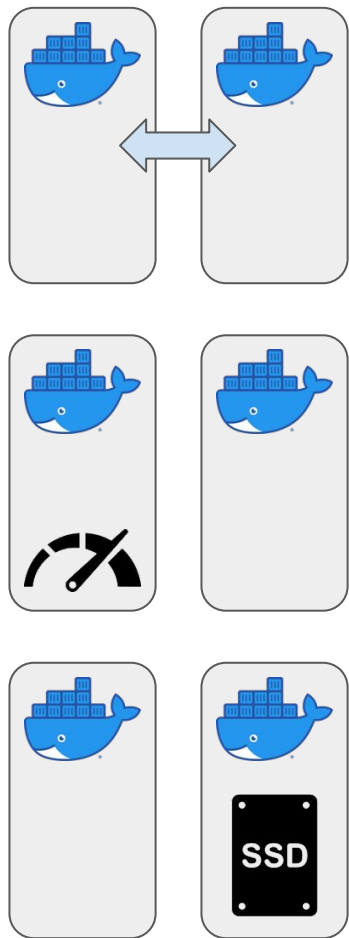
K8s предлагает высокоуровневый API для логического группирования контейнеров, их размещения, а также их последующего балансирования.



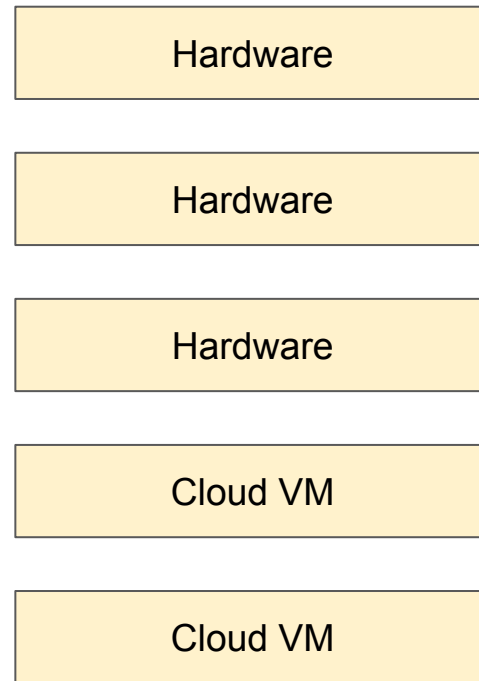
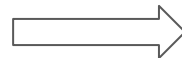


Потенциальные проблемы:

1. Контейнеры могут быть связаны и должны быть расположены на одной ноды.
2. Контейнеры могут “не убраться” на оставшиеся ноды без дополнительных перестановок.
3. При вводе в строй новой ноды придется снова выполнять “переезд” контейнеров в обратном порядке.



Требования



- Сложные правила расположения контейнеров
- Self-healing
- Infrastructure as code (IaC)

# Задачи, которые решает k8s

- Автоматизация инфраструктуры (развертывания и откаты)
- (Авто) масштабирование
- Supervision
- Service Discovery
- Логирование
- Сбор метрик (мониторинг)
- CI / CD
- Уменьшение vendor lock-in (изоляция от платформы)

# Minikube - локальный кластер kubernetes

<https://minikube.sigs.k8s.io/>



**minikube**

Локальный кластер k8s для Linux/Windows/MacOS.

Используется для знакомства и локальных экспериментов с Kubernetes.

Процедура установки предельно проста: <https://minikube.sigs.k8s.io/docs/start/>

После установки останется только запустить кластер:

```
$ minikube start
```

```
$ minikube stop
```



# Kubectl - клиент для управления кластером

Необходимо установить на всех машинах, с которых планируется управлять кластером.

Инструкция по установке: <https://kubernetes.io/ru/docs/tasks/tools/install-kubectl/>

Конфигурационный файл для kubectl:

1. `$HOME/.kube/config` - расположение по умолчанию
2. `KUBECONFIG` - переменная окружения
3. `--kubeconfig` - параметр запуска

# Node - узел кластера

Физическая или виртуальная машина, которая является частью k8s-кластера.

Может играть разные роли:

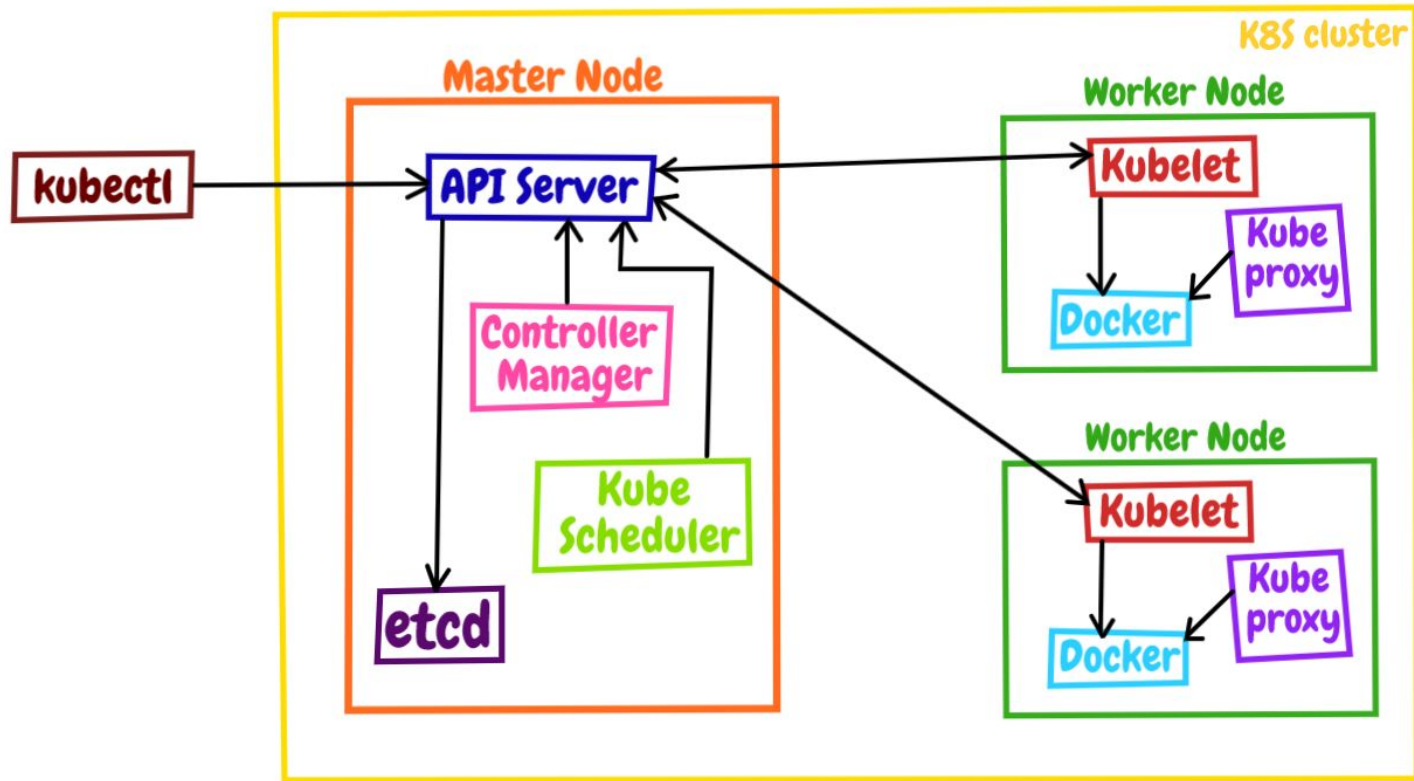
- **Master** - предназначена для размещения управляющих и координирующих элементов кластера (etcd, API server, планировщик и различные менеджеры).
- **Worker** - предназначена для рабочей нагрузки (размещение и выполнение приложений в pod'ax).

Обычно эти роли распределены по разным машинам, но в некоторых случаях могут совмещаться одной и той же машиной (например, в dev-окружениях).

```
$ kubectl get nodes - просмотр списка узлов
```

```
$ kubectl describe node minikube - просмотр данных об одном узле
```

# Архитектура Kubernetes-кластера



# Компоненты master-узлов

## API Server

Фронтенд управляющего контура, с которым взаимодействуют все прочие компоненты. Он принимает и обрабатывает запросы по REST API.

## Controller Manager

Общее название для набора служб, следящих за состоянием кластера и вносящих необходимые изменения для приведения фактического состояния кластера к желаемому.

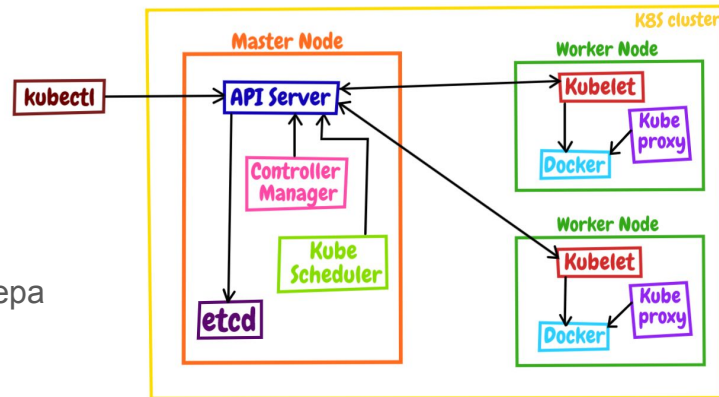
Важно: контроллеры не выполняют фактическую работу, они занимаются лишь конфигурированием.

## Kube Scheduler

Компонент, отвечающий за распределение рабочей нагрузки по доступным узлам. Распределение выполняется как на основании заданных настроек, так и на основании текущего использования ресурсов.

## etcd

Простое распределенное хранилище типа “ключ-значение”. Хранит в себе информацию о кластере и его состоянии.



# Компоненты worker-узлов

## Kubelet

Постоянно опрашивает **API server** на предмет того, какие pod'ы назначены для данного узла и запускает/удаляет их путем взаимодействия с **container runtime engine**.

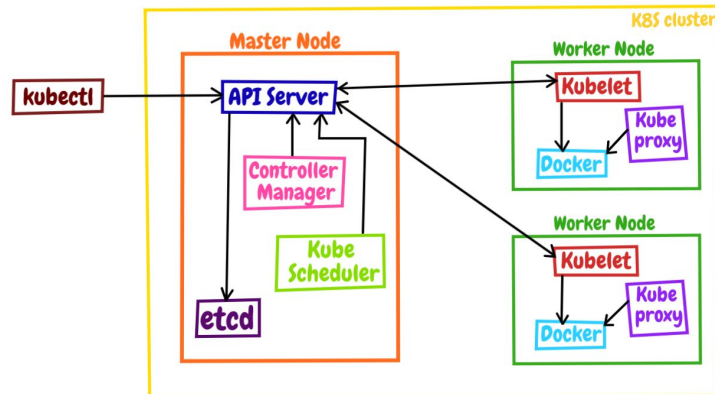
Информирует **API server** о статусе работающих на узле pod'ов.

## Kube Proxy

Компонент, управляющий сетевыми настройками узла (настройка маршрутизации).

## Container Runtime

Компонент для работы с контейнерами (Docker, Containerd, Cri-o, ...).



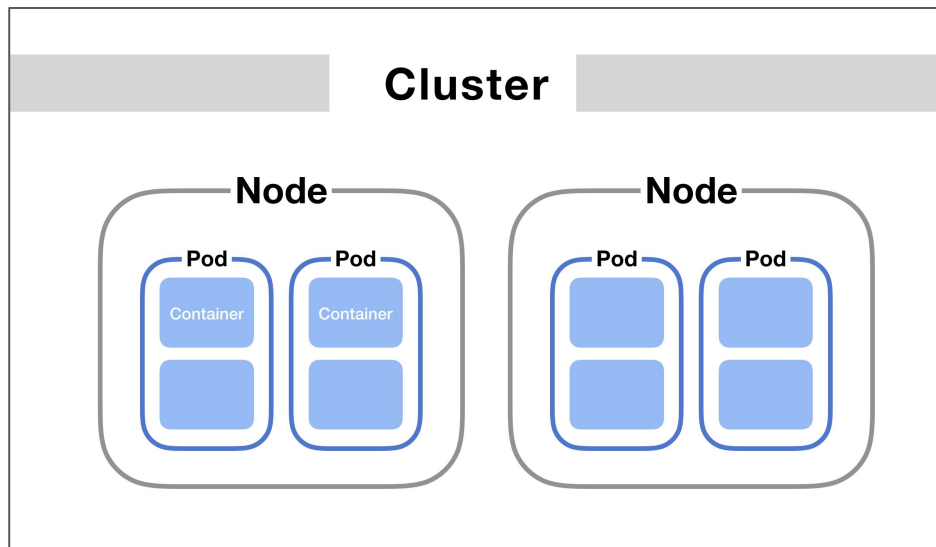
# Pod - группа контейнеров

**Pod** (англ. “стручок”) - запрос на запуск одного или более контейнеров на одном узле. Эти контейнеры разделяют доступ к таким ресурсам, как тома хранилища и сетевой стек (каждый pod имеет свой собственный внутренний IP).

Также pod’ом можно назвать и совокупность контейнеров, которые запускаются в ответ на этот запрос.

Контейнер можно запустить только внутри pod’a.

Pod’ы - базовые строительные блоки, на основе которых строятся другие сущности k8s.



# Pod: пример

```
apiVersion: v1          ← версия API
kind: Pod                ← тип сущности
metadata:                ← метаданные
  name: my-pod
  namespace: default    ← пространство имен
  labels:
    name: nginx
spec:                    ← спецификация
  containers:
    - name: nginx
      image: nginx
      ports:
        - containerPort: 80
```

# Pod: запуск

```
$ kubectl apply -f 01_pod.yml  
pod/my-pod created
```

```
$ kubectl get pods -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
my-pod	1/1	Running	0	30s	172.17.0.4	minikube

```
$ kubectl describe pod my-pod
```

```
$ kubectl delete pod my-pod
```

```
pod "my-pod" deleted
```



# Версии API

Kubernetes поддерживает несколько версий API, например:

- v1
- v2beta4
- v3alpha1
- extensions/v1beta1

**Alpha-версии API** (выключены по умолчанию): могут содержать баги, поддержка может быть прекращена в любое время, совместимость с будущими версиями не гарантируется. Не рекомендуется использовать в production.

**Beta-версии API** (включены по умолчанию): хорошо протестированы и поддержка не может быть прекращена, но возможны незначительные изменения в семантике. При нарушении совместимости с будущими версиями предоставляются инструкции по миграции. Не рекомендуется использовать в production в общем случае.

**Стабильные версии API** - production-ready, совместимость с будущими версиями.

# Namespaces - пространства имен

1. Имена ресурсов должны быть уникальными в пределах одного и того же пространства имён.
2. Каждый ресурс Kubernetes может находиться только в одном пространстве имён.
3. Пространства имён не могут быть вложенными.

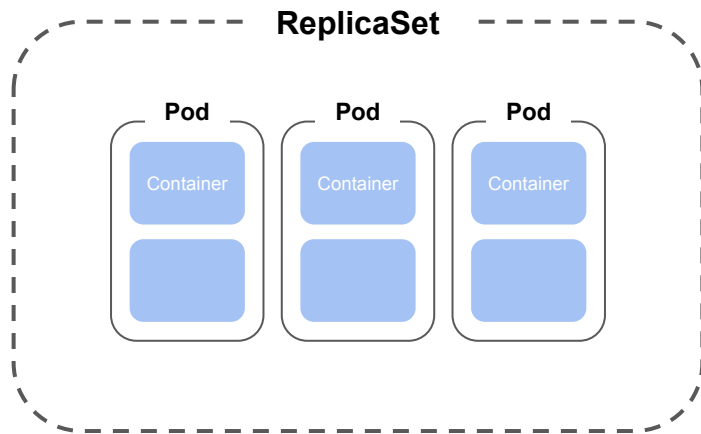
Пространства имён — это способ разделения ресурсов кластера между несколькими пользователями/проектами. По умолчанию в Kubernetes определены три пространства имён:

- **default** — пространство имён по умолчанию для объектов без какого-либо другого пространства имён.
- **kube-system** — пространство имён для объектов, созданных Kubernetes (системных).
- **kube-public** — создаваемое автоматически пространство имён, которое доступно для чтения всем пользователям (включая также неаутентифицированных пользователей).

`$ kubectl get namespaces` - получение списка namespace'ов

# ReplicaSet - группа однотипных pod'ов (реплик)

Запускает определенное количество однотипных pod'ов и гарантирует поддержание данного количества. Pod'ы могут быть запущены на разных узлах кластера.



# ReplicaSet: пример

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: my-replicaset
spec:
  replicas: 2           ← кол-во контейнеров
  selector:
    matchLabels:
      app: my-app      ← селектор для поиска "своих" pod'ов
  template:             ← шаблон pod'а
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - image: nginx:1.12
          name: nginx
          ports:
            - containerPort: 80
```

# ReplicaSet: запуск

```
$ kubectl apply -f 02_replicaset.yml
replicaset.apps/my-replicaset created
```

```
$ kubectl get pods -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
my-replicaset-fqcsr	1/1	Running	0	33s	172.17.0.5	minikube
my-replicaset-pwp5m	1/1	Running	0	33s	172.17.0.4	minikube

```
$ kubectl get replicaset -o wide
```

NAME	DESIRED	CURRENT	READY	AGE	CONTAINERS	IMAGES	SELECTOR
my-replicaset	2	2	2	35s	nginx	nginx:1.12	app=my-app

Пробуем удалить один из pod'ов и убеждаемся, что ReplicaSet автоматически восстанавливает заданное кол-во.

```
$ kubectl delete pod my-replicaset-pwp5m
pod "my-replicaset-pwp5m" deleted
```

# ReplicaSet: обновление версии образа

Пробуем обновить версию `image` в описании `ReplicaSet` и заново применить изменения. Для этого меняем значение параметра `"image"` на `"nginx:1.13"` и делаем `"apply"`.

```
$ kubectl apply -f 02_replicaset.yml  
replicaset.apps/my-replicaset configured
```

Видим, что `pod`'ы не пересоздались и они используют старую версию образа.

```
$ kubectl describe pod my-replicaset-fqcsr
```

Меняем значение параметра `"replicas"` на 5 и еще раз делаем `"apply"`.

```
$ kubectl apply -f 02_replicaset.yml  
replicaset.apps/my-replicaset configured
```

Это изменение также можно сделать через `"kubectl scale --replicas=5 my-replicaset"`, но это может нарушить принцип IaC.

```
$ kubectl describe replicaset my-replicaset
```

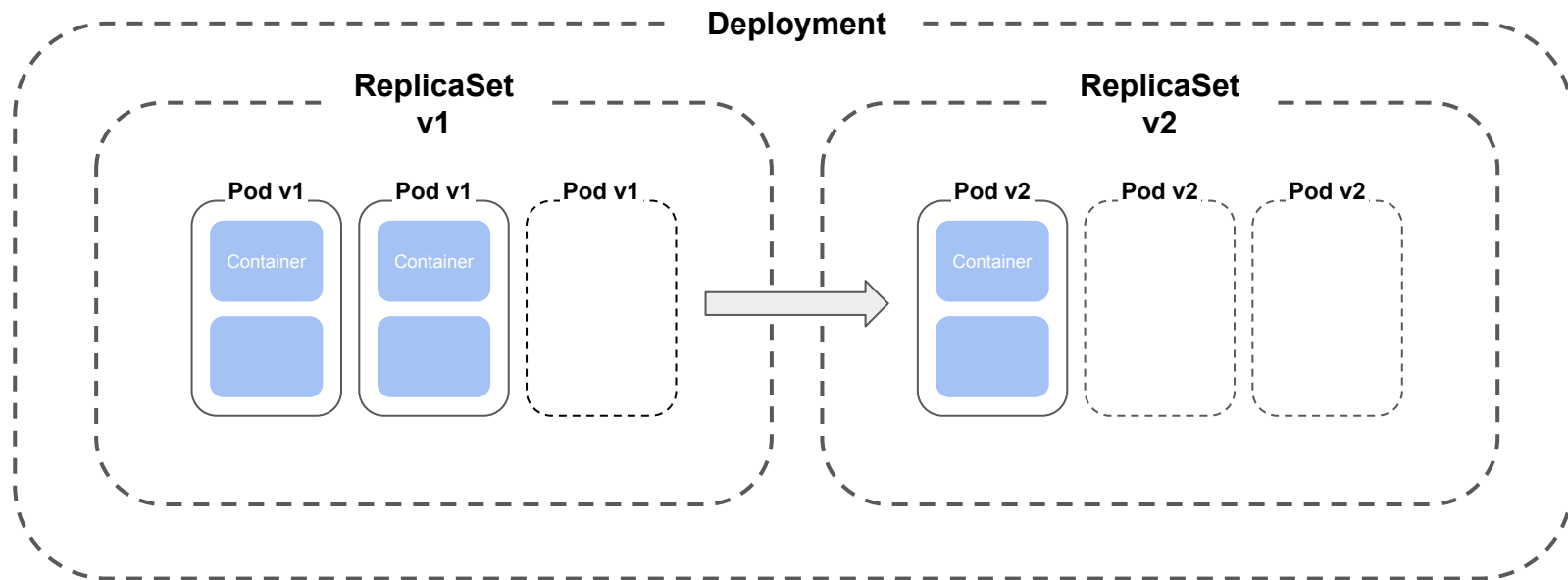
Видим, что новая версия образа используется только для новых `pod`'ов.

То есть `ReplicaSet` не имеет механизмов для обновления уже запущенных `pod`'ов.

```
$ kubectl delete replicaset my-replicaset  
replicaset.apps "my-replicaset" deleted
```

# Deployment - развертывание

Обеспечивает механизм обновления pod'ов с помощью ReplicaSet'ов.



# Deployment: пример

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment
spec:
  replicas: 2                ← кол-во контейнеров
  strategy:
    type: Recreate           ← стратегия обновления
  selector:                  ← селектор для поиска "своих" pod'ов
    matchLabels:
      name: my-app
  template:                  ← шаблон pod'а
    metadata:
      labels:
        name: my-app
    spec:
      containers:
        - name: nginx
          image: nginx:1.12
          ports:
            - containerPort: 80
```



# Deployment: запуск

```
$ kubectl apply -f 03_deployment.yml  
deployment.apps/my-deployment created
```

```
$ kubectl get deployments -o wide
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE	CONTAINERS	IMAGES	SELECTOR
my-deployment	2/2	2	2	12s	nginx	nginx:1.12	name=my-app

```
$ kubectl get pods -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
my-deployment-6f9b6dd5dc-2tw4d	1/1	Running	0	15s	172.17.0.5	minikube
my-deployment-6f9b6dd5dc-mh69k	1/1	Running	0	15s	172.17.0.4	minikube

```
$ kubectl describe deployment my-deployment
```

```
$ kubectl get replicaset
```

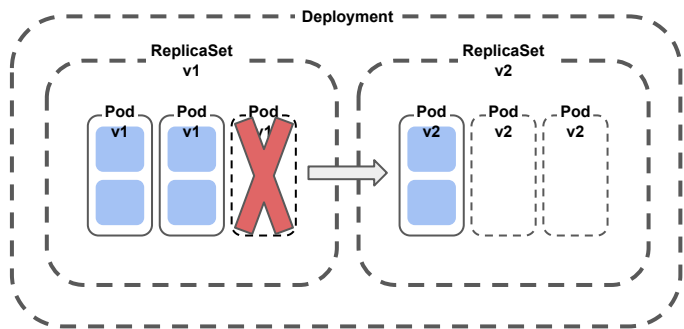
Теперь можно выполнить запрос "curl <IP-адрес Pod'a>" с любой машины кластера (не извне!)

```
$ kubectl run --rm -it --image amouat/network-utils test bash
```

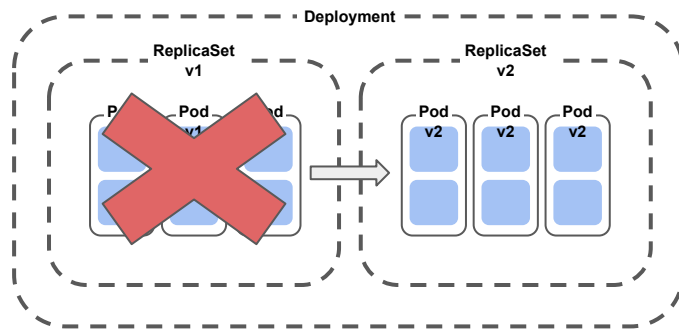
```
# curl 172.17.0.5
```

# Deployment: стратегии обновления

```
spec:
  strategy:
    type: RollingUpdate    ← тип по умолчанию
    rollingUpdate:
      maxSurge: 25%          ← ЛИМИТ превышения
      maxUnavailable: 25%   ← ЛИМИТ недоступности
```



```
spec:
  strategy:
    type: Recreate
```



ReplicaSet'ы остаются пустыми после обновления. Это история изменений данного deployment'а. В параметре "**revisionHistoryLimit**" можно задать, сколько replicaset'ов должен хранить данный deployment (по умолчанию равно 10).

# Deployment: обновление версии образа

Пробуем обновить версию image в описании Deployment и заново применить изменения. Для этого меняем значение параметра "image" на "nginx:1.13" и делаем "apply".

```
$ kubectl apply -f 03_deployment.yml  
deployment.apps/my-deployment configured
```

Видим, что старые pod'ы удалились и были запущены новые, с новой версией образа. Новые pod'ы принадлежат новому replicaset'у.

```
$ kubectl get replicaset
```

NAME	DESIRED	CURRENT	READY	AGE
my-deployment-6f9b6dd5dc	0	0	0	3m15s
my-deployment-746f5c5d87	2	2	2	20s

Удаляем deployment (это удалит также и связанные с ним replicaset'ы):

```
$ kubectl delete -f 03_deployment.yml
```

# Deployment: откат изменений

Запускаем новый deployment с фиксацией изменений:

```
$ kubectl apply -f 03_deployment.yml --record=true
deployment.apps/my-deployment created
```

Пробуем обновить версию образа (заведомо с ошибкой):

```
$ kubectl set image deployment.apps/my-deployment nginx=nginx:1.131 --record=true
deployment.apps/my-deployment image updated
```

Видим, что deployment не может развернуться:

```
$ kubectl rollout status deployment.apps/my-deployment
Waiting for deployment "my-deployment" rollout to finish: 0 of 2 updated replicas are available...
```

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
my-deployment-658c68868d-d245d	0/1	ImagePullBackOff	0	27s
my-deployment-658c68868d-k2bpg	0/1	ImagePullBackOff	0	27s

Смотрим историю изменений данного deployment'a:

```
$ kubectl rollout history deployment.apps/my-deployment
deployment.apps/my-deployment
REVISION  CHANGE-CAUSE
1          kubectl apply --filename=03_deployment.yml --record=true
2          kubectl set image deployment.apps/my-deployment nginx=nginx:1.131 --record=true
```

# Deployment: откат изменений (продолжение)

Можем подробнее посмотреть описание каждой ревизии:

```
$ kubectl rollout history deployment.apps/my-deployment --revision=2
```

Откатываемся на предыдущую ревизию (также есть параметр "--to-revision"):

```
$ kubectl rollout undo deployment.apps/my-deployment  
deployment.apps/my-deployment rolled back
```

Проверяем, что откат прошел успешно:

```
$ kubectl rollout status deployment.apps/my-deployment  
deployment "my-deployment" successfully rolled out
```

Проверяем версию образа:

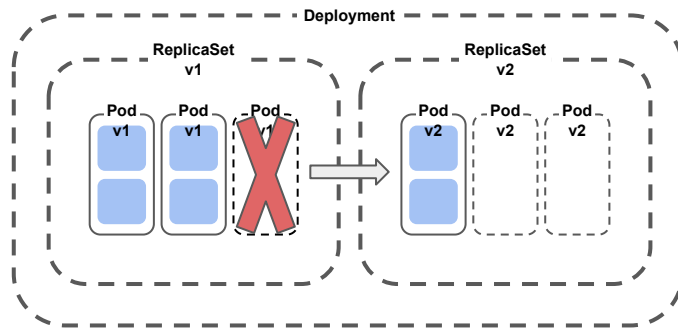
```
$ kubectl describe deployment my-deployment | grep Image  
Image:          nginx:1.12
```

Удаляем deployment:

```
$ kubectl delete -f 03_deployment.yml  
deployment.apps "my-deployment" deleted
```

# Service - сетевой доступ к приложению

Тип ресурса Kubernetes, который заставляет прокси настраиваться для пересылки запросов на набор контейнеров.



IP-адреса pod'ов постоянно меняются.

Service - это абстракция для предоставления сетевого доступа к приложению, работающему на группе pod'ов, а также для балансировки запросов к этим pod'ам.

# Service: пример

```
apiVersion: v1
```

```
kind: Service
```

```
metadata:
```

```
  name: my-service
```

```
spec:
```

```
  type: ClusterIP
```

← тип сервиса

```
  ports:
```

```
    - port: 80
```

← входящий порт сервиса

```
      targetPort: 80
```

← целевой порт у pod'ов сервиса

```
  selector:
```

← селектор для определения целевых pod'ов

```
    app: my-app
```

# Service: запуск

```
$ kubectl apply -f 04_deployment_for_service.yml -f 05_service_cluster_ip.yml
```

```
configmap/my-configmap created
```

```
deployment.apps/my-deployment created
```

```
service/my-service-cluster-ip created
```

```
$ kubectl get services
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
my-service-cluster-ip	ClusterIP	10.108.193.101	<none>	80/TCP	114s

Сервис с указанным селектором автоматически создает еще один тип сущности - endpoints.

```
$ kubectl get endpoints
```

NAME	ENDPOINTS	AGE
my-service-cluster-ip	172.17.0.4:80,172.17.0.5:80	130s

Теперь можно выполнить запрос "curl <IP сервиса>" с любой машины кластера (не извне!)

```
$ kubectl run --rm -it --image amouat/network-utils test bash
```

```
# curl 10.108.193.101
```

```
Hello from my-deployment-784598767cdz87v
```

```
# curl 10.108.193.101
```

```
Hello from my-deployment-784598767ces77b
```

```
# curl my-service-cluster-ip
```

```
Hello from my-deployment-784598767ces77b
```

Не удаляем сервис и deployment, они нужны для дальнейших экспериментов.



# Container probes - проверки доступности

```
containers:
  - image: nginx:1.12
    name: nginx
    ports:
      - containerPort: 80
    readinessProbe:      ← проверка готовности контейнера
      httpGet:
        path: /
        port: 80
      periodSeconds: 2
      failureThreshold: 3
      successThreshold: 1
      timeoutSeconds: 1
    livenessProbe:       ← проверка работоспособности контейнера
      httpGet:
        path: /
        port: 80
      periodSeconds: 10
      failureThreshold: 3
      successThreshold: 1
      timeoutSeconds: 1
      initialDelaySeconds: 10
```

# Service: типы сервисов

- Без селектора
- ClusterIP
- NodePort
- LoadBalancer
- ...

# Service: сервисы без селектора

У сервиса можно не заполнять поле “selector”, в этом случае endpoints придется создать отдельно. Это позволяет, например, создать сервис для некой службы извне кластера.

```
apiVersion: v1
kind: Service
metadata:
  name: my-postgresql
spec:
  ports:
    - protocol: TCP
      port: 5432
      targetPort: 5432
```

```
apiVersion: v1
kind: Endpoints
metadata:
  name: my-postgresql
subsets:
  - addresses:
      - ip: 192.0.2.42
    ports:
      - port: 5432
```

# Service: тип сервиса ClusterIP

Тип "ClusterIP" используется по умолчанию.

Сервису выделяется отдельный IP внутри кластера.

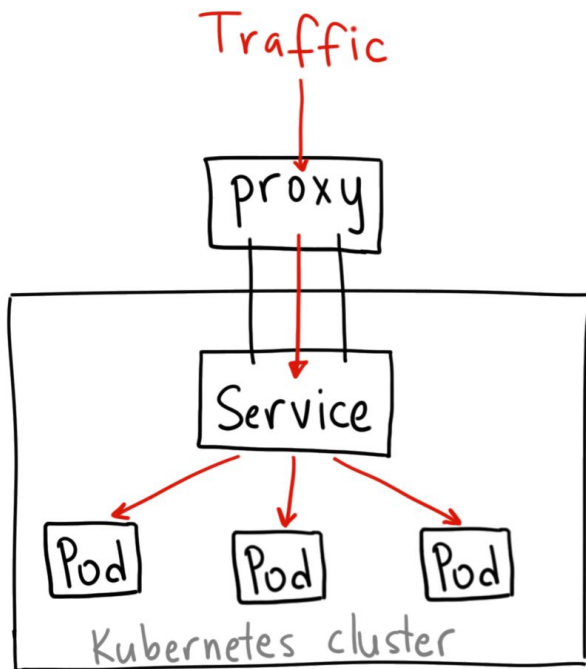
Так как IP внутренний, то доступа к сервису снаружи кластера нет.

Но его можно предоставить с помощью proxy:

```
$ kubectl proxy --port=8080
```

Теперь доступ к сервису можно получить так:

```
http://localhost:8080/api/v1/namespaces/  
<NAMESPACE>/services/<SERVICE-NAME>:<PORT-NAME>/proxy/
```



# Minikube: доступ к сервисам ClusterIP

На предыдущем шаге у нас уже был запущен сервис с типом ClusterIP:

```
$ kubectl get services
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
my-service-cluster-ip	ClusterIP	10.108.193.101	<none>	80/TCP	2m23s

В новом окне терминала запускаем проху:

```
$ kubectl proxy --port=8080
```

```
Starting to serve on 127.0.0.1:8080
```

Теперь мы можем обратиться к нашему сервису по данному адресу:

```
$ curl http://localhost:8080/api/v1/namespaces/default/services/my-service-cluster-ip/proxy/
```

```
Hello from my-deployment-784598767c-dz87v
```

Удаляем наш сервис:

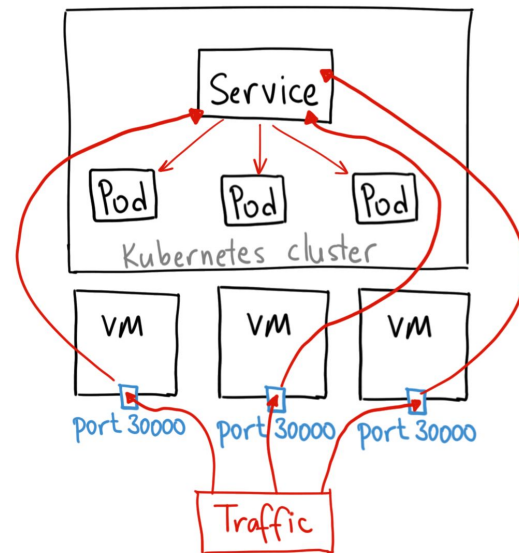
```
$ kubectl delete -f 05_service_cluster_ip.yml
```

# Service: тип сервиса NodePort

Сервису выделяется отдельный IP внутри кластера (по аналогии с ClusterIP) и на каждом узле кластера открывается порт (из диапазона **30000-32767**), который пробрасывается на этот внутренний IP сервиса.

Доступ снаружи кластера осуществляется по данному порту у любого узла кластера (<IP узла>:30000).

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: NodePort
  selector:
    app: my-app
  ports:
    - port: 80
      targetPort: 80
      nodePort: 30000
```



# Minikube: доступ к сервисам NodePort

```
$ kubectl apply -f 06_service_node_port.yml  
service/my-service-node-port created
```

```
$ kubectl get services
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
my-service-node-port	NodePort	10.96.250.242	<none>	80:30000/TCP	114s

```
$ minikube ip  
192.168.64.3
```

```
$ minikube service --url my-service-node-port  
http://192.168.64.3:30000/
```

```
$ curl 192.168.64.3:30000  
Hello from my-deployment-784598767c-rms7m
```

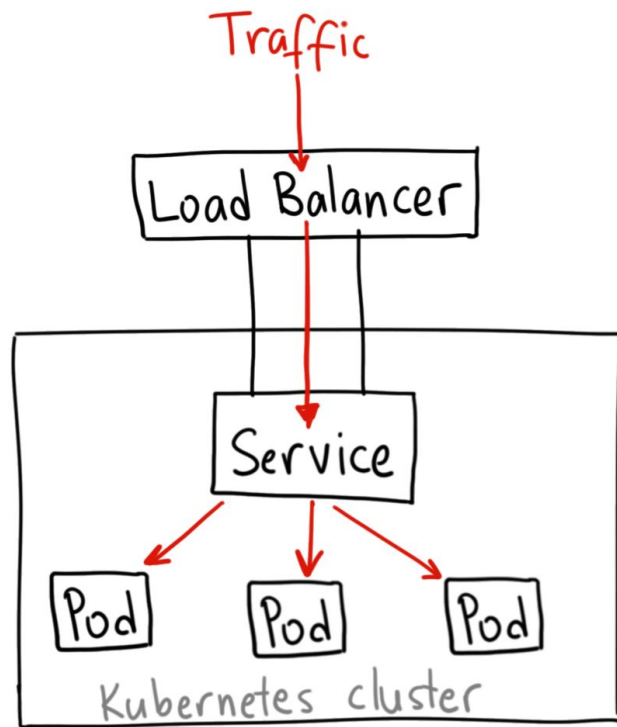
```
$ kubectl delete -f 06_service_node_port.yml
```

# Service: тип сервиса LoadBalancer

Сервис данного типа использует внешний балансировщик (внешний IP), как правило предоставляемый облачным провайдером.

Внешний балансировщик направляет трафик на NodePort и ClusterIP сервисы, который создаются автоматически.

Недостаток этого типа в том, что для каждого сервиса у провайдера будет запрашиваться отдельный внешний IP-адрес, за который, как правило, взимается плата.





# Minikube: доступ к сервисам LoadBalancer

```
$ kubectl apply -f 07_service_load_balancer.yml  
service/my-service-load-balancer created
```

```
$ kubectl get services
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
my-service-load-balancer	LoadBalancer	10.101.73.163	<b>&lt;pending&gt;</b>	80:30844/TCP	23s

В отдельном терминале запускаем туннель:

```
$ minikube tunnel
```

```
$ kubectl get services
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
my-service-load-balancer	LoadBalancer	10.101.73.163	<b>10.101.73.163</b>	80:30844/TCP	23s

```
$ curl 10.101.73.163
```

```
Hello from my-deployment-784598767c-rms7m
```

```
$ kubectl delete -f 07_service_load_balancer.yml
```

# Service: тип сервиса ExternalName

Этот тип сервиса перенаправляет трафик на указанное “externalName” (например, foo.bar.example.com). Это делается путем добавления CNAME-записи в DNS кластера.

По сути, это разновидность сервиса без селектора, в endpoint’е которого указано другое DNS-имя.

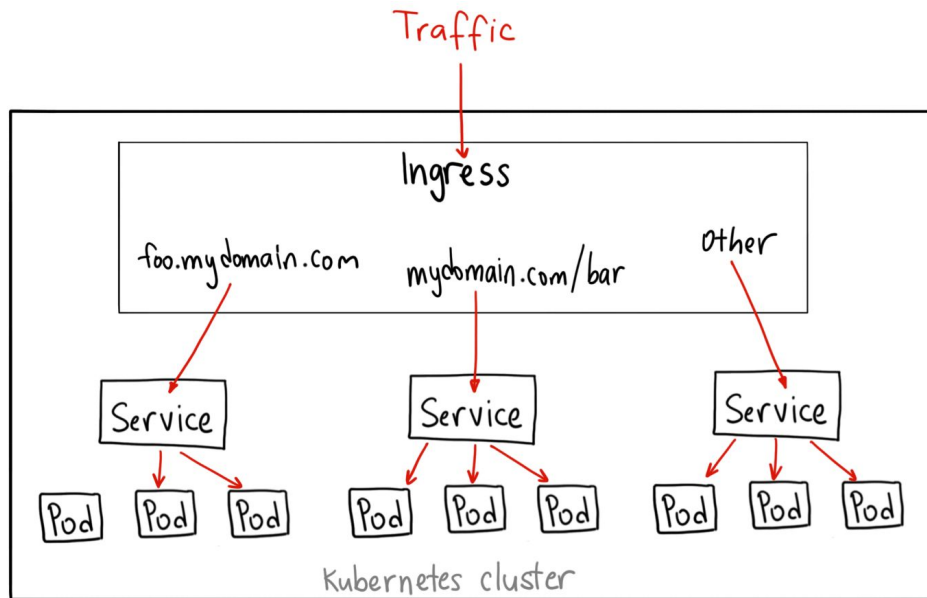
```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: ExternalName
  externalName: my.database.example.com
```

# Ingress: маршрутизатор для сервисов

Это не тип сервиса, но эта абстракция связана с предоставлением доступа к сервисам извне.

Для его использования необходимо дополнительно настроить Ingress Controller.

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: my-ingress
spec:
  rules:
    - host: foo.mydomain.com
      http:
        paths:
          - backend:
              serviceName: foo
              servicePort: 8080
    - host: mydomain.com
      http:
        paths:
          - path: /bar/*
            backend:
              serviceName: bar
              servicePort: 8080
```



# Minikube: доступ к сервисам с помощью Ingress

```
$ kubectl apply -f 06_service_node_port.yml
service/my-service-node-port created
```

```
$ kubectl get services
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
my-service-node-port	NodePort	10.100.29.80	<none>	80:30000/TCP	4s

Устанавливаем NGINX Ingress controller внутрь Minikube:

```
$ minikube addons enable ingress
```

```
The 'ingress' addon is enabled
```

```
$ kubectl apply -f 08_ingress.yml
```

```
ingress.networking.k8s.io/my-ingress created
```

```
$ kubectl get ingresses
```

NAME	CLASS	HOSTS	ADDRESS	PORTS	AGE
my-ingress	<none>	hello-world.info	192.168.64.3	80	40s

```
$ curl hello-world.info --resolve hello-world.info:80:192.168.64.3
```

```
Hello from my-deployment-784598767c-nwzdl
```

```
$ kubectl delete -f 08_ingress.yml -f 06_service_node_port.yml -f 04_deployment_for_service.yml
```

# Job - одноразовая задача

Создает один или несколько `pod`'ов и ожидает успешного их завершения.

Если какие-то `pod`'ы завершились ошибкой, то Job будет запускать их новые копии до тех пор, пока количество успешных выполнений не будет равно заданному.

Типовые применения:

- Запуск тестов
- Применение миграций базы данных
- Выполнение одноразовых скриптов

# Job: пример

```
apiVersion: v1
kind: Job
metadata:
  name: pi
spec:
  completions: 8           ← ожидаемое кол-во успешных выполнений
  parallelism: 4           ← кол-во параллельных запусков
  backoffLimit: 4          ← макс. кол-во попыток
  activeDeadlineSeconds: 60 ← макс. время выполнения
  ttlSecondsAfterFinished: 100 ← макс. время жизни заверченного job'а (alpha-версия)
  template:
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
```

# CronJob - задача, выполняемая по расписанию

Абстракция, которая автоматически создает Job'ы по указанному расписанию.

Расписание запуска Job'ов задается в Cron-формате.

Типовые применения:

- Создание бекапов
- Рассылка писем/уведомлений
- Планирование тяжелых задач на наименее загруженное время

# CronJob: пример

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"           ← расписание в cron-формате
  jobTemplate:                       ← шаблон job'а
    spec:
      template:
        spec:
          containers:
            - name: hello
              image: busybox
              args:
                - /bin/sh
                - -c
                - date; echo Hello from the Kubernetes cluster
          restartPolicy: OnFailure
```



# Volume - абстракция файлового хранилища

Volume решает две проблемы:

- Файловая система контейнера существует только до его удаления/перезапуска.
- Некоторым контейнерам в pod'е может потребоваться общее место для хранения файлов (или общие конфигурационные файлы).
- Изолирует приложение от конкретных технологий хранения данных.

Volume живет только вместе с pod'ом.

# Volume: типы файловых хранилищ

- emptyDir
- hostPath
- configMap
- secret
- persistentVolumeClaim
- ...

# Volume: тип emptyDir

Каталог, создаваемый для pod'а и живущий до тех пор, пока pod не будет удален. Все контейнеры pod'а могут одновременно писать и читать данные из этого volume.

В качестве системы хранения используется файловая система узла, на котором выполняется данный pod.

Существует возможность расположить каталог в оперативной памяти:

```
emptyDir: {  
  medium: Memory  
}
```

```
apiVersion: v1  
kind: Pod  
metadata:  
  name: test-pd  
spec:  
  containers:  
    - image: k8s.gcr.io/test-webserver  
      name: test-container  
      volumeMounts:  
        - mountPath: /cache  
          name: cache-volume  
  volumes:  
    - name: cache-volume  
      emptyDir: {}
```

# Volume: тип hostPath

Каталог узла, подключаемый к pod'у.

Обычно используется для тех случаев, когда pod отслеживает состояние самого узла (предоставляется доступ к системным каталогам /var, /proc, /sys и т.д.).

В качестве системы хранения используется файловая система узла.

```
apiVersion: v1
kind: Pod
metadata:
  name: test-webserver
spec:
  containers:
  - name: test-webserver
    image: k8s.gcr.io/test-webserver:latest
    volumeMounts:
    - mountPath: /var/local/aaa
      name: mydir
    - mountPath: /var/local/aaa/1.txt
      name: myfile
  volumes:
  - name: mydir
    hostPath:
      # Ensure the file directory is created.
      path: /var/local/aaa
      type: DirectoryOrCreate
  - name: myfile
    hostPath:
      path: /var/local/aaa/1.txt
      type: FileOrCreate
```

# ConfigMap - хранилище параметров

API объект, используемый для хранения неконфиденциальных параметров в виде пар “ключ-значение”.

Pod'ы могут использовать значения из ConfigMap'ов несколькими способами:

- Конфигурационные файлы, подключаемые как “volumes”.
- Через переменные окружения
- Через параметры запуска контейнеров

# ConfigMap: пример (volumes)

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-configmap
data:
  default.conf: |
    server {
      listen 80 default_server;
      server_name _;
      default_type text/plain;
      location / {
        return 200 'Hello from $hostname\n';
      }
    }
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: nginx
          image: nginx:1.12
          ports:
            - containerPort: 80
          volumeMounts:
            - name: config
              mountPath: /etc/nginx/conf.d/
      volumes:
        - name: config
          configMap:
            name: my-configmap
```

# ConfigMap: пример (переменные окружения)

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: game-demo
data:
  player_initial_lives: 3
  ui_theme: "dark_moon"
```

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-demo-pod
spec:
  containers:
    - name: demo
      image: game.example/demo-game
      env:
        - name: PLAYER_INITIAL_LIVES
          valueFrom:
            configMapKeyRef:
              name: game-demo
              key: player_initial_lives
        - name: UI_THEME
          valueFrom:
            configMapKeyRef:
              name: game-demo
              key: ui_theme
```

# Secret - конфиденциальные данные

Объект, содержащий конфиденциальное значение (например, пароль, токен, ключ). Хранение важных параметров в виде секретов более безопасно и гибко, чем включение их в конфигурацию pod'а или в образы контейнеров.

Для использования секрета pod'у необходимо указать его.

Секрет может использоваться следующими способами:

- Файл, подключенный через “volume” контейнера.
- Значение переменной окружения.
- kubelet использует секреты для подключения к API-серверу, для загрузки образов из docker registry и т.д.



# Secret: пример (volumes)

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
```

```
data:                                ← base64 encoded
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
```

или

```
stringData:                        ← raw values
  username: admin
  password: 1f2d1e2e67df
```

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
      readOnly: true
    volumes:
    - name: foo
      secret:
        secretName: mysecret
```

```
/etc/foo/username
/etc/foo/password
```

# Secret: пример (переменные окружения)

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
```

```
data:                                ← base64 encoded
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
```

ИЛИ

```
stringData:                        ← raw values
  username: admin
  password: 1f2d1e2e67df
```

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: mypod
      image: redis
      env:
        - name: DB_USERNAME
          valueFrom:
            secretKeyRef:
              name: mysecret
              key: username
        - name: DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysecret
              key: password
```

# Secret: расшифровка исходного значения

```
$ kubectl get secret mysecret -o yaml
```

```
apiVersion: v1
kind: Secret
metadata:
  creationTimestamp: 2016-01-22T18:41:56Z
  name: mysecret
  namespace: default
  resourceVersion: "164619"
  uid: cfee02d6-c137-11e5-8d73-42010af00002
type: Opaque
data:
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
```

```
$ echo 'MWYyZDFlMmU2N2Rm' | base64 --decode
```

# PersistentVolume, PersistentVolumeClaim

**PersistentVolume (PV)** - такой же ресурс кластера, как и узел (только он предоставляет не вычислительные ресурсы, а дисковые тома). Примеры: NFS, iSCSI, RBD, CephFS, Glusterfs, FC (Fibre Channel) и другие. В рамках PV впоследствии можно выделять pod'ам место для хранения данных.

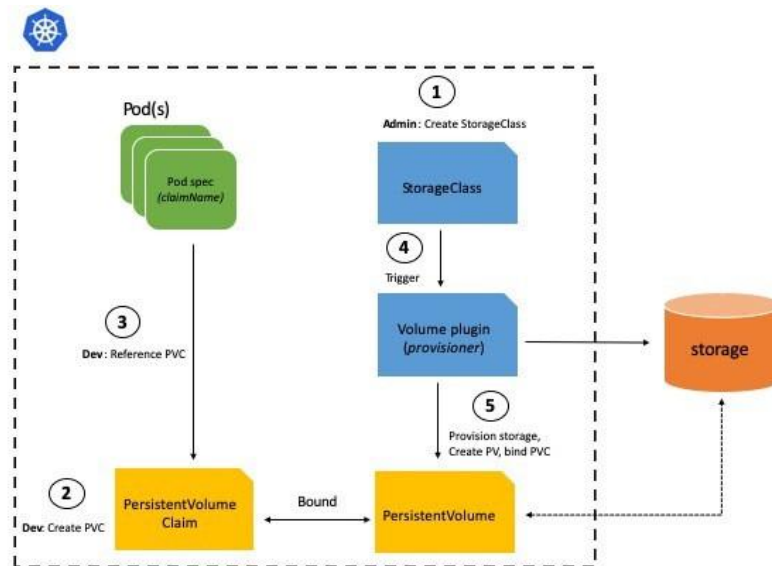
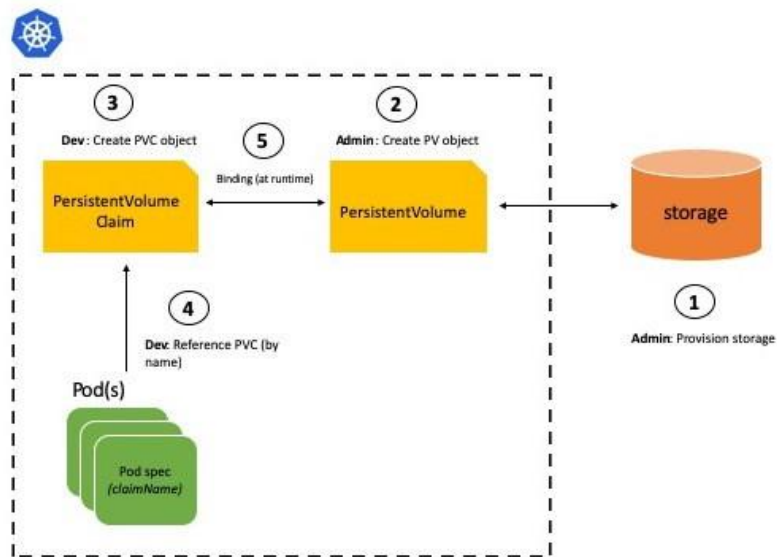
**PersistentVolumeClaim (PVC)** - запрос к PV на выделение места под хранение данных. Это аналог создания pod'а на узле. Pod'ы могут запрашивать определенные ресурсы узла, то же самое делает и PVC. Основные параметры запроса:

- Объем места
- Тип доступа

Типы доступа у PVC могут быть следующие:

- ReadWriteOnce – том может быть смонтирован на чтение и запись к одному pod'у.
- ReadOnlyMany – том может быть смонтирован на много pod'ов в режиме только чтения.
- ReadWriteMany – том может быть смонтирован к множеству pod'ов в режиме чтения и записи.

# Статическое и динамическое создание PersistentVolume'ов



# Volume: тип persistentVolumeClaim (пример)

Создаем каталог на узле, в котором будем хранить данные, и запускаем tunnel:

```
$ minikube ssh
(minikube) $ sudo mkdir /mnt/share && sudo chmod 777 /mnt/share
(minikube) $ exit
```

Запускаем tunnel:

```
$ minikube tunnel
```

Запускаем deployment с подмонтированным каталогом и получаем внешний IP-адрес сервиса:

```
$ kubectl apply -f 09_deployment_for_volume_pvc.yml
$ kubectl get services
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
my-service-load-balancer	LoadBalancer	10.98.14.152	10.98.14.152	80:32565/TCP	37s

# Volume: тип persistentVolumeClaim (пример)

Тестируем наш сервис:

```
$ curl http://10.98.14.152
Hello from my-deployment-86bc66c6c4-cqfls
```

Получаем список файлов в общем каталоге:

```
$ curl http://10.98.14.152/files/
<html>
<head><title>Index of /files/</title></head>
<body bgcolor="white">
<h1>Index of /files/</h1><hr><pre><a href="..">../</a>
</pre><hr></body>
</html>
```

Загружаем файл:

```
$ curl http://10.98.14.152/files/ -T 01_pod.yml
```

Проверяем доступность файла:

```
$ curl http://10.98.14.152/files/01\_pod.yml
```

Удаляем за собой:

```
$ kubectl apply -f 09_deployment_for_volume_pvc.yml
```

# Распределение и балансировка pod'ов по узлам

- nodeSelector, nodeName
- Taints, Tolerations
- Requests, Limits
- Affinity, Anti-Affinity
- cordon, uncordon, drain





# nodeSelector, nodeName

В спецификации pod'a указываются метки узлов (nodeSelector), на которых мы хотим его запустить, или указывается определенный узел (nodeName).

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
  nodeSelector:
    disktype: ssd
```

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
  nodeName: kube-01
```

# Taints, Tolerations

Задаем для узла определенный список блокировок (taints). Если в спецификации pod'a явно не указана сопротивляемость этим блокировкам (tolerations), то он не сможет попасть на этот узел.

Для обхода блокировки pod использует tolerations:

```
apiVersion: v1
kind: Pod
metadata:
  ...
spec:
  containers:
    ...
  tolerations:
  - key: "key"
    operator: "Equal"
    value: "value"
    effect: "NoSchedule"
```

```
apiVersion: v1
kind: Pod
metadata:
  ...
spec:
  containers:
    ...
  tolerations:
  - key: "key"
    operator: "Exists"
    effect: "NoSchedule"
```

Задаем блокировку для узла:

```
$ kubectl taint nodes node1 key=value:NoSchedule
```

Удаляем блокировку для узла:

```
$ kubectl taint nodes node1 key:NoSchedule-
```

Виды блокировок:

## NoSchedule

Не размещать на узле pod'ы без соответствующих tolerations.

## PreferNoSchedule

Предпочитать не размещать на узле pod'ы без соответствующих tolerations (но не требовать).

## NoExecute

Не размещать на узле pod'ы без соответствующих tolerations. Все уже работающие на узле pod'ы без соответствующих tolerations будут удалены с узла.

# Requests, Limits

В спецификации pod'a указывается план (requests) и лимиты (limits) использования ресурсов. Подходящий узел подбирается автоматически с учетом текущей нагрузки.

```
containers:
```

```
- image: nginx:1.12
  name: nginx
  ports:
    - containerPort: 80
```

```
resources:
```

```
  requests:
```

```
    cpu: 50m
```

```
    memory: 100Mi
```

```
  limits:
```

```
    cpu: 100m
```

```
    memory: 100Mi
```

← сколько ресурсов запрашивает контейнер

← при превышении контейнер будет убит (перезапущен)

QoS-классы pod'ов (Quality of Service):

- Guaranteed (limits = requests)
- Burstable (limits > requests)
- BestEffort (не указаны limits и requests)

QoS-класс pod'a можно увидеть в его описании (через `kubectl describe pod ...`).

# Affinity, Anti-Affinity

В спецификации pod'а указываются как требования, так и пожелания к узлам, а также к уже существующим pod'ам на этих узлах.

```
apiVersion: v1
kind: Pod
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: kubernetes.io/e2e-az-name
                operator: In
                values:
                  - e2e-az1
                  - e2e-az2
      preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 1
          preference:
            matchExpressions:
              - key: another-node-label-key
                operator: In
                values:
                  - another-node-label-value
  containers:
    - name: with-node-affinity
      image: k8s.gcr.io/pause:2.0
```

```
apiVersion: v1
kind: Pod
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchExpressions:
              - key: security
                operator: In
                values:
                  - S1
          topologyKey: kubernetes.io/hostname
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 100
          podAffinityTerm:
            labelSelector:
              matchExpressions:
                - key: security
                  operator: In
                  values:
                    - S2
            topologyKey: failure-domain.beta.kubernetes.io/zone
  containers:
    - name: with-pod-affinity
      image: k8s.gcr.io/pause:2.0
```

# cordon, uncordon, drain

Запрет размещать на узле новые pod'ы:

```
$ kubectl cordon $NODENAME
```

Снятие запрета на размещение новых pod'ов:

```
$ kubectl uncordon $NODENAME
```

Запрет размещать на узле новые pod'ы и удаление всех существующих:

```
$ kubectl drain $NODENAME
```

Для последующего ввода узла в строй необходимо выполнить "uncordon".

# Helm - пакетный менеджер для Kubernetes



Установка Helm: <https://helm.sh/docs/intro/install/>

Добавление репозитория (аналог "add-apt-repository"):

```
$ helm repo add stable https://kubernetes-charts.storage.googleapis.com/
```

Обновление репозитория (аналог "apt update"):

```
$ helm repo update
```

Поиск доступных пакетов по ключевым словам:

```
$ helm search repo redis
```

NAME	CHART VERSION	APP VERSION	DESCRIPTION
stable/prometheus-redis-exporter	3.4.1	1.3.4	Prometheus exporter for Redis metrics
stable/redis	10.5.7	5.0.7	DEPRECATED Open source, advanced key-value stor...
stable/redis-ha	4.4.4	5.0.6	Highly available Kubernetes implementation of R...
stable/sensu	0.2.3	0.28	Sensu monitoring framework backed by the Redis ...

Просмотр списка установленных в кластер "пакетов" (во всех namespace'ах):

```
$ helm ls -A
```

# Helm: установка пакета “kube-ops-view”

Исследуем содержимое пакет, скачав его:

```
$ helm fetch stable/kube-ops-view --untar
```

Отдельно скачиваем настройки пакета для их модификации:

```
$ helm inspect values stable/kube-ops-view > kube-ops-view-values.yml
```

Заполним следующие поля в файле “kube-ops-view-values.yml”:

```
ingress:
  enabled: true
  hosts:
    - kube-ops-view.local
```

Добавим данный хост в наш локальный файл “/etc/hosts”:

```
$ sudo sh -c "echo \"$(minikube ip) kube-ops-view.local\" >> /etc/hosts"
```

Включим Ingress для minikube:

```
$ minikube addons enable ingress
```

Установим пакет с нашими настройками:

```
$ helm install --namespace=kube-system ops-view stable/kube-ops-view -f kube-ops-view-values.yml
```

Проверяем доступность приложения: <http://kube-ops-view.local/>

# Helm: обновление и откат, удаление

Проверяем ревизию установленного пакета "kube-ops-view":

```
$ helm ls -A
```

NAME	NAMESPACE	REVISION	UPDATED	STATUS	CHART	APP VERSION
ops-view	kube-system	1	2020-05-28 15:25:28.671502 +0300 MSK	deployed	kube-ops-view-1.1.4	19.9.0

Меняем значение "replicaCount" на "2" в файле "kube-ops-view-values.yml" и обновляем пакет:

```
$ helm upgrade --namespace=kube-system -f kube-ops-view-values.yml ops-view stable/kube-ops-view
```

Проверяем, что обновление завершилось:

```
$ helm history ops-view --namespace=kube-system
```

```
$ helm get values ops-view --namespace=kube-system | grep replicaCount
```

```
replicaCount: 2
```

Откатываем наше изменение:

```
$ helm rollback ops-view 1 --namespace=kube-system
```

```
Rollback was a success! Happy Helming!
```

```
$ helm get values ops-view --namespace=kube-system | grep replicaCount
```

```
replicaCount: 1
```

Удаляем пакет:

```
$ helm uninstall --namespace=kube-system ops-view
```

```
release "ops-view" uninstalled
```



# Дополнительная информация и источники

- Репозиторий с примерами из лекции: [https://git.cloud-team.ru/lections/kubernetes\\_intro](https://git.cloud-team.ru/lections/kubernetes_intro)
- Официальная документация Kubernetes: <https://kubernetes.io/docs/home/>
- Официальная документация Minikube: <https://minikube.sigs.k8s.io/>
- Видео “Введение в Kubernetes - Discovery - Javascript.Ninja”: <https://www.youtube.com/watch?v=L3tgJXsMUTU>
- Видео “Наш опыт с Kubernetes в небольших проектах (Флант, RootConf 2017)”: <https://www.youtube.com/watch?v=CgCLPYJRxbU>
- Статья “Основы Kubernetes”: <https://habr.com/ru/post/258443/>
- Статья “K8S Architecture”: <https://medium.com/@keshiha/k8s-architecture-bb6964767c12>
- Статья “Так что же такое pod в Kubernetes?”: <https://habr.com/ru/company/flant/blog/427819/>
- Статья “Kubernetes Networking Guide for Beginners”: <https://matthewpalmer.net/kubernetes-app-developer/articles/kubernetes-networking-guide-beginners.html>
- Статья “Kubernetes NodePort vs LoadBalancer vs Ingress? When should I use what?”: <https://medium.com/google-cloud/kubernetes-nodeport-vs-loadbalancer-vs-ingress-when-should-i-use-what-922f010849e0>
- Статья “Хранилища данных (Persistent Volumes) в Kubernetes”: <https://serveradmin.ru/hranilishha-dannyh-persistent-volumes-v-kubernetes/>
- Статья “Kubernetes Volumes: the definitive guide (Part 2)”: <https://itnext.io/tutorial-basics-of-kubernetes-volumes-part-2-b2ea6f397402>



Спасибо за внимание!