## ### Управляющий узел (Control Plane)

1. \*\*kube-apiserver\*\*:

- \*\*Функция\*\*: Обеспечивает основной интерфейс управления кластером через REST API. Все взаимодействия (чтение и запись) с состоянием кластера проходят через API-сервер.

- \*\*Безопасность\*\*: Поддерживает аутентификацию и авторизацию, а также механизмы управления доступом на основе ролей (RBAC).

- \*\*Масштабирование\*\*: Поддерживает высокую доступность путем горизонтального масштабирования и балансировки нагрузки.

2. \*\*etcd\*\*:

- \*\*Функция\*\*: Высоконадежное, распределенное хранилище данных ключ-значение, используемое для хранения состояния всего кластера Kubernetes.

- \*\*Репликация и согласованность\*\*: Использует протокол Raft для обеспечения согласованности данных и высокой доступности.

- \*\*Резервное копирование\*\*: Важно регулярно создавать резервные копии данных etcd, так как потеря данных может привести к потере состояния кластера.

3. \*\*kube-scheduler\*\*:

- \*\*Функция\*\*: Назначает поды на узлы кластера на основе ресурсных требований (ЦПУ, память), политики аффинности/антиаффинности, доступных ресурсов и других ограничений.

- \*\*Алгоритмы\*\*: Использует различные алгоритмы и правила для принятия решений о размещении подов, такие как алгоритм фильтрации и алгоритм приоритизации.

4. \*\*kube-controller-manager\*\*:

- \*\*Функция\*\*: Запускает различные контроллеры, которые следят за состоянием кластера и поддерживают его в соответствии с заданной конфигурацией.

- \*\*Контроллеры\*\*:

- \*\*Replication Controller\*\*: Обеспечивает запуск указанного количества реплик определенных подов.

- \*\*Node Controller\*\*: Следит за состоянием узлов и реагирует на их недоступность.

- \*\*Job Controller\*\*: Управляет выполнением краткосрочных задач.

- \*\*Endpoint Controller\*\*: Обновляет объекты Endpoints для сервисов.

- \*\*Service Account & Token Controllers\*\*: Управляют учетными записями и токенами доступа.

## ### Рабочие узлы (Worker Nodes)

1. \*\*kubelet\*\*:

- \*\*Функция\*\*: Основной агент на каждом рабочем узле, который отвечает за управление подами. Он получает инструкции от kube-apiserver и запускает контейнеры через контейнерное окружение (container runtime).

- \*\*Мониторинг\*\*: Следит за состоянием контейнеров и докладывает их состояние обратно в управляющий узел.

2. \*\*kube-proxy\*\*:

- \*\*Функция\*\*: Сетевой прокси, который управляет сетевым взаимодействием между подами и сервисами. Он реализует правила сетевого трафика на каждом узле, обеспечивая доступность сервисов и балансировку нагрузки.

- \*\*Типы проксирования\*\*: Поддерживает разные способы проксирования, включая iptables и IPVS.

3. \*\*Container Runtime\*\*:

- \*\*Функция\*\*: Отвечает за запуск контейнеров. Kubernetes поддерживает несколько контейнерных окружений, таких как Docker, containerd и CRI-O.

- \*\*Интерфейс\*\*: Использует CRI (Container Runtime Interface) для взаимодействия с различными контейнерными средами.

## ### Основные Объекты Kubernetes

1. \*\*Под (Pod)\*\*:

- \*\*Описание\*\*: Основная единица развертывания, которая представляет собой группу контейнеров с общими ресурсами (сетевой стек, файловая система).

- \*\*Жизненный цикл\*\*: Управляется контроллерами, такими как Deployment, ReplicaSet и другие.

2. \*\*ReplicaSet\*\*:

- \*\*Описание\*\*: Обеспечивает поддержание заданного количества подов. Автоматически создает и удаляет поды для достижения желаемого состояния.

3. \*\*Deployment\*\*:

- \*\*Описание\*\*: Высокоуровневый контроллер, который управляет развертыванием и обновлением приложений. Поддерживает стратегии развертывания, такие как RollingUpdate и Recreate.

- \*\*Преимущества\*\*: Позволяет откатывать изменения и управлять историей версий.

4. \*\*Service\*\*:

- \*\*Описание\*\*: Абстракция, которая определяет логический набор подов и политику доступа к ним. Службы обеспечивают постоянный сетевой идентификатор для набора подов.

- \*\*Типы сервисов\*\*:

- \*\*ClusterIP\*\*: Доступен только внутри кластера.

- \*\*NodePort\*\*: Открывает порт на каждом узле, доступный извне.

- \*\*LoadBalancer\*\*: Создает внешний балансировщик нагрузки (если поддерживается облачным провайдером).

- \*\*ExternalName\*\*: Служит для перенаправления на внешние DNS-имена.

5. \*\*ConfigMap и Secret\*\*:

- \*\*ConfigMap\*\*: Хранит неконфиденциальные данные конфигурации в виде ключ-значение.

- \*\*Secret\*\*: Хранит конфиденциальные данные, такие как пароли и токены, в зашифрованном виде.

6. \*\*Namespace\*\*:

- \*\*Описание\*\*: Логическое разделение ресурсов в кластере, позволяющее изолировать разные проекты или команды.

- \*\*Использование\*\*: Полезно для многопользовательских сценариев и управления ресурсами.

## ### Поток работы в Kubernetes

1. \*\*Пользователь создает манифесты (файлы конфигурации) для ресурсов (например, Deployment, Service) и отправляет их в kube-apiserver.\*\*

2. \*\*kube-apiserver проверяет и сохраняет конфигурацию в etcd.\*\*

3. \*\*Контроллеры в kube-controller-manager реагируют на изменения состояния и создают/обновляют ресурсы, чтобы достичь желаемого состояния.\*\*

4. \*\*kube-scheduler назначает новые поды на рабочие узлы, основываясь на доступных ресурсах и ограничениях.\*\*

5. \*\*kubelet на рабочих узлах получает задание от kube-apiserver и запускает контейнеры, следя за их состоянием.\*\*

6. \*\*kube-proxy конфигурирует сетевые правила, чтобы обеспечивать связь между подами и службами.\*\*

Эта подробная архитектура Kubernetes демонстрирует, как различные компоненты взаимодействуют друг с другом для обеспечения надежного и масштабируемого управления контейнеризованными приложениями.

# YAML ИНСТРУКЦИИ ВСЕХ ОБЪЕКТОВ

## 1. Pod

apiVersion: v1  # Указывает на версию API Kubernetes, используемую для создания этого объекта. Для подов используется "v1".

kind: Pod  # Определяет тип объекта, который вы создаете. В данном случае это Pod.

metadata:

  name: my-pod  # Уникальное имя для этого пода в рамках текущего namespace. Имя должно быть DNS-совместимым.

  labels:  # Набор меток для организации и поиска объектов. Метки состоят из пар ключ-значение.

    app: my-app  # Произвольная метка с ключом "app" и значением "my-app". Используется для идентификации и группировки объектов.

spec:

  containers:  # Список контейнеров, которые будут запущены в этом поде. Каждый под может содержать один или несколько контейнеров.

  - name: my-container  # Уникальное имя для этого контейнера в рамках пода.

    image: nginx:latest  # Docker-образ, используемый для создания контейнера. "nginx:latest" указывает на образ Nginx последней версии.

    ports:

    - containerPort: 80  # Порт внутри контейнера, который будет открыт и прослушиваться. В данном случае, это порт 80.

    env:  # Список переменных окружения, которые будут переданы контейнеру.

    - name: ENV\_VAR  # Имя переменной окружения.

      value: value  # Значение переменной окружения.

## 2. Deployment

apiVersion: apps/v1  # Указывает на версию API Kubernetes, используемую для создания объекта Deployment. Для деплоев используется "apps/v1".

kind: Deployment  # Определяет тип объекта, который вы создаете. В данном случае это Deployment.

metadata:

  name: my-deployment  # Уникальное имя для этого деплоя в рамках текущего namespace. Имя должно быть DNS-совместимым.

  labels:  # Набор меток для организации и поиска объектов. Метки состоят из пар ключ-значение.

    app: my-app  # Произвольная метка с ключом "app" и значением "my-app". Используется для идентификации и группировки объектов.

spec:

  replicas: 3  # Количество реплик (подов), которые будут запущены. В данном случае создаются 3 реплики.

  selector:

    matchLabels:

      app: my-app  # Метки, по которым будут выбраны поды для этого деплоя. Эти метки должны совпадать с метками подов.

  template:

    metadata:

      labels:

        app: my-app  # Метки, применяемые к подам, созданным этим деплоем. Эти метки должны совпадать с метками в селекторе.

    spec:

      containers:

      - name: my-container  # Уникальное имя для этого контейнера в рамках пода.

        image: nginx:latest  # Docker-образ, используемый для создания контейнера. "nginx:latest" указывает на образ Nginx последней версии.

        ports:

        - containerPort: 80  # Порт внутри контейнера, который будет открыт и прослушиваться. В данном случае, это порт 80.

        resources:  # Указывает на ресурсы, запрашиваемые и лимитируемые для контейнера.

          requests:

            memory: "64Mi"  # Минимальное количество памяти, которое требуется контейнеру.

            cpu: "250m"  # Минимальное количество CPU, которое требуется контейнеру. 250m означает 0.25 CPU.

          limits:

            memory: "128Mi"  # Максимальное количество памяти, которое может использовать контейнер.

            cpu: "500m"  # Максимальное количество CPU, которое может использовать контейнер. 500m означает 0.5 CPU.

## 3. Service

apiVersion: v1  # Указывает на версию API Kubernetes, используемую для создания объекта Service. Для сервисов используется "v1".

kind: Service  # Определяет тип объекта, который вы создаете. В данном случае это Service.

metadata:

  name: my-service  # Уникальное имя для этого сервиса в рамках текущего namespace. Имя должно быть DNS-совместимым.

  labels:  # Набор меток для организации и поиска объектов. Метки состоят из пар ключ-значение.

    app: my-app  # Произвольная метка с ключом "app" и значением "my-app". Используется для идентификации и группировки объектов.

spec:

  selector:

    app: my-app  # Метки, по которым сервис будет находить поды, которые он обслуживает. Эти метки должны совпадать с метками подов.

  ports:

  - protocol: TCP  # Протокол, используемый сервисом. В данном случае это TCP.

    port: 80  # Порт, на который сервис будет принимать запросы от клиентов.

    targetPort: 80  # Порт внутри пода, на который сервис будет перенаправлять запросы. Этот порт должен совпадать с портом контейнера.

  type: ClusterIP  # Тип сервиса. ClusterIP создает внутренний IP-адрес для использования внутри кластера. Другие типы включают NodePort, LoadBalancer и ExternalName.

## 4. ConfigMap

apiVersion: v1  # Указывает на версию API Kubernetes, используемую для создания объекта ConfigMap. Для конфигурационных карт используется "v1".

kind: ConfigMap  # Определяет тип объекта, который вы создаете. В данном случае это ConfigMap.

metadata:

  name: my-config  # Уникальное имя для этой конфигурационной карты в рамках текущего namespace. Имя должно быть DNS-совместимым.

data:  # Данные, хранящиеся в ConfigMap, представлены в виде пар ключ-значение. Значения хранятся в виде обычного текста.

  config-key: config-value  # Произвольная пара ключ-значение. "config-key" это ключ, а "config-value" это значение.

  another-config-key: another-config-value  # Еще одна произвольная пара ключ-значение. "another-config-key" это ключ, а "another-config-value" это значение.

## 5. Secret

apiVersion: v1  # Указывает на версию API Kubernetes, используемую для создания объекта Secret. Для секретов используется "v1".

kind: Secret  # Определяет тип объекта, который вы создаете. В данном случае это Secret.

metadata:

  name: my-secret  # Уникальное имя для этого секрета в рамках текущего namespace. Имя должно быть DNS-совместимым.

type: Opaque  # Тип секрета. Opaque указывает на произвольные данные, закодированные в base64. Другие типы включают kubernetes.io/tls и kubernetes.io/dockerconfigjson.

data:  # Данные, хранящиеся в секрете, представлены в виде пар ключ-значение. Значения должны быть закодированы в base64.

  username: YWRtaW4=  # Закодированное в base64 значение "admin". "username" это ключ.

  password: MWYyZDFlMmU2N2Rm  # Закодированное в base64 значение "1f2d1e2e67df". "password" это ключ.

## 6. PersistentVolume (PV)

apiVersion: v1  # Указывает на версию API Kubernetes, используемую для создания объекта PersistentVolume. Для персистентных томов используется "v1".

kind: PersistentVolume  # Определяет тип объекта, который вы создаете. В данном случае это PersistentVolume.

metadata:

  name: my-pv  # Уникальное имя для этого персистентного тома в рамках текущего namespace. Имя должно быть DNS-совместимым.

spec:

  capacity:

    storage: 1Gi  # Объем хранилища, выделенного для этого PV. Указывается в единицах измерения, таких как Gi (гигабайты).

  accessModes:

    - ReadWriteOnce  # Режим доступа к тому. ReadWriteOnce позволяет одному узлу читать и записывать данные. Другие режимы включают ReadOnlyMany и ReadWriteMany.

  persistentVolumeReclaimPolicy: Retain  # Политика рекламации для тома. Retain сохраняет данные после удаления PVC. Другие опции включают Recycle и Delete.

  hostPath:

    path: /mnt/data  # Путь на хосте, который будет смонтирован в PV. Используется для тестирования и разработки.

## 7. PersistentVolumeClaim (PVC)

7. PersistentVolumeClaim (PVC)

apiVersion: v1  # Указывает на версию API Kubernetes, используемую для создания объекта PersistentVolumeClaim. Для запросов персистентных томов используется "v1".

kind: PersistentVolumeClaim  # Определяет тип объекта, который вы создаете. В данном случае это PersistentVolumeClaim.

metadata:

  name: my-pvc  # Уникальное имя для этого запроса персистентного тома в рамках текущего namespace. Имя должно быть DNS-совместимым.

spec:

  accessModes:

    - ReadWriteOnce  # Режим доступа, запрашиваемый для тома. Должен совпадать с режимом доступа PV. ReadWriteOnce позволяет одному узлу читать и записывать данные.

  resources:

    requests:

      storage: 1Gi  # Объем хранилища, запрашиваемый для этого PVC. Указывается в единицах измерения, таких как Gi (гигабайты).

## 8.Ingress

apiVersion: networking.k8s.io/v1  # Указывает на версию API Kubernetes, используемую для создания объекта Ingress. Для Ingress используется "networking.k8s.io/v1".

kind: Ingress  # Определяет тип объекта, который вы создаете. В данном случае это Ingress.

metadata:

  name: my-ingress  # Уникальное имя для этого Ingress в рамках текущего namespace. Имя должно быть DNS-совместимым.

  annotations:  # Аннотации для дополнительной информации и конфигурации Ingress. Аннотации состоят из пар ключ-значение.

    nginx.ingress.kubernetes.io/rewrite-target: /  # Пример аннотации для переписывания пути. В данном случае все пути будут переписаны на корень.

spec:

  rules:  # Правила, определяющие как запросы будут обрабатываться Ingress.

  - host: my-app.example.com  # Хост, который будет обслуживаться этим Ingress. В запросах к этому хосту будут применяться следующие правила.

    http:

      paths:  # Список путей и связанных с ними сервисов.

      - path: /  # Путь, который будет обрабатываться (например, /api). В данном случае корневой путь "/".

        pathType: Prefix  # Тип пути (Exact или Prefix). Prefix означает, что все запросы, начинающиеся с указанного пути, будут обрабатываться.

        backend:

          service:

            name: my-service  # Имя сервиса, к которому будут направляться запросы. Сервис должен существовать в том же namespace.

            port:

              number: 80  # Порт сервиса, на который будут направляться запросы. Этот порт должен быть определен в сервисе.

## 9. Конфиг кластера для миникуба

apiVersion: v1  # Версия API, используемая для конфигурации Minikube.

kind: Config  # Определяет тип объекта, в данном случае это конфигурация Minikube.

metadata:

  name: minikube-config  # Уникальное имя для этой конфигурации Minikube.

# Настройки для Minikube

minikube:

  driver: virtualbox  # Драйвер для запуска Minikube. В данном случае используется VirtualBox. Другие возможные значения: docker, vmwarefusion, hyperv, kvm2 и т.д.

  cpus: 4  # Количество CPU, выделяемое для Minikube.

  memory: 8192  # Количество памяти (в МБ), выделяемое для Minikube.

  diskSize: "20g"  # Размер диска, выделяемого для Minikube. Указывается в гигабайтах.

  kubernetesVersion: "v1.20.0"  # Версия Kubernetes для Minikube.

  extraConfig:  # Дополнительные конфигурации для различных компонентов Kubernetes.

    kubelet:  # Настройки для kubelet.

      node-labels: "env=dev,role=minikube"  # Метки для узла Minikube.

    apiserver:  # Настройки для API-сервера.

      authorization-mode: "RBAC,Node"  # Режим авторизации для API-сервера.

# Настройки для сетевых плагинов

networking:

  cni: flannel  # Сетевой плагин (CNI) для Minikube. В данном случае используется Flannel. Другие возможные значения: calico, cilium, weave и т.д.

  podSubnet: "10.244.0.0/16"  # Подсеть для подов (Pod).

# Настройки для хранилища

storage:

  provisioner: standard  # Настройки для провайдера хранилища. В данном случае используется стандартный провайдер.

# Настройки для дополнений (addons)

addons:

  enable:  # Список дополнений, которые необходимо включить.

    - metrics-server  # Включение Metrics Server для сбора метрик узлов и подов.

    - dashboard  # Включение Kubernetes Dashboard для визуального управления кластером.

    - ingress  # Включение Ingress Controller для маршрутизации трафика.

  disable:  # Список дополнений, которые необходимо отключить.

    - default-storageclass  # Отключение стандартного класса хранилища.

# Настройки для прокси

proxy:

  httpProxy: "http://proxy.example.com:8080"  # Адрес HTTP-прокси.

  httpsProxy: "https://proxy.example.com:8080"  # Адрес HTTPS-прокси.

  noProxy: ".example.com,localhost"  # Список доменов, которые не используют прокси.

# Настройки для Docker

docker:

  registryMirrors:  # Список зеркал для Docker Registry.

    - "https://registry.example.com"

# Настройки для DNS

dns:

  domain: "cluster.local"  # DNS-домен для кластера Minikube.

# Настройки для сертификатов

certificates:

  renewal: 8760h0m0s  # Срок действия сертификатов (1 год).

# Настройки для логирования и мониторинга

logging:

  level: "info"  # Уровень логирования (info, debug, warn, error).

monitoring:

  enabled: true  # Включение мониторинга.

# Настройки для профилей Minikube

profiles:

  - name: dev  # Имя профиля.

    cpus: 2  # Количество CPU для этого профиля.

    memory: 4096  # Количество памяти (в МБ) для этого профиля.

  - name: test  # Имя второго профиля.

    cpus: 4  # Количество CPU для этого профиля.

    memory: 8192  # Количество памяти (в МБ) для этого профиля.

## 10. Kubernetes Cluster Configuration (kubeadm)

apiVersion: kubeadm.k8s.io/v1beta3  # Указывает на версию API, используемую kubeadm для настройки кластера. В данном случае это "v1beta3".

kind: ClusterConfiguration  # Определяет тип объекта, который описывает конфигурацию всего кластера.

metadata:

  name: my-cluster  # Уникальное имя для этой конфигурации кластера.

# Основная конфигурация контроллера (Control Plane)

controlPlaneEndpoint: "LOAD\_BALANCER\_DNS:LOAD\_BALANCER\_PORT"  # Указывает на конечную точку доступа контроллера, может быть IP-адресом или DNS-именем балансировщика нагрузки.

apiServer:

  extraArgs:  # Дополнительные аргументы для API-сервера.

    authorization-mode: Node,RBAC  # Режим авторизации, включающий Node и RBAC.

  timeoutForControlPlane: 4m0s  # Таймаут для операций управления контроллером.

controllerManager:

  extraArgs:  # Дополнительные аргументы для менеджера контроллеров.

    bind-address: "0.0.0.0"  # Адрес, на который будет привязан менеджер контроллеров.

scheduler:

  extraArgs:  # Дополнительные аргументы для планировщика.

    bind-address: "0.0.0.0"  # Адрес, на который будет привязан планировщик.

# Настройки сети

networking:

  dnsDomain: "cluster.local"  # DNS-домен для кластера.

  serviceSubnet: "10.96.0.0/12"  # Подсеть для служб (Service).

  podSubnet: "192.168.0.0/16"  # Подсеть для подов (Pod).

# Настройки для различных компонентов кластера

etcd:

  local:

    dataDir: /var/lib/etcd  # Каталог для хранения данных etcd.

# Настройки контейнерного рантайма

containerRuntime:

  name: containerd  # Имя контейнерного рантайма, в данном случае используется containerd.

# Настройки для kube-proxy

kubeProxy:

  config:

    mode: ipvs  # Режим работы kube-proxy, в данном случае используется IPVS.

# Настройки для kubelet

kubeletConfiguration:

  baseConfig:

    cgroupDriver: systemd  # Драйвер cgroup для kubelet.

    clusterDNS:

      - "10.96.0.10"  # IP-адрес службы DNS.

# Настройки для использования внешнего хранилища (например, AWS EBS, GCE PD и т.д.)

storage:

  useCloud: false  # Указывает, использовать ли облачные хранилища.

# Настройки для сетевого плагина (например, Calico, Flannel и т.д.)

networkingPlugin:

  name: calico  # Имя сетевого плагина, в данном случае используется Calico.

  calico:

    ipPools:

      - cidr: "192.168.0.0/16"  # Подсеть, используемая Calico для IP-адресов подов.

    nodeAddressAutodetection: canReach=8.8.8.8  # Автоматическое обнаружение IP-адресов узлов, проверка доступности до 8.8.8.8.

# Настройки для управления версиями Kubernetes (апгрейд, даунгрейд)

kubernetesVersion: "1.20.0"  # Версия Kubernetes для кластера.

imageRepository: "k8s.gcr.io"  # Репозиторий образов для компонентов Kubernetes.

certificatesDir: "/etc/kubernetes/pki"  # Каталог для хранения сертификатов.

clusterName: "my-cluster"  # Имя кластера.

# Настройки для DNS (например, CoreDNS)

dns:

  type: CoreDNS  # Тип DNS-сервиса, в данном случае используется CoreDNS.

  imageRepository: "k8s.gcr.io/coredns"  # Репозиторий образов для DNS-сервиса.

  imageTag: "v1.7.0"  # Тег образа для DNS-сервиса.

# Настройки для аудита и логирования

auditPolicy:

  logPath: "/var/log/kubernetes/audit.log"  # Путь к файлу журналов аудита.

  logMaxAge: 30  # Максимальное количество дней для хранения журналов аудита.

  logMaxBackup: 10  # Максимальное количество резервных копий журналов аудита.

  logMaxSize: 100  # Максимальный размер журнала аудита в мегабайтах.

# Настройки для управления сертификатами

certificates:

  duration: 8760h0m0s  # Срок действия сертификатов (1 год).

  renewBefore: 240h0m0s  # Время до истечения срока действия сертификатов для их обновления (10 дней).

# Настройки для контроля RBAC (Role-Based Access Control)

rbac:

  enabled: true  # Включение RBAC для управления доступом.

# Настройки для Node (например, taints, labels)

nodeRegistration:

  name: "my-node"  # Имя узла.

  taints:  # Настройки taints для узла.

    - key: "key"

      value: "value"

      effect: "NoSchedule"  # Эффект taints (NoSchedule, PreferNoSchedule, NoExecute).

  kubeletExtraArgs:  # Дополнительные аргументы для kubelet.

    node-labels: "role=worker"  # Метки для узла.

# Настройки для API-сервера

apiServerExtraVolumes:

  - name: "audit-log"

    hostPath: "/var/log/kubernetes/audit.log"

    mountPath: "/var/log/kubernetes/audit.log"

    readOnly: false  # Указывает, что путь не является только для чтения.

# Настройки для авторизации и аутентификации

authentication:

  webHook: null  # Настройки веб-хуков для аутентификации.

  tokenWebhookConfigFile: null  # Файл конфигурации веб-хуков для токенов.

# Настройки для облачной интеграции

cloudProvider:

  name: "aws"  # Имя облачного провайдера, в данном случае AWS.

  config:

    global:

      disable-security-group-ingress: false  # Отключение автоматического управления группами безопасности.

# Настройки для мониторинга и логирования

monitoring:

  enabled: true  # Включение мониторинга.

  prometheus:

    enabled: true  # Включение Prometheus для сбора метрик.

# Настройки для других компонентов и плагинов

addons:

  - name: "metrics-server"

    enabled: true  # Включение Metrics Server для сбора метрик узлов и подов.

+--------------------------------------+

| Master Node |

| +-----------------------------+ |

| | API Server | |

| +-----------------------------+ |

| +-----------------------------+ |

| | etcd | |

| +-----------------------------+ |

| +-----------------------------+ |

| | Controller Manager | |

| +-----------------------------+ |

| +-----------------------------+ |

| | Scheduler | |

| +-----------------------------+ |

+--------------------------------------+

/|\ /|\

| |

| |

| |

| |

| |

| |

| |

| |

+--------------------------------------+ +--------------------------------------+

| Worker Node 1 | | Worker Node 2 |

| +-----------------------------+ | | +-----------------------------+ |

| | Kubelet | | | | Kubelet | |

| +-----------------------------+ | | +-----------------------------+ |

| +-----------------------------+ | | +-----------------------------+ |

| | Kube-proxy | | | | Kube-proxy | |

| +-----------------------------+ | | +-----------------------------+ |

| +-----------------------------+ | | +-----------------------------+ |

| | Container Runtime | | | Container Runtime | |

| +-----------------------------+ | | +-----------------------------+ |

| +-----------------------------+ | | +-----------------------------+ |

| | Pod | | | | Pod | |

| | +-------------------------+ | | | | +-------------------------+ | |

| | | Container | | | | | | Container | | |

| | +-------------------------+ | | | | +-------------------------+ | |

| +-----------------------------+ | | +-----------------------------+ |

| +-----------------------------+ | | +-----------------------------+ |

| | Pod | | | | Pod | |

| | +-------------------------+ | | | | +-------------------------+ | |

| | | Container | | | | | | Container | | |

| | +-------------------------+ | | | | +-------------------------+ | |

| +-----------------------------+ | | +-----------------------------+ |

+--------------------------------------+ +--------------------------------------+

/|\ /|\

| |

| |

| |

| |

| |

| |

| |

+--------------------------------------+

| External Clients |

| +-----------------------------+ |

| | Ingress | |

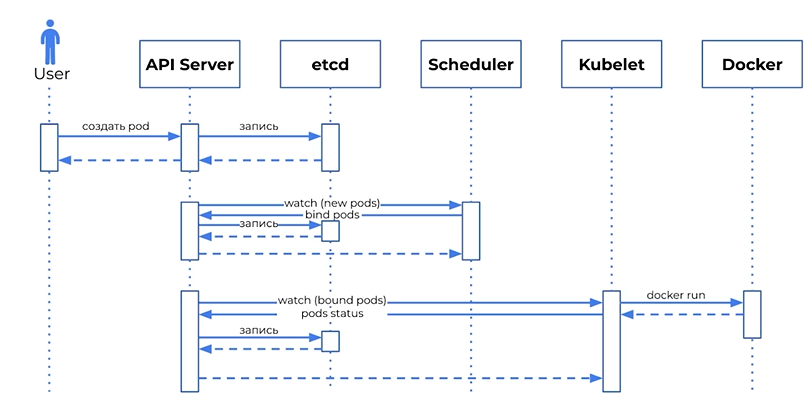
| +-----------------------------+ |

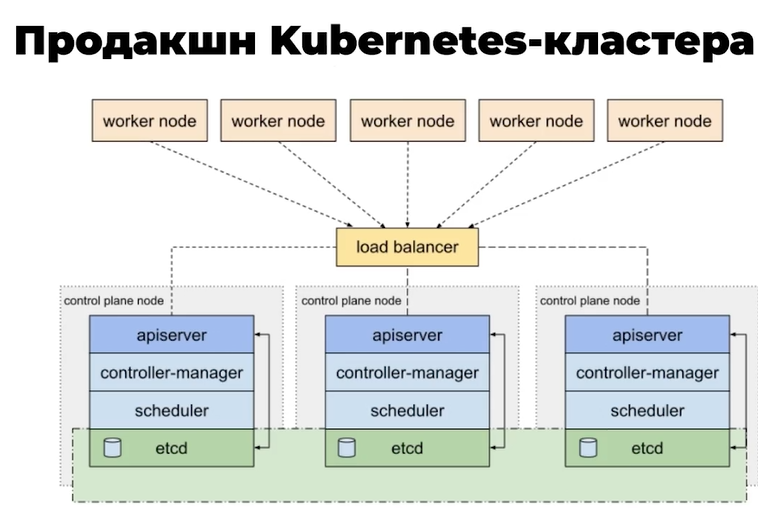
| +-----------------------------+ |

| | Service | |

| +-----------------------------+ |

+--------------------------------------+





# ИНСТРУКЦИИ KUBECTL

## Основные команды

Общая информация о кластере:

kubectl cluster-info

Информация о компонентах кластера:

kubectl get componentstatuses

Список всех узлов (нод):

kubectl get nodes

Подробная информация о ноде:

kubectl describe node <node\_name>

## Работа с Namespaces

Список всех namespaces:

kubectl get namespaces

Создание namespace:

kubectl create namespace <namespace\_name>

Удаление namespace:

kubectl delete namespace <namespace\_name>

## Работа с Pods

Список всех подов:

kubectl get pods

Список подов в указанном namespace:

kubectl get pods -n <namespace>

Подробная информация о поде:

kubectl describe pod <pod\_name>

Логи пода:

kubectl logs <pod\_name>

Запуск шелла в контейнере пода:

kubectl exec -it <pod\_name> -- /bin/bash

## Работа с Deployments

Список всех deployments:

kubectl get deployments

Создание deployment из YAML файла:

kubectl apply -f <file.yaml>

Обновление deployment:

kubectl set image deployment/<deployment\_name> <container\_name>=<new\_image>

Откат к предыдущей версии deployment:

kubectl rollout undo deployment/<deployment\_name>

## Работа с Services

Список всех сервисов:

kubectl get services

Подробная информация о сервисе:

kubectl describe service <service\_name>

## Работа с ConfigMaps и Secrets

Список всех ConfigMaps:

kubectl get configmaps

Создание ConfigMap из файла:

kubectl create configmap <configmap\_name> --from-file=<file\_path>

Список всех Secrets:

kubectl get secrets

Создание Secret из файла:

kubectl create secret generic <secret\_name> --from-file=<file\_path>

## Работа с RBAC (Role-Based Access Control)

Список всех ролей:

kubectl get roles

Список всех rolebindings:

kubectl get rolebindings

Создание роли из YAML файла:

kubectl apply -f <role.yaml>

Создание привязки роли (rolebinding) из YAML файла:

kubectl apply -f <rolebinding.yaml>

## Полезные опции

Переключение namespace:

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

Фильтрация вывода по меткам (labels):

kubectl get pods -l <label\_selector>

Сортировка вывода:

kubectl get pods --sort-by=.metadata.name

Вывод в формате YAML/JSON:

kubectl get pods -o yaml

kubectl get pods -o json

# PODS

## ИНСТРУКЦИИ. Создание и управление Pods

Создание пода из YAML файла:

kubectl apply -f <file.yaml>

Удаление пода:

kubectl delete pod <pod\_name>

Перезапуск пода:

kubectl delete pod <pod\_name> --grace-period=0 --force

Список подов:

kubectl get pods

Список подов в указанном namespace:

kubectl get pods -n <namespace>

Подробная информация о поде:

kubectl describe pod <pod\_name>

Логи пода:

kubectl logs <pod\_name>

Логи конкретного контейнера в поде:

kubectl logs <pod\_name> -c <container\_name>

Потоковые логи пода:

kubectl logs -f <pod\_name>

Потоковые логи конкретного контейнера в поде:

kubectl logs -f <pod\_name> -c <container\_name>

Запуск шелла в контейнере пода:

kubectl exec -it <pod\_name> -- /bin/bash

Выполнение команды в контейнере пода:

kubectl exec <pod\_name> -- <command>

Проброс порта из пода на локальную машину:

kubectl port-forward <pod\_name> <local\_port>:<remote\_port>

Копирование файлов из пода на локальную машину:

kubectl cp <pod\_name>:<path/to/file> <local\_path>

Копирование файлов с локальной машины в под:

kubectl cp <local\_path> <pod\_name>:<path/to/file>

Фильтрация подов по меткам (labels):

kubectl get pods -l <label\_selector>

Сортировка вывода подов:

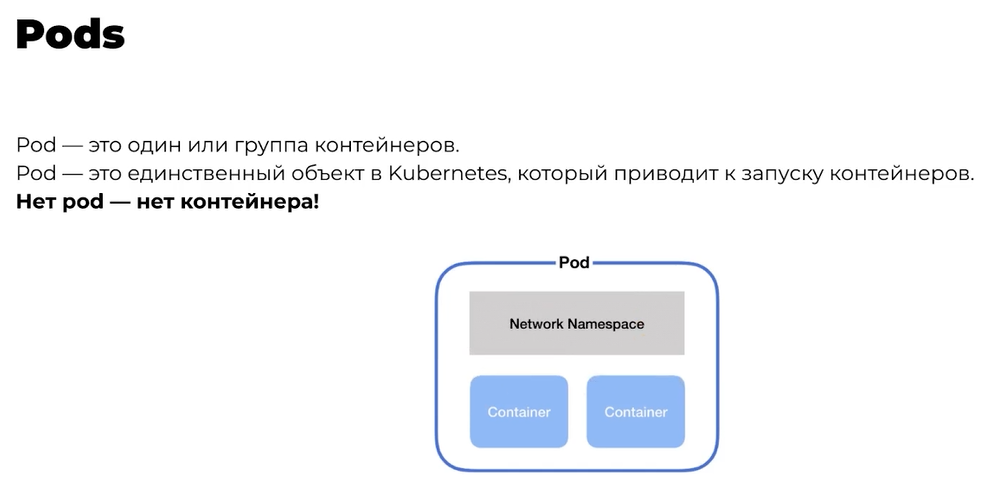
kubectl get pods --sort-by=.metadata.name

Вывод списка подов в формате YAML:

kubectl get pods -o yaml

Вывод списка подов в формате JSON:

kubectl get pods -o json



## Пример спецификации pod

apiVersion: v1

kind: Pod

metadata:

 name: my-pod

spec:

 containers:

  - name: nginx-container

    image: nginx

Спецификация Pod в Kubernetes описывает один или несколько контейнеров, которые будут запущены в этом Pod. Давай разберем пример спецификации Pod по строкам:

# ПРАКТИКА

## Установка

<https://minikube.sigs.k8s.io/docs/start/?arch=%2Flinux%2Fx86-64%2Fstable%2Fbinary+download>

1. устанавливаем на убунту сервер бинарники  
   **curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64  
   sudo install minikube-linux-amd64 /usr/local/bin/minikube && rm minikube-linux-amd64**
2. Устанавливаем kubectl <https://kubernetes.io/docs/tasks/tools/#kubectl>  
   **curl -LO https://dl.k8s.io/release/$(curl -L -s ttps://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl  
   sudo install -o root -g root -m 0755 kubect l /usr/local/bin/kubectl**
3. Для удобства делаем алиас в .bashrc в каталоге пользователя **alias k='kubectl'**

## Первые шаги

1. Запускаем кластер **minikube start**
   1. Минипрактика  
      kubectl create deployment hello-minikube --image=docker.io/nginx:1.23

kubectl expose deployment hello-minikube --type=NodePort --port=80

* 1. Узнать подробности о деплойменте, поде, реплике
     1. **kubectl describe deployment hello-minikube**
     2. **kubectl** get services hello-minikube
     3. **minikube** service hello-minikube
     4. kubectl get replicaset
     5. kubectl get po
     6. kubectl describe pod hello-minikube-6d8d87db86-7r5ld
  2. kubectl port-forward service/hello-minikube --address=192.168.0.173 7080:80 Сделаем проброс и проверим в браузере на хосте. Все работает

1. **minikube status**
2. **minikube ssh** тоже самое чтое **docker exec -it minikube sh**
3. config кластера хранится ~/.kube/config

apiVersion: v1

clusters:

- cluster:

    certificate-authority: /home/alex/.minikube/ca.crt

    extensions:

    - extension:

        last-update: Fri, 07 Jun 2024 10:07:46 UTC

        provider: minikube.sigs.k8s.io

        version: v1.33.1

      name: cluster\_info

    server: https://192.168.49.2:8443

  name: minikube

contexts:

- context:

    cluster: minikube

    extensions:

    - extension:

        last-update: Fri, 07 Jun 2024 10:07:46 UTC

        provider: minikube.sigs.k8s.io

        version: v1.33.1

      name: context\_info

    namespace: default

    user: minikube

  name: minikube

current-context: minikube

kind: Config

preferences: {}

users:

- name: minikube

  user:

    client-certificate: /home/alex/.minikube/profiles/minikube/client.crt

    client-key: /home/alex/.minikube/profiles/minikube/client.key

Тот же самый конфик можно вызвать **kubectl config view**

1. **kubectl cluster info**
2. Посмотреть ноды в кластере **kubectl get no**
   1. Подробнее **kubectl get no -o wide**
3. **kubectl version**  можем посмотреть версию теперь и сервера и клиента

# Pods

1. **kubectl get pods**  смотрим список подов
2. **kubectl get namespaces**  Podы сгруппированы и разделены между собой неймспейсами
3. **kubectl get pods -n kube-system** Просмотреть поды конекретного неймспейса
4. **kubectl get po -A** просмотреть все поды к которым есть доступ
5. Создаем pod.yaml в /.kube

apiVersion: v1

kind: Pod

metadata:

 name: my-pod

spec:

 containers:

  - name: nginx-container

    image: nginx

1. **kubectl create -f pod.yaml** создаем под. Создается новый объект. kubectl apply применяет правила к существующему объекту
2. kubectl get pods проверяем
3. Статусы запуска pod
   1. Pending - объект создан
   2. Running - все контейнеры Poda запущены
   3. Succeeded - все контейнеры корректно завершили работу
   4. Failed -- все контейнеры остановлены
   5. Unknown - kube-apiserver не в курcе, что происходит с подом
4. kubectl logs my-pod просмотр логов пода. Если несколько контейнеров, то с параметром -с указываем конкретный контейнер

Собираем приложение и Pod

Кладем requirements.txt c текстом Flask

app.py

from flask import Flask, jsonify

import datetime

app = Flask(\_\_name\_\_)

@app.route('/date')

def index():

    dateNow = datetime.date.today().strftime("%m/%d/%Y")

    return jsonify(date=dateNow)

if \_\_name\_\_ == '\_\_main\_\_':

    app.run(debug=False, host='0.0.0.0', port='8080')

Собираем docker образ

FROM python:3.8-slim-buster

WORKDIR /app

COPY requirements.txt requirements.txt

RUN pip3 install -r requirements.txt

COPY . .

CMD [ "python3", "app.py"]

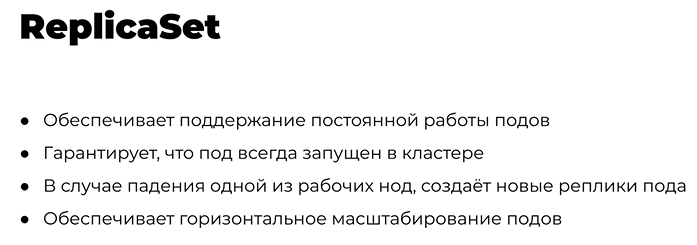
**docker build -t server:1.0 .**

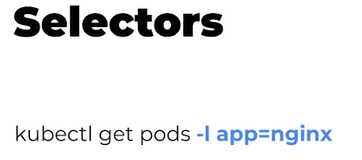
Кладем его в реестр миникуба **minikube image load server:1.0**

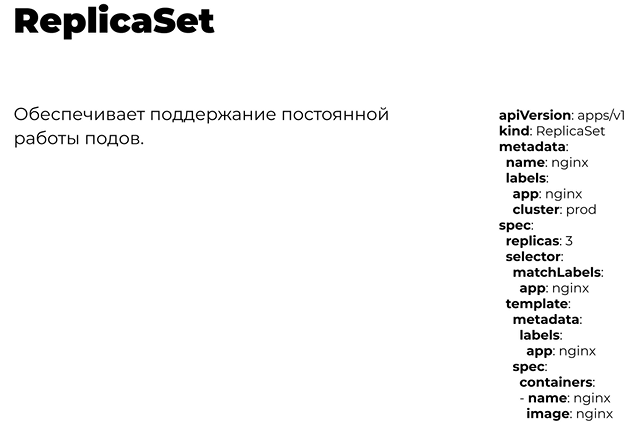
Проверяем все доступные образы в реестре миникуба **minikube image list**

# ReplicaSet

## Теория

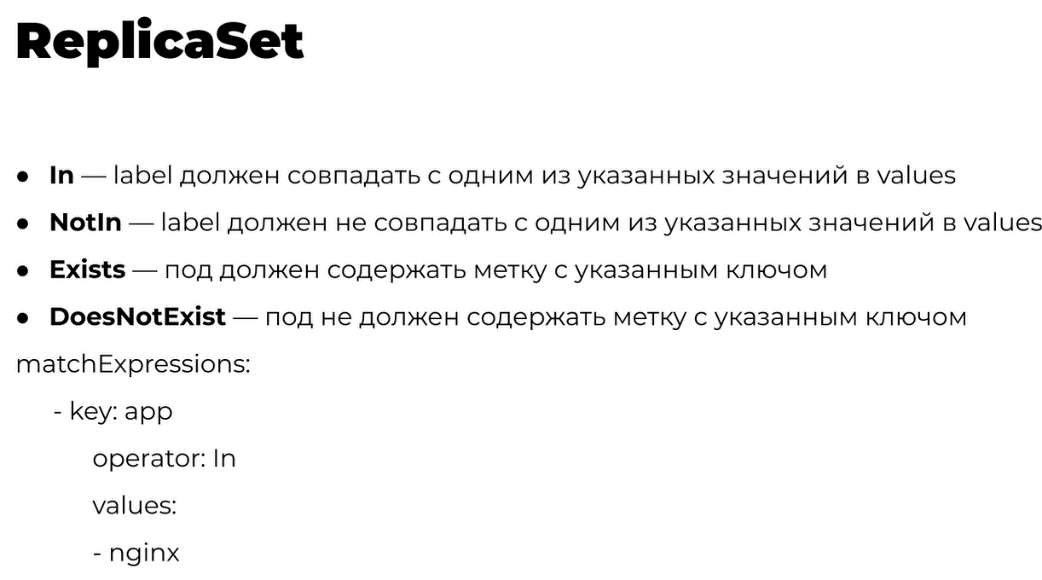
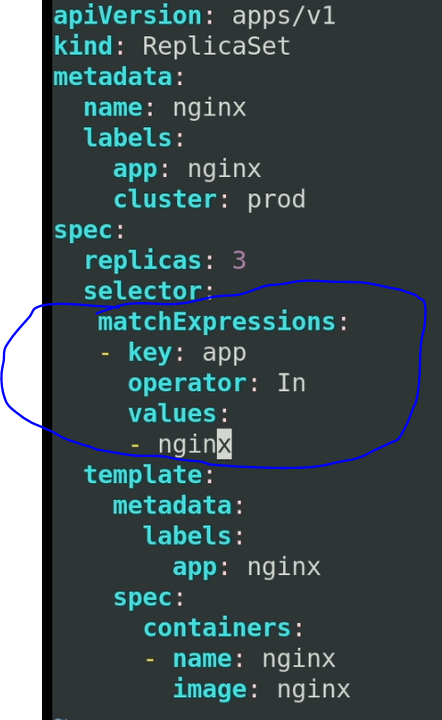




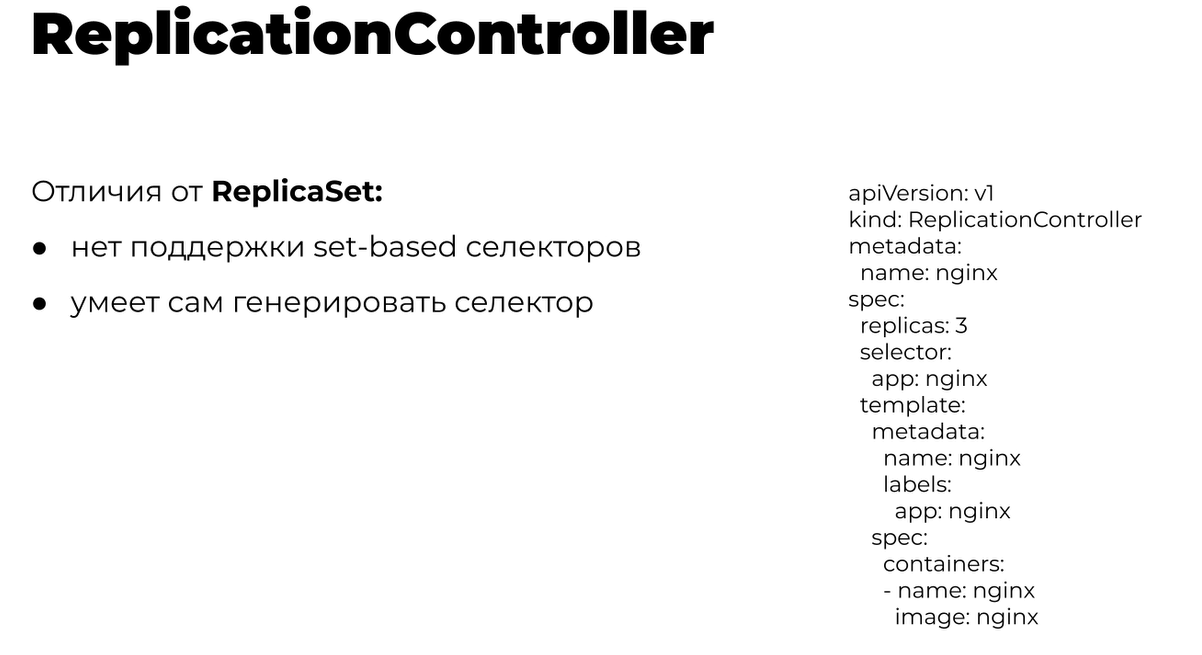


Если мы меняем в шаблоне что-то, то репликасет создает новые поды по этому шаблону, но старые созданные не удаляет и не выключает

Set-based селекторы:



Это новая версия ReplicationController



## Продолжение Практики

Создадим yaml файл

apiVersion: apps/v1

kind: ReplicaSet

metadata:

  name: frontend

  labels:

    app: nginx

    cluster: prod

spec:

  replicas: 3

  selector:

    matchLabels:

      app: nginx

  template:

    metadata:

      labels:

        app: nginx

    spec:

      containers:

      - name: nginx

        image: nginx

И применим его **kubectl apply -f replica.yaml**

Получить список репликасетов **kubectl get replicaset**

Получим список подов **kubectl get po** и отредактируем нужный **kubectl edit pod nginx-hdddz**

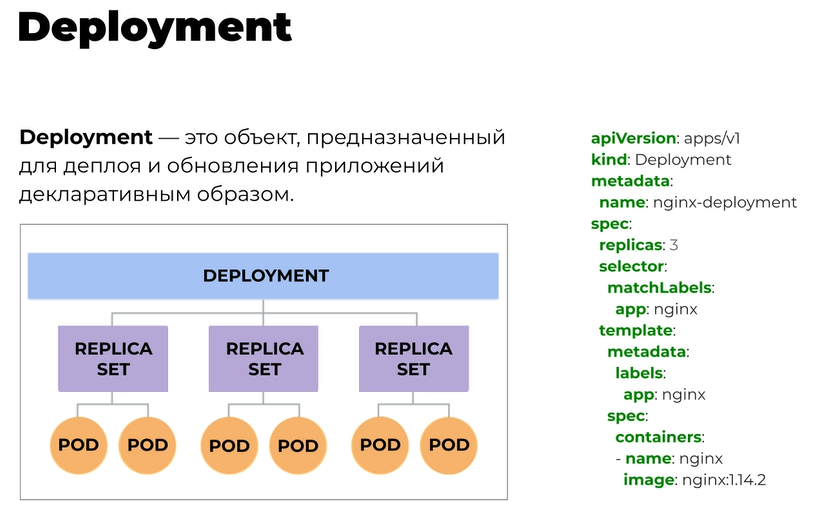
Увеличим число желаемых подов до 5 **kubectl scale replicaset nginx --replicas=5**

Удалим текущий replicaset **kubectl delete replicaset nginx**

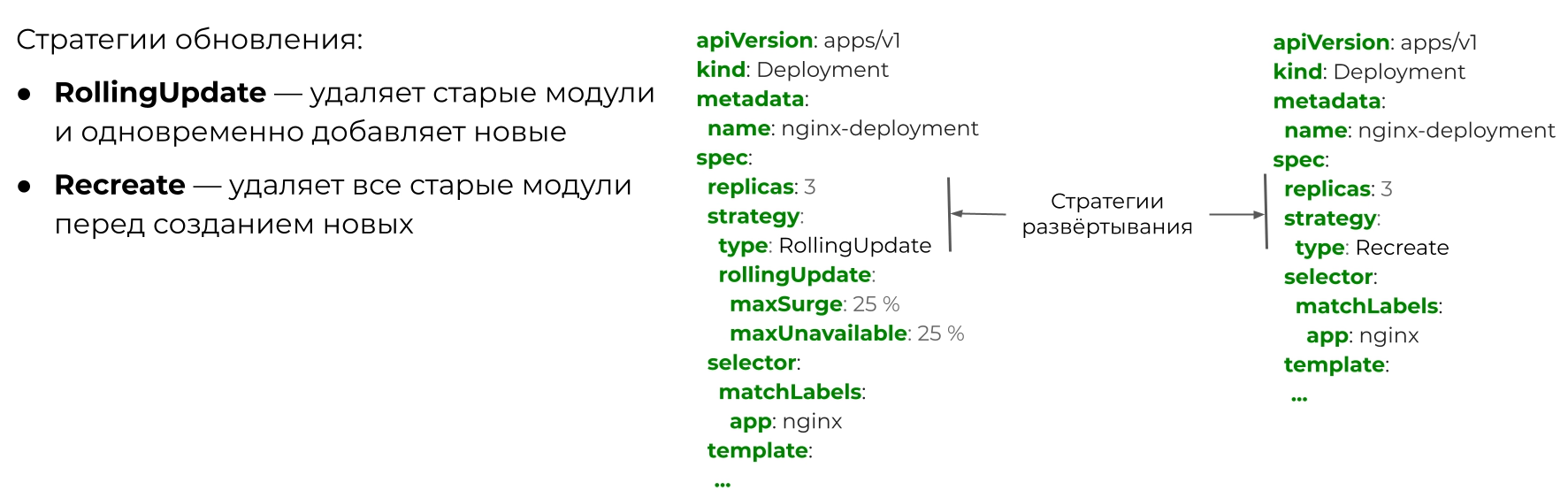
Удаление replicaset без подов **kubectl delete replicaset nginx --cascade=false**

# DEPLOYMENT

Объект для создания и обновления приложения декларативным способом. Координирует создание и изменение двух и более репликасетов во время обновления приложения



## Стратегии обновления



# ПРАКТИКА ОСНОВЫ

## Установка и первые шаги

<https://minikube.sigs.k8s.io/docs/start/?arch=%2Flinux%2Fx86-64%2Fstable%2Fbinary+download>

1. устанавливаем на убунту сервер бинарники  
   **curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64  
   sudo install minikube-linux-amd64 /usr/local/bin/minikube && rm minikube-linux-amd64**
2. Устанавливаем kubectl <https://kubernetes.io/docs/tasks/tools/#kubectl>  
   **curl -LO https://dl.k8s.io/release/$(curl -L -s ttps://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl  
   sudo install -o root -g root -m 0755 kubect l /usr/local/bin/kubectl**
3. Для удобства делаем алиас в .bashrc в каталоге пользователя **alias k='kubectl'**
4. **minikube ip** узнаем ip кластера
5. Посмотреть поды для неймспейса **kubectl get pods --namespace=kube-system**
6. Создание пода командой **kubectl run my-nginx-pod --image=nginx**
7. **kubectl describe pod my-nginx-pod**  проанализировать все детали и события запуска пода. Узнали ip пода
8. заходим на ноду **minikube ssh** С ноды под доступен через **ping 10.244.0.34.** Проверить можно не через браузер вес сайт а через curl, например **curl 10.244.0.34** и видим htlml текст
9. **kubectl get pods -o wide** подробности о поде включая ip
10. **kubectl delete pod my-nginx-pod**

## Deployment

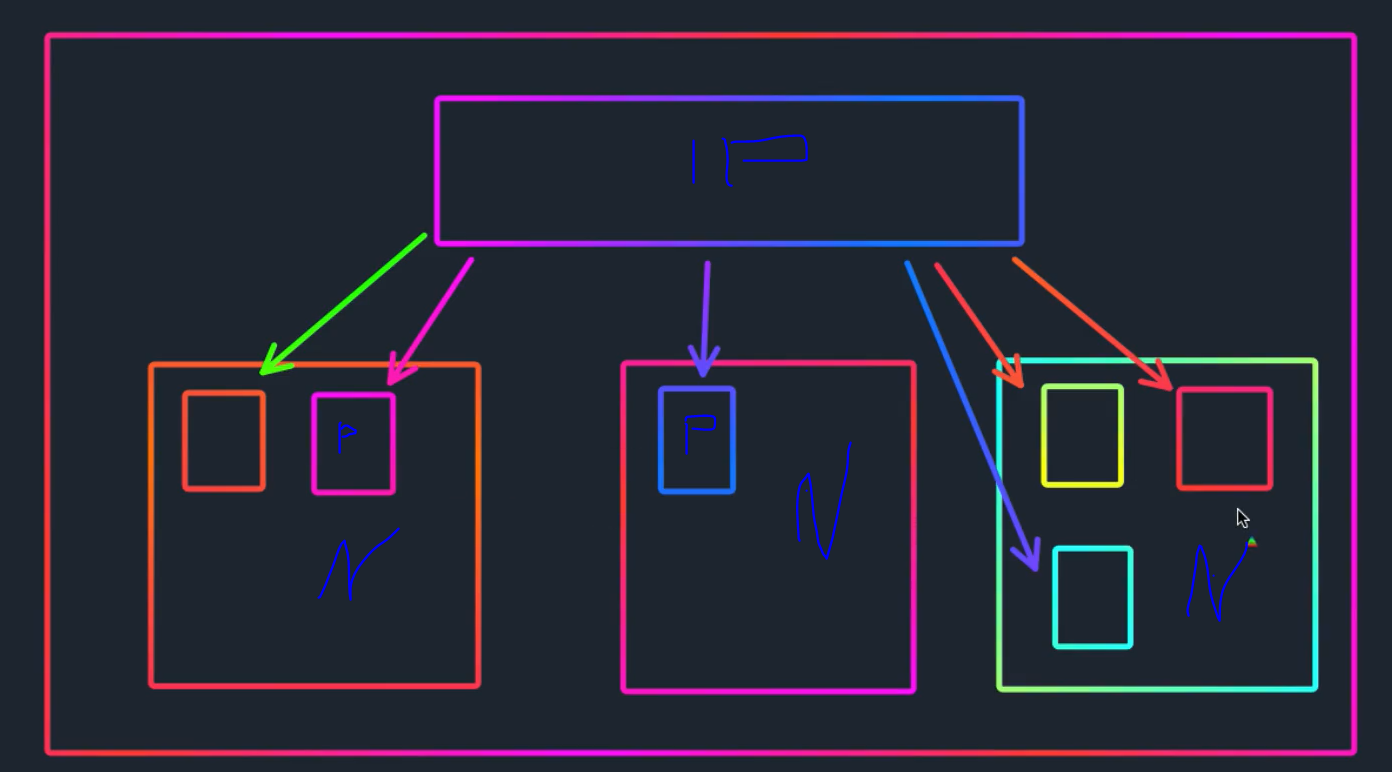
1. **kubectl create deployment my-nginx-deploy --image=nginx** Создаем деплоймент
2. **k describe deployment my-nginx-deploy**  Подробности нашего деплоймента. В **Selector** указывается лейбл для поиска соответствующих подов
3. **kubectl scale deploy my-nginx-deploy --replicas=3** увеличение или уменьшение числа подов в деплойменте с помощью репликасет до 3
4. **kubectl get -o wide po** просмотр ip подов, мы можем подключиться к кластеру и проверить доступность их, а так же курлом что выдают
5. **k delete pod my-nginx-deploy-7b69c488c9-tgzh6** Удаляем наш под, при этом создается вместо него другой под автоматически, т.к. в деплойменте у нас есть желаемое количество ресурсов

## Services

Позволяет используя один ip подключаться к подам рандомно. Создается по умолчанию при создании деплоймента

## Cluster ip

Доступен только внутри кластера кубернетес и не доступен снаружи



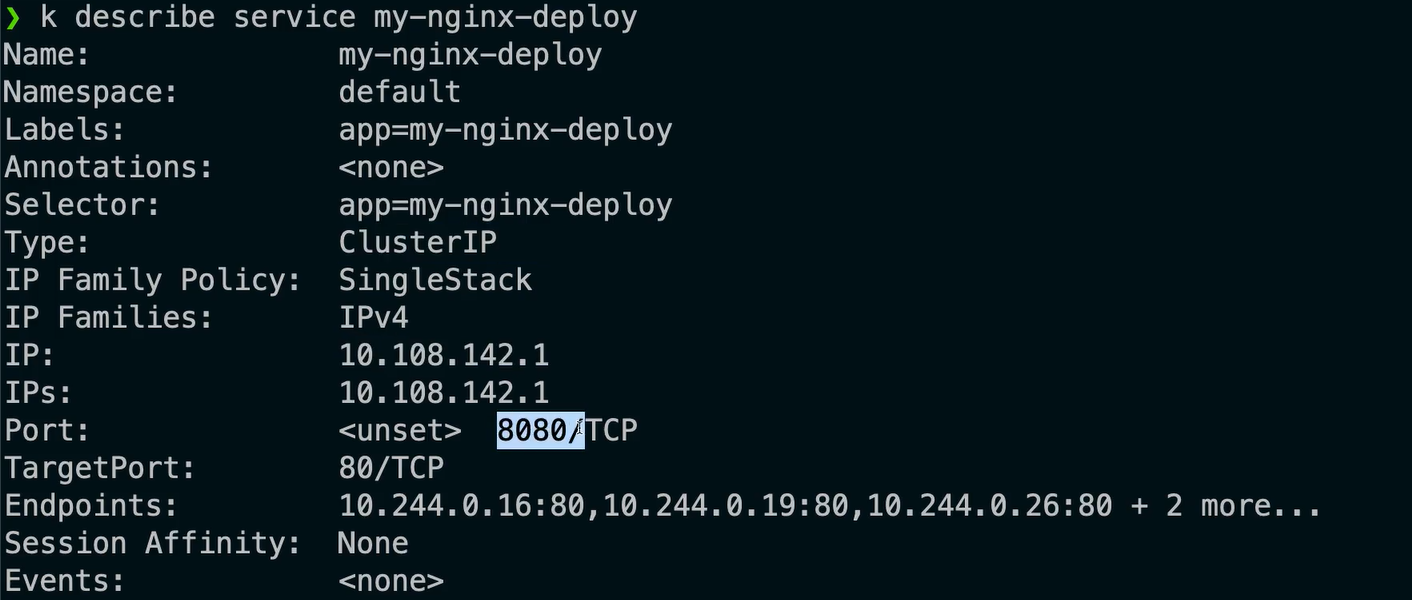
**kubectl get services** или **kubectl get svc**

1. Создаем сервис для деплоймента

**kubectl expose deploy my-nginx-deploy --port=8080 --target-port=80**

**--port** открываем порт для доступа к нашему деплойменту

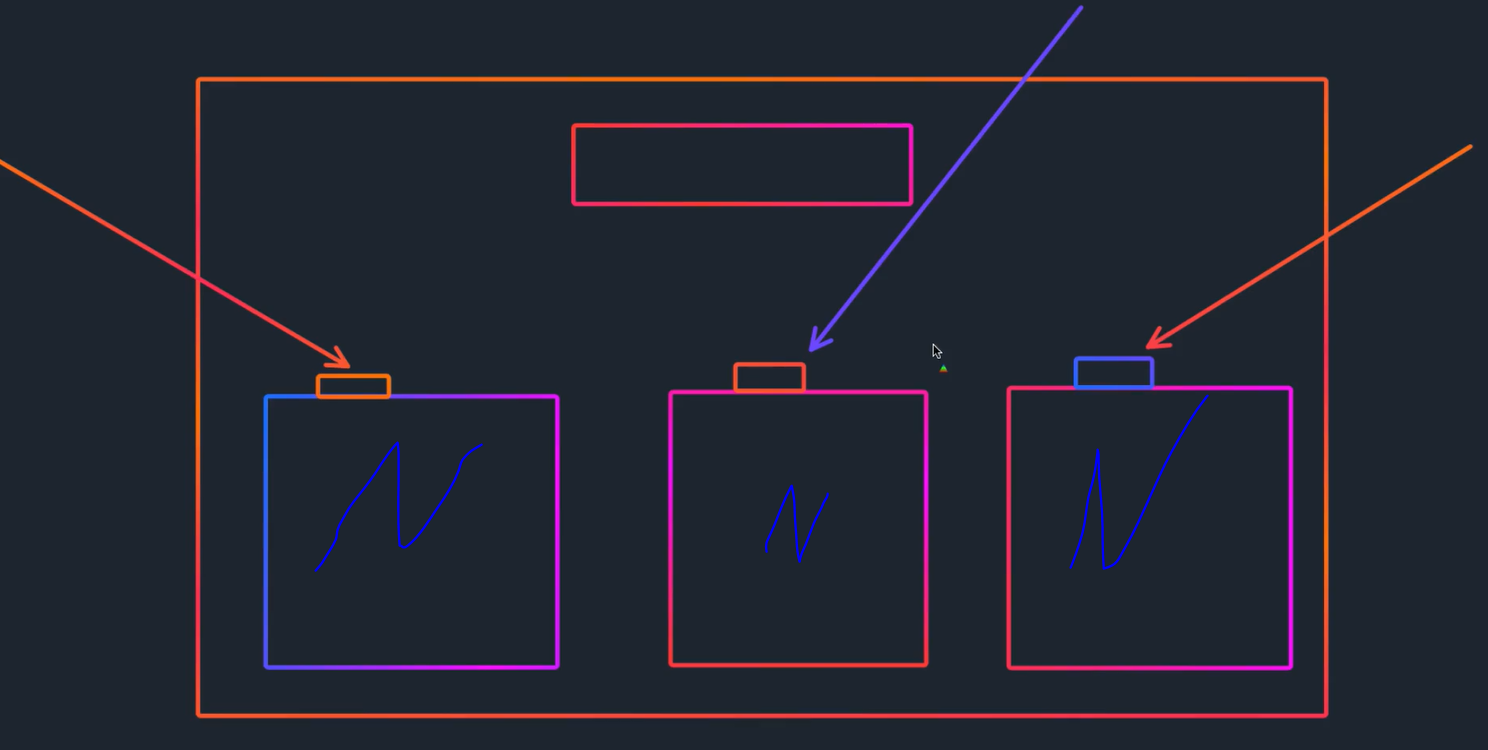
1. **k get svc**
2. **k describe svc my-nginx-deploy**

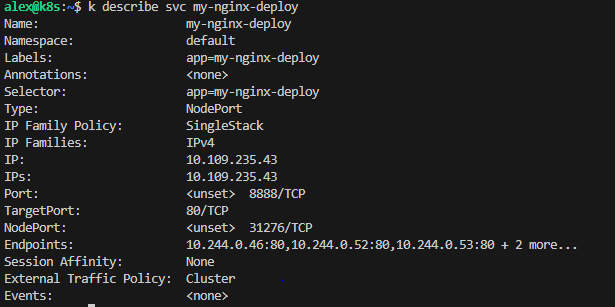


1. Проверяме подключение к случайному поду
   1. **minikube ssh**
   2. **curl 10.100.132.48:8080** Получаем html

## NodePort

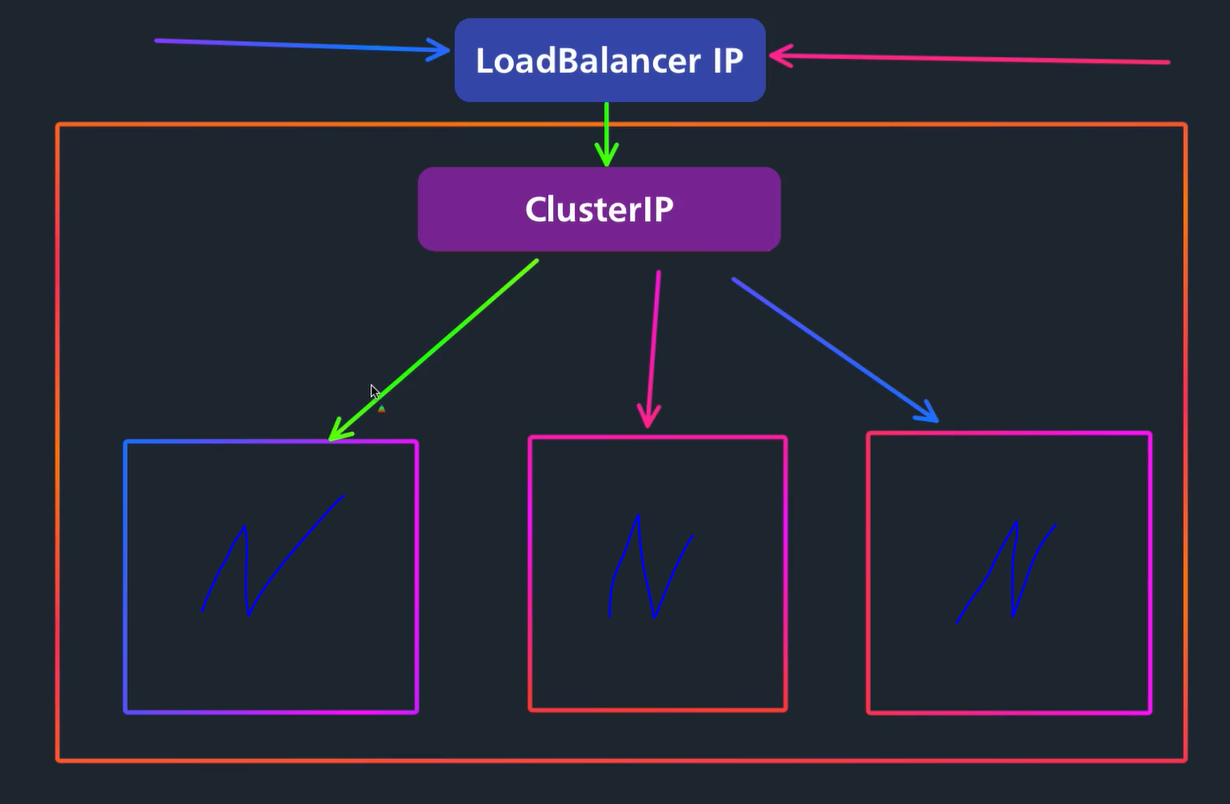
На каждом узле открывается свой порт и через ip его можно подключиться снаружи



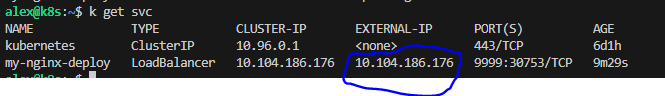
1. Сначала удалим старый сервис kubectl **delete svc my-nginx-deploy**
2. **kubectl expose deploy my-nginx-deploy --type=NodePort --port=8888 --target-port=80** Cоздали сервис с типом NodePort. **deploy my-nginx-deploy**: Указывает, что мы хотим создать сервис для деплоймента (Deployment) с именем my-nginx-deploy.
3. 
4. **minikube ip** узнаем ip кластера
5. Проверяем доступность nginx снаружи **curl 192.168.49.2:31276** Все работает, идет 2 перенаправления – С **192.168.49.2:31276 -> 10.109.235.43:8888 -> pod\_ip:80**
6. **kubectl delete svc my-nginx-deploy**

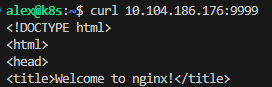
## LoadBalancer IP

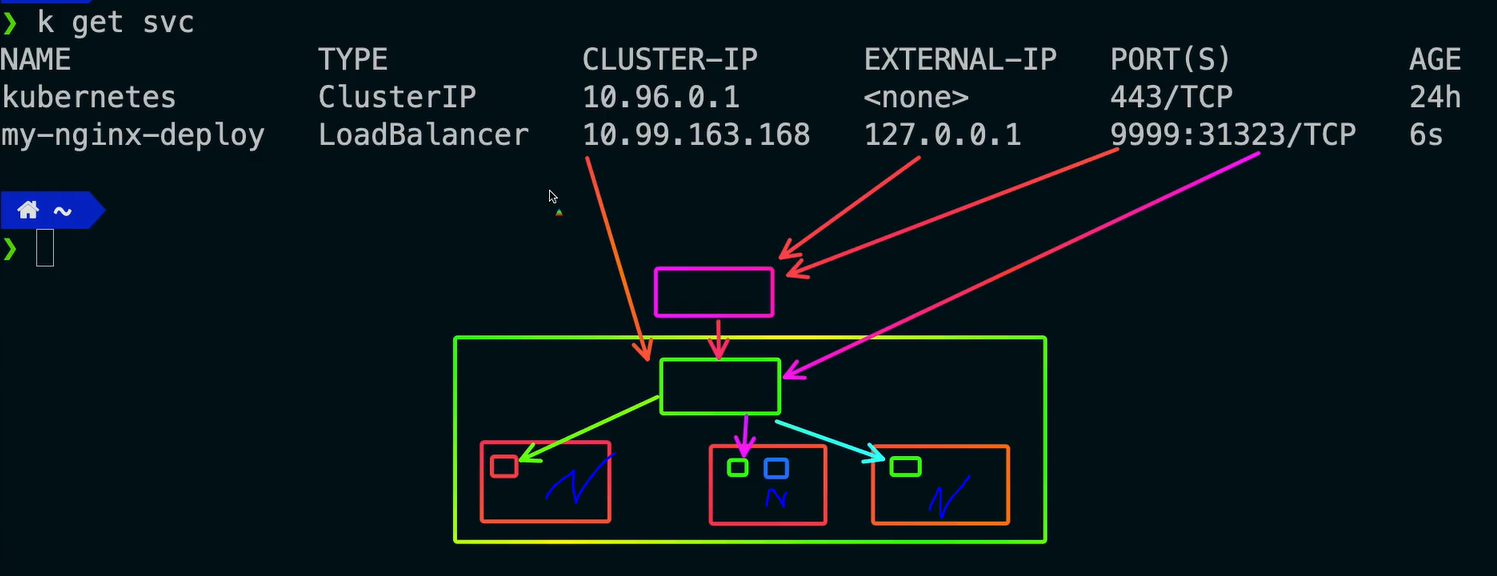
Для всего деплоймента создается один внешний ip через который можно подключаться внутрь кластера

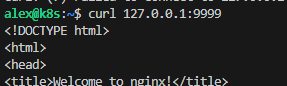


1. **kubectl expose deploy my-nginx-deploy --type=LoadBalancer --port=9999 --target-port=80** 
   1. **deploy my-nginx-deploy**: Указывает, что мы хотим создать сервис для деплоймента (Deployment) с именем my-nginx-deploy.
   2. **--type=LoadBalancer**: Определяет тип сервиса. Тип LoadBalancer создаёт внешний балансировщик нагрузки (Load Balancer) для распределения трафика. Это особенно полезно в облачных средах, таких как AWS, GCP или Azure, где провайдер автоматически создаёт балансировщик нагрузки.
   3. **--port=9999**: Указывает порт, на котором сервис будет доступен снаружи кластера. Внешние запросы будут направляться на этот порт.
   4. **--target-port=80**: Указывает порт, на который сервис будет направлять трафик внутри кластера. Это порт контейнера, на котором приложение (в данном случае Nginx) принимает запросы.
2. Открываем втурую вкладку и набираем **minikube tunnel**, чтобы появился external ip
3. Возвращаемся на первую и **k get svc**   
   Появляется Внешний ip







1. Если external ip не 127.0.0.1, а другой, то можно сделать перенаправление  
   **kubectl port-forward svc/my-nginx-deploy 9999:9999**
2. **delete svc my-nginx-deploy**
3. **k delete deploy my-nginx-deploy**

## Создание образа с нуля

### Установка yarn и node js

### Yarn установка

[https://classic.yarnpkg.com/en/docs/install#debian-stable](https://classic.yarnpkg.com/en/docs/install%23debian-stable)

**curl -sS https://dl.yarnpkg.com/debian/pubkey.gpg | sudo apt-key add -**

**echo "deb https://dl.yarnpkg.com/debian/ stable main" | sudo tee /etc/apt/sources.list.d/yarn.list**

**sudo apt update && sudo apt install yarn**

**sudo apt update && sudo apt install --no-install-recommends yarn** If using nvm you can avoid the node installation by doing

## Nodejs установка

**sudo apt purge nodejs**

**sudo apt install build-essential checkinstall**

**sudo apt install libssl-dev**

**wget -qO- https://raw.githubusercontent.com/nvm-sh/nvm/v0.37.2/install.sh | bash**

Далее открываем новый терминал, чтобы следующая команда сработала

**nvm ls-remote** Затем смотрим список доступных версий Node js: **nvm install 18.20.3**

**nvm list** список установленных версий

## Создание приложения nodejs

<https://github.com/bstashchuk/k8s>

1. **npm init -y**  создадим папку и в ней инициализируем package.json
2. npm install express
3. создаем файл **index.mjs**

import express from 'express'

import os from 'os'

const app = express()

const PORT = 3000

app.get("/", (req, res) => {

  const helloMessage = `<h1>VERSION 2: Hello from the ${os.hostname()}</h1>`

  console.log(helloMessage)

  res.send(helloMessage)

})

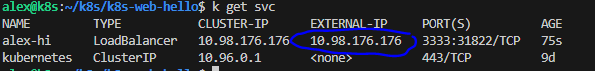
app.listen(PORT, () => {

  console.log(`Web server is listening at port ${PORT}`)

})

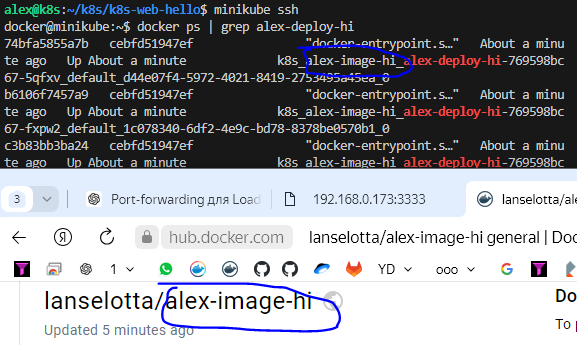
1. **npm start** запускаем и проверяем
2. удаляем папу **Node\_modules**
3. создаем **dockerfile**  
   FROM --platform=linux/amd64 node:alpine  
   WORKDIR /app  
   EXPOSE 3000  
   COPY package.json package-lock.json ./  
   RUN npm install  
   COPY . ./  
   CMD ["npm", "start"]
4. Создаем образ **docker build . -t lanselotta/alex-image-hi:latest -t lanselotta/alex-image-hi:1.0.0**
5. Заливаем на докерхаб. Если впервые то **docker login** потом **docker push lanselotta/alex-image-hi --all-tags**

## Cоздание кластера kubernetes на основе своего образа

1. **k create deploy alex-deploy-hi --image=lanselotta/alex-image-hi:1.0.0**
2. Проверяем. Берем ip адрес пода **k get -o wide**, заходим в кластер **minikube ssh** и курлом проверяем **curl 10.244.0.69:80**
3. **k expose deploy alex-deploy-hi --type=LoadBalancer --port=3333 --target-port=3000** Создаем сервис с типом loadbalancer для деплоя **alex-deploy-hi**
4. **minikube tunnel**  создаем тунель и проверяем что он открылся  
   
5. **kubectl port-forward service/alex-deploy-hi --address=192.168.0.173 3333:3333** Чтобы был доступ по сети с локалки делаем проброс портов для сервиса alex-hi с типом LoadBalancer с сетевого интерфейса 192.168.0.173 с порта 3333 на внешний порт LoadBalancer 3333. Это нужно потому что LoadBalancer изначально работает только на отдельном своем внешнем ip. Стоит учесть что обращение идет только к первому поду. Чтобы увидеть обращение к другим, нужно использовать lynx на убунте сервер – к примеру **lynx 10.98.176.176:3333**
6. **k scale deploy alex-deploy-hi --replicas=7** увеличим число реплик и проверим балансировку **lynx 10.98.176.176:3333** Обновление страницы ctrl+R
7. Изменяем файл index.mjs и создаем образ новый   
   **docker build . -t lanselotta/alex-image-hi:latest -t lanselotta/alex-image-hi:2.0.0**
8. И заливаем на dockerhub - **docker push lanselotta/alex-image-hi --all-tags**

## Обновление деплоя deploy

Имя используемого образа в названии контейнера внутри контейнера кластера. minikube ssh, docker ps | grep alex-hi. alex-hi это имя деплоя



1. **k set image deploy alex-deploy-hi alex-image-hi=alex-image-hi:2.0.0  
   alex-image-hi** имя образа которое мы используем в контейнерах существующих
2. **kubectl rollout status deploy alex-deploy-hi**

## Создание деплоймента и сервиса с помощью yaml

Инфа как создавать ресурсы

<https://kubernetes.io/docs/reference/kubernetes-api/workload-resources/>

1. **kubectl delete --all all** Удалим все предыдущие сервисы
2. Создаем файл deployment.yaml
3. В vs code в файле начинаем печатать deployment и должен написаться весь шаблон. Вписываем имя, образ контейнера, порт для документации, т.к. не влияет по сути, ограничение памяти и процессора.
4. **kubectl apply -f deployment.yaml** Все запустилось и работает и доступно и curl отрабатывает
5. Увеличим число подов в реплике <https://kubernetes.io/docs/reference/kubernetes-api/workload-resources/deployment-v1/> Видим по документации что replicaset пишется на уровне selector и template

apiVersion: apps/v1

kind: Deployment

metadata:

  name: alex-deploy-yml

spec:

  replicas: 5

  selector:

    matchLabels:

      app: alex-deploy-yml

  template:

    metadata:

      labels:

        app: alex-deploy-yml

    spec:

      containers:

      - name: alex-deploy-yml

        image: lanselotta/alex-image-hi:1.0.0

        resources:

          limits:

            memory: "128Mi"

            cpu: "250m"

        ports:

        - containerPort: 3000

1. Применяем изменения **kubectl apply -f deployment.yaml**

Создаем сервис

1. Создаем файл service.yaml
2. Начинаем писать service и применяем шаблон
3. Прописываем все необходимое

apiVersion: v1

kind: Service

metadata:

  name: alex-deploy-yml

spec:

  type: LoadBalancer

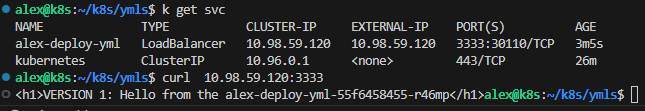
  selector:

    app: alex-deploy-yml

  ports:

  - port: 3333

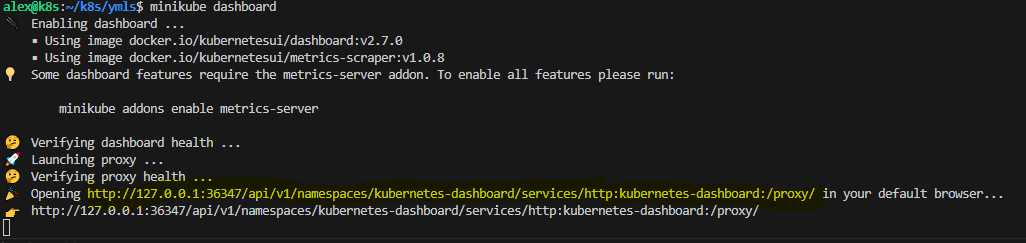
    targetPort: 3000

1. Применяем **kubectl apply -f service.yaml**
2. На новом терминале создаем туннель **minikube tunnel**
3. На первой вкладке смотрим ip сервиса **k get svc**
4. Проверяем curl’ом через внешний порт **curl 10.98.59.120:3333**   
   
5. **kubectl delete -f deployment.yaml -f service.yaml** Удаляем сервисы и деплойменты

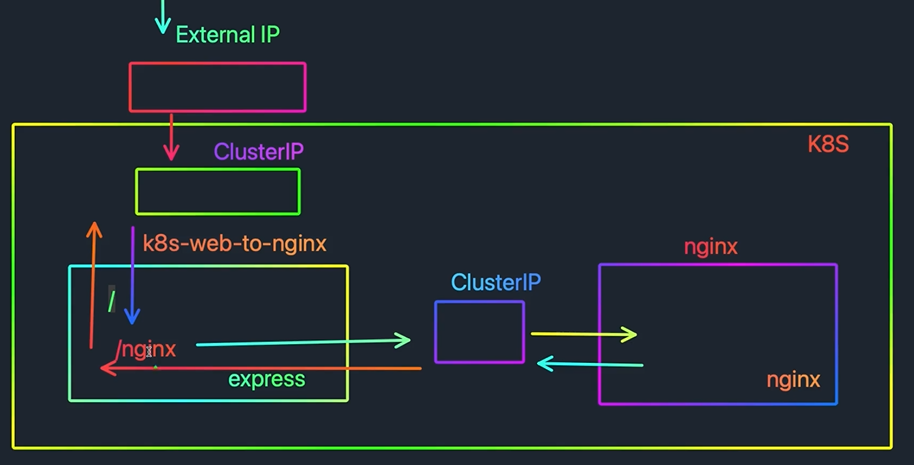
## Dashboard

minikube dashboard

Открываем ссылку в браузере в винде и все открывается



## Создаем 2 деплоймента



Берем файлы из hello версии предыдущей и изменяем index.mjs

import express from 'express'

import os from 'os'

const app = express()

const PORT = 3000

app.get("/", (req, res) => {

  const helloMessage = `<h1>VERSION 2: Hello from the ${os.hostname()}</h1>`

  console.log(helloMessage)

  res.send(helloMessage)

})

app.get("/nginx", async (req, res) => {

  const url = 'http://nginx'

  const response = await fetch(url);

  const body = await response.text();

  res.send(body)

})

app.listen(PORT, () => {

  console.log(`Web server is listening at port ${PORT}`)

})

В рамках кластера кубернетес можно взаимодействовать с сервисами по их именам

Создадим nginx.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

  name: nginx

spec:

  selector:

    matchLabels:

      app: nginx

  template:

    metadata:

      labels:

        app: nginx

    spec:

      containers:

      - name: nginx

        image: nginx

        resources:

          limits:

            memory: "128Mi"

            cpu: "125m"

        ports:

        - containerPort: 80

---

apiVersion: v1

kind: Service

metadata:

  name: nginx

spec:

  selector:

    app: nginx

  ports:

  - port: 80

    targetPort: 80

и to-nginx.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

  name: to-nginx

spec:

  selector:

    matchLabels:

      app: to-nginx

  template:

    metadata:

      labels:

        app: to-nginx

    spec:

      containers:

      - name: to-nginx

        image: lanselotta/web-to-nginx-image

        resources:

          limits:

            memory: "128Mi"

            cpu: "250m"

        ports:

        - containerPort: 3000

---

apiVersion: v1

kind: Service

metadata:

  name: to-nginx

spec:

  type: LoadBalancer

  selector:

    app: to-nginx

  ports:

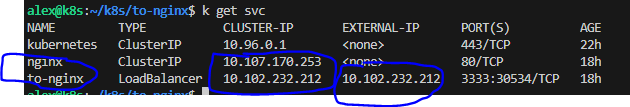
  - port: 3333

    targetPort: 3000

Запускаем туннель **minikube tunnel**

**kubectl apply -f nginx.yaml -f to-nginx.yaml**

Доступность и по ip и по имени



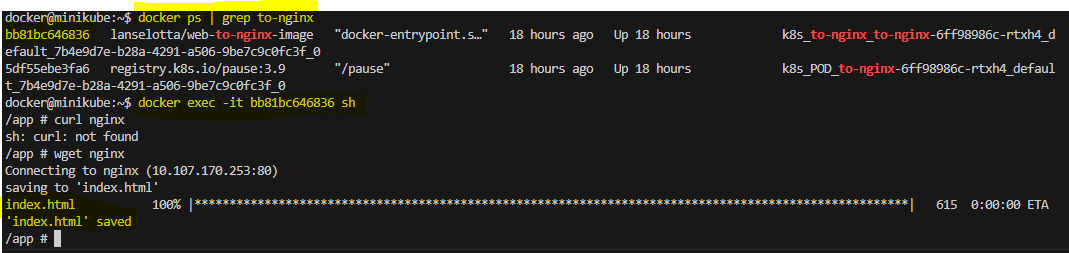
Внутри кластера из узла по имени недоступно

**minikube ssh**

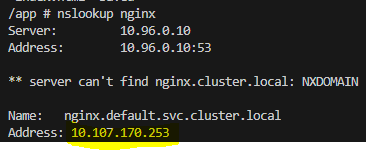
**curl nginx**

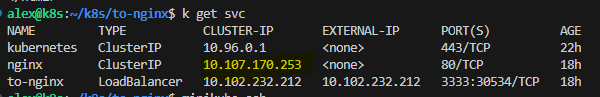


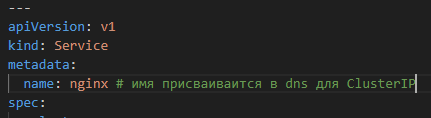
Заходим внутрь контейнера to-nginx. По curl не доступен, но по wget да. Т.е. изнутри контейнера имя nginx доступно, а изнутри кластера нет



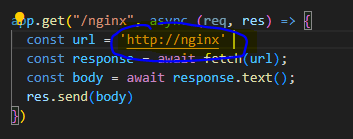
Запрос к dns серверу подтверждает







Обращаемся по имени:

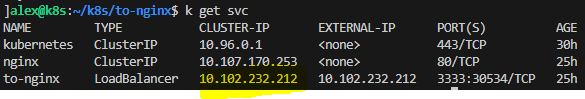


## Домашнее Задание

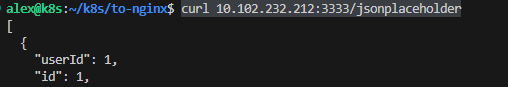
Добавляем новый тег для образа **docker tag lanselotta/web-to-nginx-image:latest lanselotta/web-to-nginx-image:1.0.0**

И пушим все тэги **docker push lanselotta/web-to-nginx-image --all-tags**

1. Меняем приложение
2. **docker build . -t lanselotta/web-to-nginx-image:latest -t lanselotta/web-to-nginx-image:2.0.0**
3. **docker push lanselotta/web-to-nginx-image --all-tags**
4. Меняем yaml файлы, меняем image с нужным тэгом
5. **k apply -f nginx.yaml -f to-nginx.yaml**



1. **curl 10.102.** **232.212:3333/jsonplaceholder**
2. Все работает



1. Cнова изменим приложение

app.get("/jsonplaceholder", async (req, res) => {

  const url = 'https://jsonplaceholder.typicode.com/posts'

  const response = await fetch(url);

  const body = await response.text();

  res.setHeader("Content-Type", "application/json")

  res.send(body)

})

1. Повторим шаги создания образа, пуш в доккерхаб, изменим версию образа в to-nginx.yaml
2. **kubectl port-forward service/to-nginx --address=192.168.0.173 3333:3333** В отдельном терминале сделаем проброс с локалки по сети
3. [**http://192.168.0.173:3333/jsonplaceholder**](http://192.168.0.173:3333/jsonplaceholder) **в браузере и все работает**
4. **k delete -f nginx.yaml -f to-nginx.yaml**  удаляем