KARPOV.COURSES >>>
КОНСПЕКТ

> Конспект > 5 урок > DE и Kubernetes, часть 1

> Оглавление

ConfigMaps
Secrets
Env
Value from Volume
CronJob

> Введение и основные концепции Kubernetes

Путь к Kubernetes. Монолит.

- Жесткие зависимости затрудняли разработку и делали ее медленнее
- Отсутствие гибкости, из-за чего приложение было тяжело изменять
- Тяжело скалируется: если запросы выросли только к одному модулю, приходилось скалировать приложение целиком
- Медленные релизы и фичи

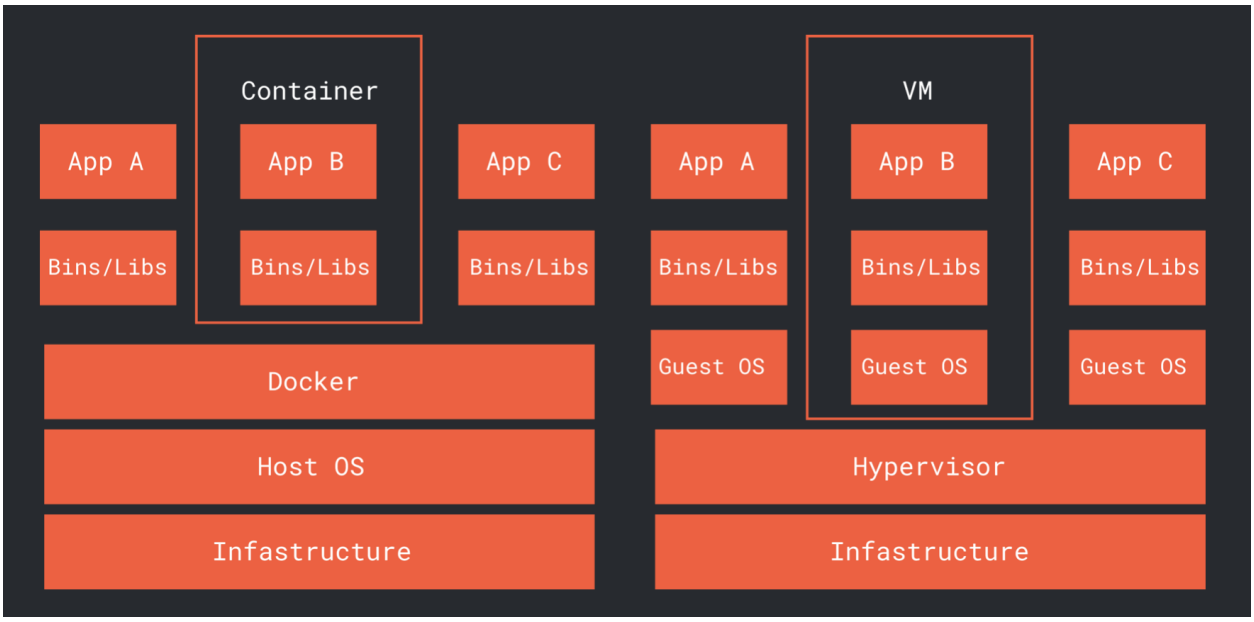
Микросервисы

Ответом на вызовы, которые создавала разработка приложений в монолитной архитектуре стала архитектура, делающая акцент на микросервисы.

Что нам это дало?

- Гибкость, так как можно было выбирать разные фреймворки/языки/инструменты
- Легко скалируются, потому что каждый модуль можно скалировать отдельно
- Быстрые релизы

Контейнеры и ВМ



Когда мы говорим про микросервисы, предполагается, что каждый из микросервисов мы будем упаковывать в контейнеры. Но изоляция окружений возникла до контейнеров: первоначально для этих целей использовались виртуальные машины (ВМ).

Зачем тогда понадобились контейнеры, если существовала такая концепция, как ВМ? Причина в том что, контейнеры более легковесные и с ними удобнее работать. Как видно на слайде, каждая ВМ включает в себя уровень операционной системы (Guest OS). Иными словами, каждая ВМ - мини-сервер с отдельной ОС. На обслуживание ВМ тратится много ресурсов. В контейнерах слой с ОС всего один (Host OS) и существует на уровне инфраструктуры. Поверх этого слоя работает Docker и оболочка Docker позволяет запускать внутри себя отдельные контейнеры.

Однако, контейнеры гораздо менее изолированы, чем ВМ. При запуске на одном сервере нескольких докер-контейнеров, из одного можно "пролезть" в другой. Об этом следует помнить в целях поддержки безопасности системы.

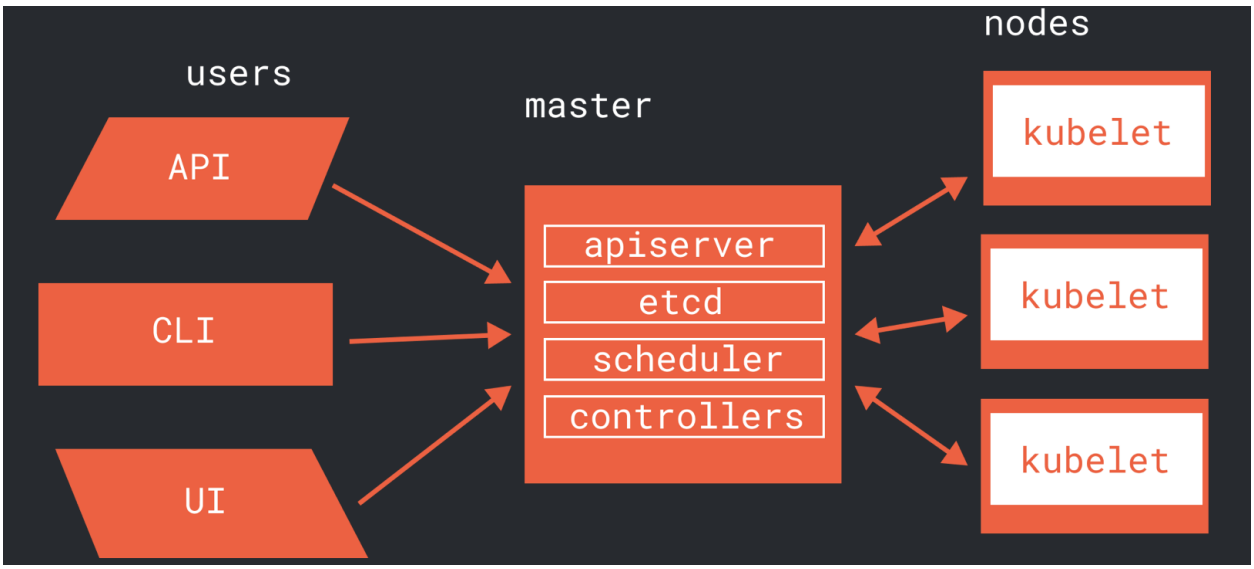
Kubernetes

Мы начали запускать все в контейнерах, и возник вопрос, как мы теперь будем управлять сотнями контейнеров? Приложение может состоять из тысяч функций, которые упакованы в свой докер-контейнер. Мы должны настроить между ними сетевые правила, следить за тем, чтобы контейнер был живым. Это достаточно

большой скоп задач, их можно решать, написав свою оболочку для управления всеми контейнерами, но проще всего использовать Kubernetes. Что он умеет?

- Оркестрация контейнезированных приложений
- Управление жизненным циклом приложений
- Организация инфраструктуры для работы с приложениями

Основные концепции Kubernetes. At large scale.



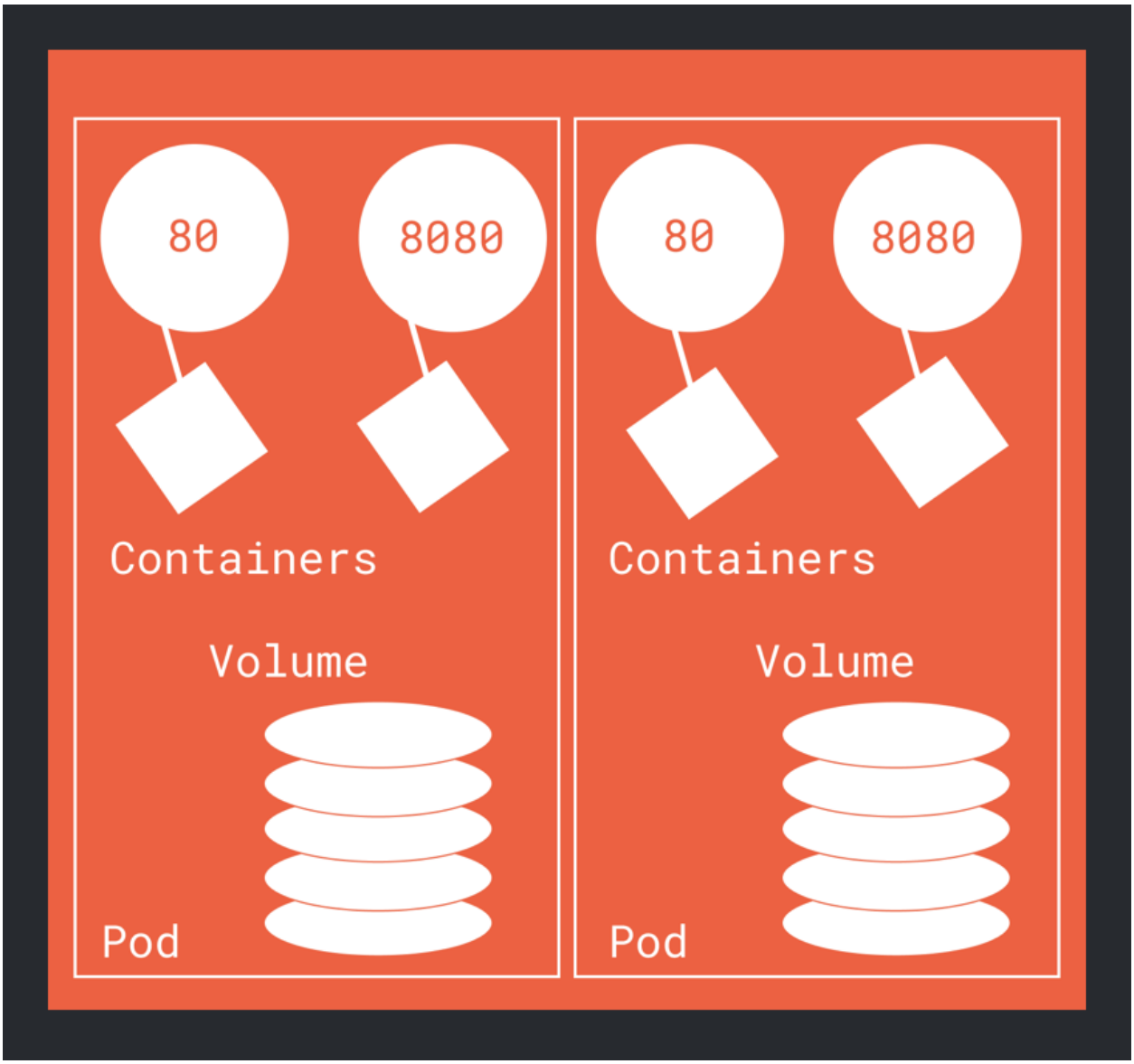
Архитектура Kubernetes

Kubernetes - кластерная система, позволяющая внутри себя запускать и управлять приложениями.

- Центральная часть - master node
- Рабочая нагрузка запускается на worker node (справа на схеме). На них может работать, к примеру, Spark.
- Пользователи - users (слева на схеме). Взаимодействие происходит путем API/CLI/UI.

> Основные абстракции Kubernetes для DE

Pod



- Атомарный объект или наименьший юнит работы в Kubernetes
- Pod-ы состоят из одного или более контейнеров, которые делят вольюмы и сети

Когда мы запускаем контейнер внутри Kubernetes, мы запускаем его, используя абстракцию Pod. Если внутри Pod будет несколько контейнеров, то они будут делить между собой диск и сетевые настройки.

Pod описывается через YAML:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-example
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx
```

- kind - описывает абстракцию, которую мы запускаем
- metadata - указывается имя (name) и уровни (labels)
- spec - указываются контейнеры, которые будут запускаться; в данном примере будет запущен один контейнер, который будет использовать образ nginx (image) с именем контейнера nginx (name)

Важно знать, что место на диске, которое доступно Pod'у для работы - не персистентно. Это означает, что если Pod перезапустится, что является нормальной ситуацией, все данные, которые были внутри Pod'а пропадут. Таким образом, их нельзя хранить внутри Pod'а.

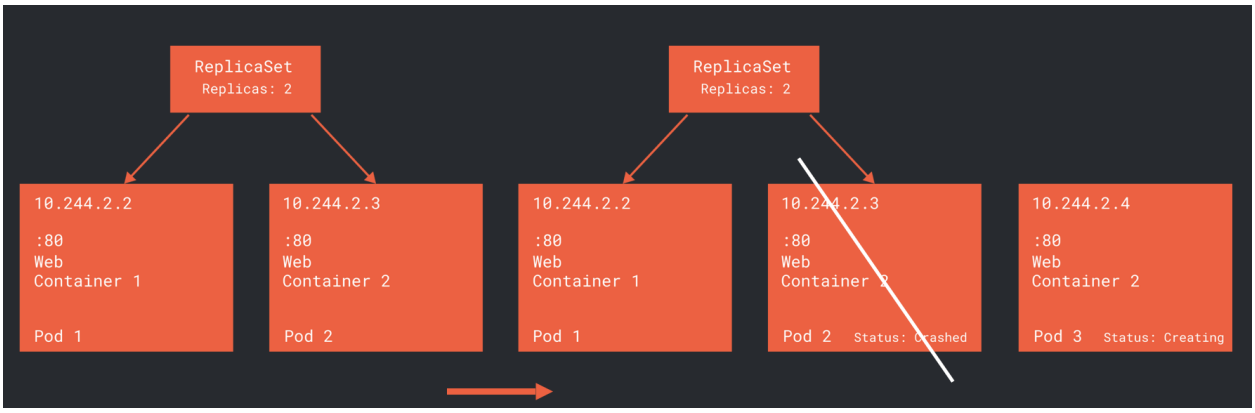
- Ресурсы одного pod-а деплоятся вместе
- Pod-ы не используются для хранения данных (эфемерны)

ReplicaSet

ReplicaSet - абстракция более высокого порядка, чем Pod.

- Основной метод управления репликами Pod и их жизненным циклом
- Обеспечивает необходимое количество запущенных реплик

Pod'ы не запускаются напрямую. YAML, который был приведен выше в качестве примера, можно использовать для отладки, но обычно приложения не запускаются таким образом, так как если контейнер, запущенный внутри Pod, "умер", Kubernetes ничего с этим делать не будет. Для перезапуска приложений/Pod'ов, управления числом реплик Pod'ов используется ReplicaSet.



ReplicaSet следит за тем, чтобы обеспечивать нужное количество Pod'ов в живом состоянии. Предположим, что нам нужно поддерживать 2 Pod'а в ReplicaSet в запущенном состоянии. Если один "умрет", ReplicaSet автоматически запустит новый Pod с той же самой спецификацией, которая была описана в YAML.

Как выглядит YAML для ReplicaSet:

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: rs-example
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
      env: prod
  template:
    <pod template>
```

- replicas: требуемое количество экземпляров Pod
- selector: определяет все Pod'ы, управляемые этим ReplicaSet

Выполнив команду `$ kubectl get pods` (взаимодействие с Kubernetes посредством интерфейса командной строки), мы увидим, что запущено 3 Pod'а, с именем `rs-example-xxxxx` (xxxxx - случайно сгенерированная часть имени).

```
$ kubectl get pods
NAME               READY     STATUS    RESTARTS   AGE
rs-example-9l4dt   1/1       Running   0          1h
```

```
rs-example-b7bcg   1/1       Running   0          1h
rs-example-mkll2   1/1       Running   0          1h
```

Deployment

Deployment - абстракция еще более высокого порядка, основной контроллер. При размещении приложения/рабочей нагрузки, чаще всего работа происходит именно с Deployment.

- Основной контроллер для управления Pods
- Управляют ReplicaSet
- Предоставляют возможность управления обновлениями и функциональность rollback'a.

YAML Deployment выглядит следующим образом:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deploy-example
spec:
  replicas: 3
  revisionHistoryLimit: 3
  selector:
    matchLabels:
      app: nginx
      env: prod
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
  template:
    <pod template>
```

- strategy: описывает метод обновления Pods на основе type
- recreate: все существующие поды убиваются до запуска новых
- rollingUpdate: циклическое обновление Pods на основе maxSurge и maxUnavailable
- maxSurge: определяет количество дополнительных реплик

- maxUnavailable: количество возможно недоступных реплик
- revisionHistoryLimit: сколько историй обновлений будет храниться в памяти
- template: задается шаблон Pod'a

Посмотрим, как это выглядит на практике. Предположим, мы создаем новую версию приложения. Запускаем RollingUpdate, используя абстракцию Deployment и ее методы.

В настоящий момент запущен ReplicaSet1, в котором есть 3 Pods.

```
$ kubectl get replicaset
NAME                         DESIRED   CURRENT   READY     AGE
mydep-6766777fff             3         3         3         5h
```

В момент обновления создается ReplicaSet2, в котором будет 0 Pods. На основе параметра maxSurge (со значением 1) создается новый Pod с новой версией контейнера.

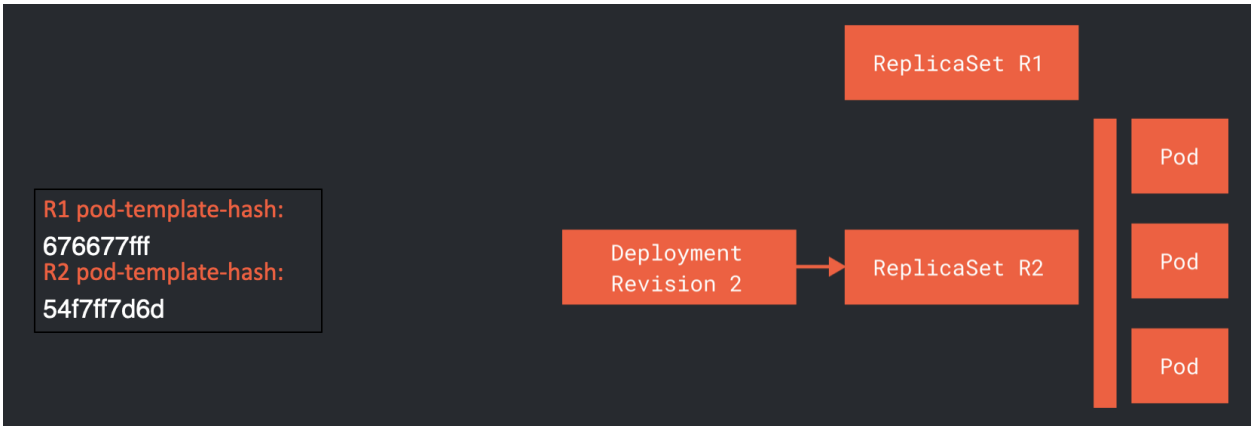
```
$ kubectl get replicaset
NAME                        DESIRED   CURRENT   READY     AGE
mydep-54f7ff7d6d            1         1         1         5s
mydep-6766777fff            2         3         3         5h
```

В настоящий момент мы имеем 4 Pod'a нашего приложения.

```
$ kubectl get pods
NAME                       READY     STATUS    RESTARTS   AGE
mydep-54f7ff7d6d-9gvll     1/1       Running   0          2s
mydep-6766777fff-9r2zn     1/1       Running   0          5h
mydep-6766777fff-hsfz9     1/1       Running   0          5h
mydep-6766777fff-sjxhf     1/1       Running   0          5h
```

Один из них будет убит из старой версии ReplicaSet1, будет добавлен второй в раздел ReplicaSet2. И так поочередно все Pods из ReplicaSet1 будут убиты и созданы в ReplicaSet2.

Итоговый результат будет выглядеть так:



ReplicaSet2 теперь имеет 3 Pods, ReplicaSet1 - 0 Pods.

Deployment помогает нам в случае необходимости откатиться на предыдущую версию быстро и легко. Предположим, что мы создали 3 новых Pods с новыми версиями приложения, но в приложении обнаружили баг.

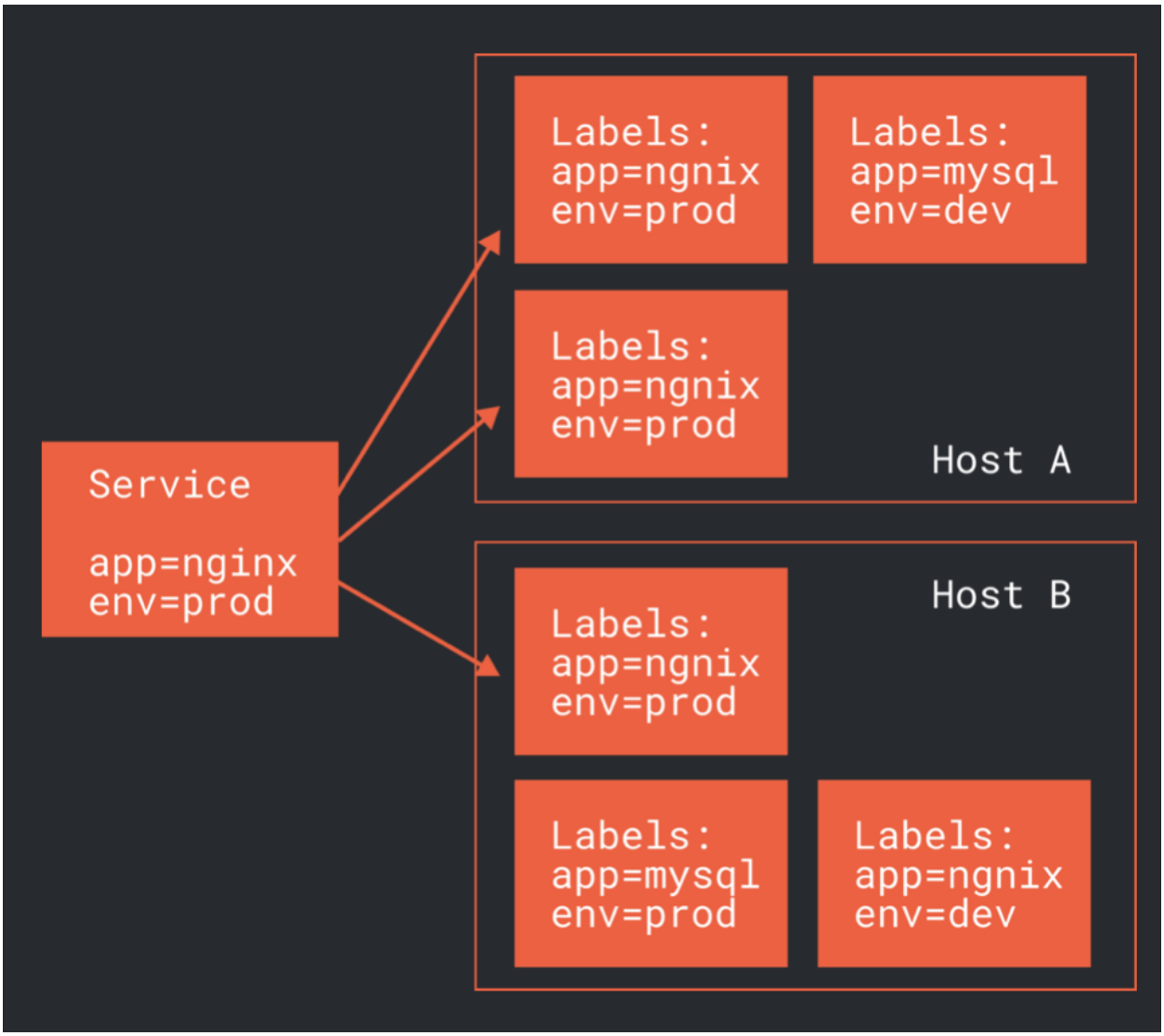
Deployment предполагает функциональность rollback'a. Мы можем явно задать на какую версию нам необходимо откатиться. Здесь помогает параметр revisionHistoryLimit, который задает количество хранимых

версий. Deployment переключится на предыдущий ReplicaSet1 и точно так же пересоздаст Pods.

> Сетевые абстракции Kubernetes

Services

Services - абстракция, задающая правила сетевого доступа к Pods, описывается посредством YAML.



- Универсальный метод доступа к приложениям в Pods

- Внутренний балансировщик для Pods
- Имеет DNS имя, привязанное к namespace

Services используются для взаимодействия приложения с внешней средой, решают задачу пропуска трафика в приложение.

Существуют 3 типа Services:

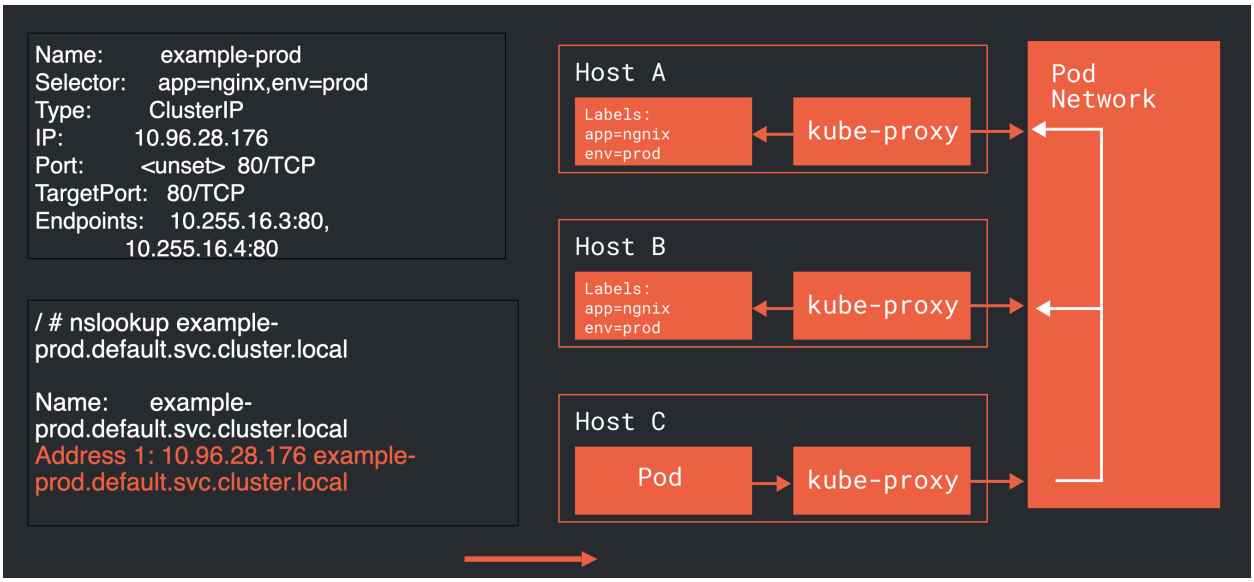
- ClusterIP
- NodePort
- Loadbalancer

ClusterIP

Описывается с помощью YAML:

```
apiVersion: v1
kind: Service
metadata:
  name: example-prod
spec:
  selector:
    app: nginx
    env: prod
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```

ClusterIP открывает доступ только по внутреннему виртуальному IP, доступному только во внутренней сети кластера. Чаще всего ClusterIP используют для взаимодействия приложений, запущенных внутри Kubernetes.



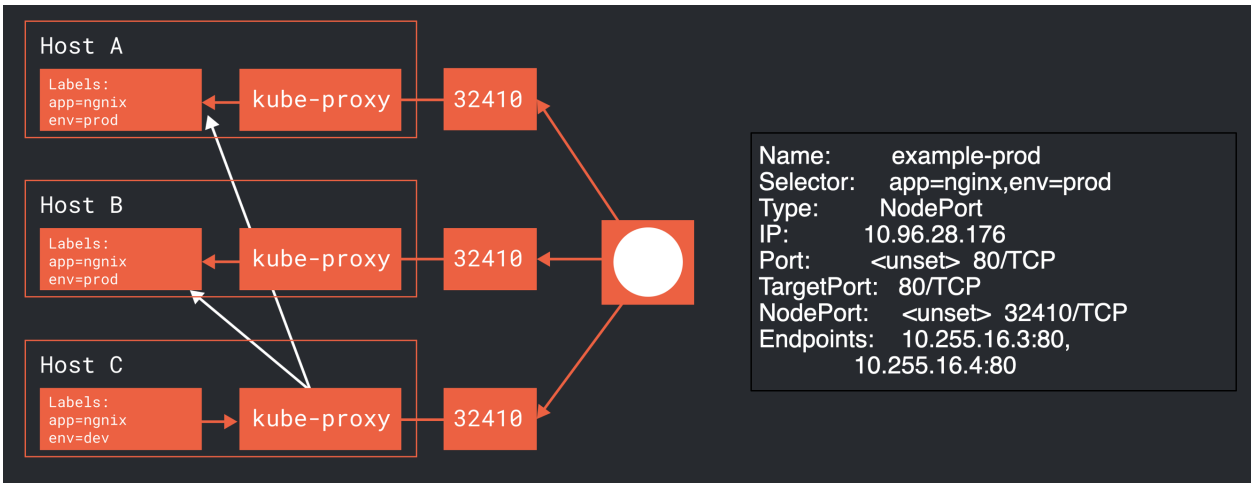
Worker nodes A/B/C (Host A/B/C) имеют kube-proxy, который отвечает за настройку сетевых правил.

NodePort

NodePort открывает во внешний доступ один из портов (выбранный явно или случайно из пула 30000-32767) на каждой node. В работе дата инженера используется редко (или вообще не используется).

YAML:

```
apiVersion: v1
kind: Service
metadata:
  name: example-prod
spec:
  type: NodePort
  selector:
    app: nginx
    env: prod
  ports:
  - nodePort: 32410
    protocol: TCP
    port: 80
    targetPort: 80
```



Используется несколько наборов воркеров (A/B/C), на которых запущены Pods, доступ к которым можно будет получить, обратившись к порту, который был задан при создании.

Loadbalancer

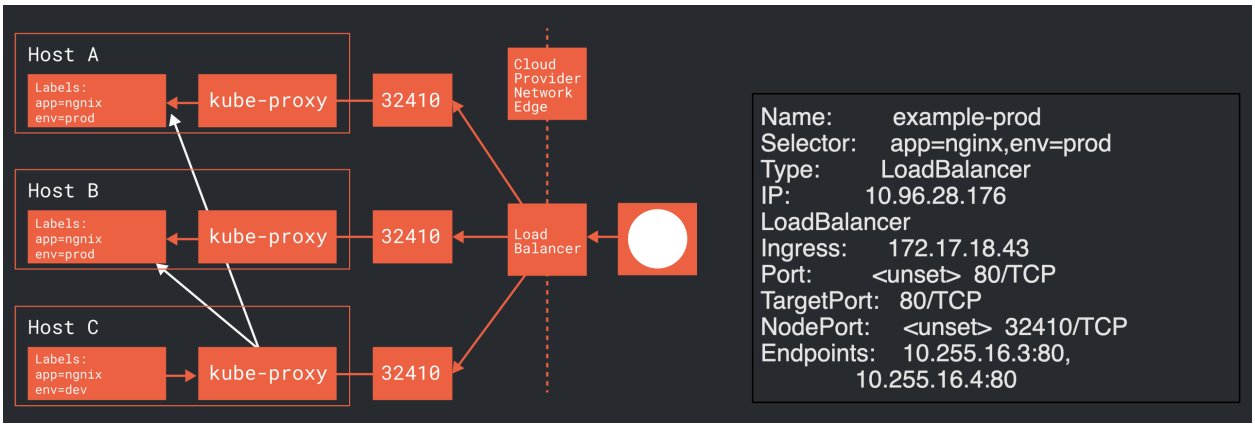
Loadbalancer - "правильный" способ открыть доступ к приложению из внешней сети.

YAML:

```
apiVersion: v1
kind: Service
metadata:
  name: example-prod
spec:
  type: LoadBalancer
  selector:
    app: nginx
    env: prod
  ports:
    protocol: TCP
    port: 80
    targetPort: 80
```

Для того, чтобы Loadbalancer функционировал, он должен интегрироваться с неким внешним Loadbalancer, который предоставляет облако или среда, в которой запущен Kubernetes. Если работа будет производиться с Kubernetes as a service в облаках, то вам не нужно будет думать о том, как реализован Loadbalancer и кто будет его предоставлять. Требуется лишь создать подобный YAML и облако

самостоятельно выделит ВМ нужного размера, настроит правила, предоставит Loadbalancer для доступа к вашим приложениям.

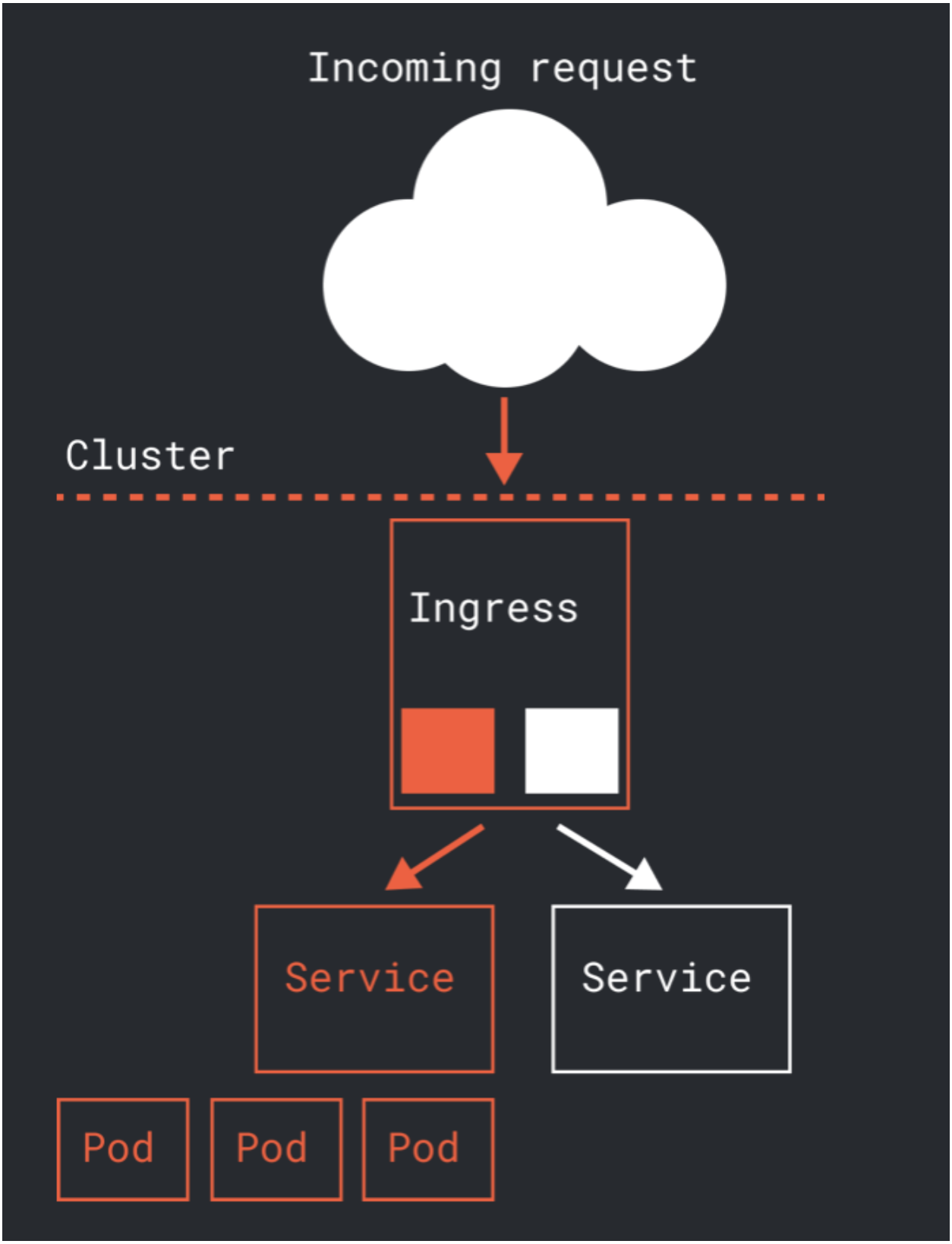


Loadbalancer расположен на границе между внутренней сетью Kubernetes и внешней сетью, весь трафик идет через него.

Ingress

Ingress используется для того, чтобы запускать внешний трафик к приложениям, созданным внутри Kubernetes. В отличие от Loadbalancer, который способен работать только с одним приложением (или частью одного большого), Ingress предоставляет доступ сразу к нескольким приложениям с единого IP. Он так же предоставляется автоматически облачным провайдером.

- Используется как внешний gateway для доступа к приложению извне
- Поддерживает L7 policy
- IngressController – аналог nginx
- Ingress – аналог vhost


Incoming request
Cluster
Ingress
Service
Service
Pod
Pod
Pod

> Управление окружением, параметрами и переменные в Kubernetes

ConfigMaps

При работе с приложениями в Kubernetes могут возникнуть задачи сохранить какие-либо настройки. При наличии одного докер-контейнера, его можно запускать с разными параметрами или внутрь этого докер-контейнера пробрасывать разные переменные окружения, на основе которых он будет вести себя по-разному.

В качестве примера предположим, что у нас есть Spark приложение, упакованное в докер-контейнер, которое принимает на вход параметр с путем к данным (например, имя S3 бакета). Spark приложение использует его, чтобы прочитать данные и далее будет использована еще одна переменная - имя бакента, куда данные будут записаны после обработки. Для того, чтобы не "хардкодить" значения с именами бакетов, мы создадим Spark приложение, упакуем его в контейнер и оно будет принимать на вход эти 2 параметра входных и выходных данных. Для этого и используется следующая абстракция - ConfigMap.

ConfigMaps представляет собой хранилище параметров типа key/value (параметр data).

YAML:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: manifest-example
data:
  state: Michigan
  city: Ann Arbor
  content: |
    Look at this,
    its multiline!
```

Secrets

Secrets - хранят данные в зашифрованном виде.

- Кодируют данные в base64

- Могут быть зашифрованы в etcd
- Используются для хранения паролей, сертификатов и других чувствительных данных

YAML:

```
apiVersion: v1
kind: Secret
metadata:
  name: manifest-secret
type: Opaque
data:
  username: ZXhhbXBsZQ==
  password: bXlwYXNzd29yZA==
```

Env

ConfigMaps и Secrets нужны нам были для того, чтобы задавать переменные окружения.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: cm-env-example
spec:
  template:
    spec:
      containers:
      - name: mypod
        image: alpine:latest
        command: [“/bin/sh”, “-c”]
        args: [“printenv CITY”]
        env:
        - name: CITY
          valueFrom:
            configMapKeyRef:
              name: manifest-example
              key: city
      restartPolicy: Never
```

В данном YAML мы видим пример абстракции Job, которая предоставляет функциональность разового запуска некой рабочей контейнезированной нагрузки внутри Kubernetes. Данный контейнер выведет переменную окружения CITY при

своем запуске. Возьмет он ее из ConfigMap, который был создан ранее. Команда `printenv CITY` выведет значение данных, хранящихся по ключу CITY внутри ConfigMap.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: secret-env-example
spec:
  template:
    spec:
      containers:
      - name: mypod
        image: alpine:latest
        command: [“/bin/sh”, “-c”]
        args: [“printenv USERNAME”]
        env:
        - name: USERNAME
          valueFrom:
            secretKeyRef:
              name: manifest-example
              key: username
      restartPolicy: Never
```

Аналогичным образом работает и абстракция Secret в данном YAML. Команда `printenv USERNAME` выведет переменную окружения USERNAME, которая будет взята из Secret с именем secret-env-example.

Value from Volume

Можно задавать переменные окружения через подключения Volume.

Контейнер будет примонтирован по адресу `/mysecret` и в нем будут храниться данные из нашего Secret. Чтобы получить доступ, запущенный контейнер должен будет сходить в директорию `/mysecret`, внутри которой расположены файлы. Каждый файл имеет имя, которое будет совпадать со значениеми ключа внутри Secret.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: secret-vol-example
spec:
  template:
```

```
    spec:
      containers:
      - name: mypod
        image: alpine:latest
        command: [“/bin/sh”, “-c”]
        args: [“cat /mysecret/username”]
        volumeMounts:
        - name: secret-volume
          mountPath: /mysecret
      restartPolicy: Never
      volumes:
      - name: secret-volume
        secret:
          secretName: manifest-example
```

CronJob

CronJob предназначен для запуска контейнеров по расписанию.

- Расширение Job, для выполнения конкретного расписания
- schedule: cron-расписание

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: cronjob-example
spec:
  schedule: "*/1 * * * *"
  successfulJobsHistoryLimit: 3
  failedJobsHistoryLimit: 1
  jobTemplate:
    spec:
      completions: 4
      parallelism: 2
      template:
        <pod template>
```

Best practice для дата инженера использовать, например, Airflow, который будет запускать по расписанию внутри Kubernetes приложения, вместо CronJob.