GeekBrains

Урок 3. Kubernetes. Часть 2. Работа в реальном кластере

На этом уроке

1. Освоим работу в удаленном кластере, построенном на принципах проектирования инфраструктуры под продакшн
2. Познакомимся с паттернами разработки контейнеризируемых приложений
3. Поговорим о жизненном цикле приложений в Kubernetes-кластере
4. Познакомимся со стратегиями развертывания приложений

Оглавление

Доступ к учебному кластеру

В прошлом уроке мы практиковались с упрощенным локальным Kubernetes-кластером, развернутым с помощью инструмента minikube. В этом уроке мы перейдём к работе в “настоящем” кластере, развернутом на базе Mail Cloud Solutions.

В реальных проектах в рамках ежедневной работы к одному и тому же кластеру, как правило, имеют доступ разные команды и разработчики. Мы смоделируем именно эту ситуацию.

Каждый студент получает **индивидуальные реквизиты** для доступа к кластеру. Эти реквизиты позволят вам запускать собственные приложения и получать информацию о состоянии кластера. Реквизиты запрещено передавать третьим лицам. Также запрещено использовать кластер с целью нанесения вреда другим лицам.

Кластер будет доступен до конца текущего модуля, им можно пользоваться для выполнения домашних заданий и самостоятельной практики.

Для доступа к кластеру сохраните полученный вами yaml-файл с реквизитами доступа на локальный компьютер. В переменной окружения **KUBECONFIG** укажите путь доступа к файлу. Например, если вы используете bash, в файл ~/.bashrc добавьте следующую строку:

```
export KUBECONFIG=~/kubeconf/geekbrains.yaml
```

И не забудьте вызвать команду для применения изменений в рамках текущей сессии:

```
source ~/.bashrc
```

После выполнения вышеуказанных действий kubectl будет настроен для работы с удаленным кластером. Для возврата в предыдущее состояние, удалите переменную окружения **KUBECONFIG**.

Теперь можно проверить параметры доступа к кластеру. Первая команда выводит информацию о конфигурации, вторая отдает список нод кластера:

```
kubectl config view
kubectl get nodes
```

Убедитесь, что конфиг настроен на работу с кластером geekbrains, и вам доступно получение списка нод кластера.

Развертывание собственных приложений в учебный кластер

Одним из популярных подходов развертывания микросервисов в кластере является вариант, когда отдельные микросервисы развёртывают в отдельные **пространства имён** (неймспейсы, namespace). В прошлом уроке мы уже поговорили про такие ресурсы и абстракции Kubernetes, как сервис, деплоймент, под и ингресс. Неймспейс - это ещё один ресурс Kubernetes. Часто неймспейсы используются для группировки ресурсов (сервисов, подов, реплика-сетов). Например, с помощью неймспейсов удобно разграничивать права доступа к кластеру, чтобы у каждой продуктовой команды был доступ только к своим сервисам. В учебном кластере у нас будет похожая история - у каждого студента есть доступ только к управлению ресурсами своего неймспейса.

Важно заметить, что неймспейсы разделяют кластер именно на **логические** группы, физически поды разных неймспейсов могут быть как на одних и тех же, так и на разных нодах.

В рамках заданий этого урока вы можете использовать собственный неймспейс для практики со своими приложениями - например, после развертывания попробуйте удалить сервис или отдельные поды и посмотрите, какие события это повлечет.

Сервер статики

В качестве примера Go-приложения в этом уроке рассмотрим простенький веб-сервер, который умеет отдавать готовые статические HTML-страницы. Этого примера будет достаточно для того, чтобы охватить все изучаемые паттерны и подходы. Вы также можете попрактиковаться с любыми вашими ранее написанными приложениями.

Для реализации сервера воспользуемся функцией FileServer из стандартной библиотеки http. Например, если мы хотим отдавать статический контент (картинки, готовые html-страницы и т. д.) из директории ./static, а сервер запущен на порте 8080, код может выглядеть примерно так:

```
package main

import (
  "log"
  "net/http"
)

func main() {
  fs := http.FileServer(http.Dir("./static"))
  http.Handle("/", fs)

  err := http.ListenAndServe(":8080", nil)
```

```
  if err != nil {
     log.Fatal(err)
  }
}
```

Из прошлого урока мы уже знаем, что конфигурацию удобно задавать через переменные окружения. Сделаем параметры “расположение статических файлов” и “порт для запуска сервера” конфигурируемыми:

```
package main

import (
  "log"
  "net/http"

  "github.com/kelseyhightower/envconfig"
)

// Config задает параметры конфигурации приложения
type Config struct {
  Port        string `envconfig:"PORT" default:"8080"`
  StaticsPath string `envconfig:"STATICS_PATH" default:"./static"`
}

func main() {
  config := new(Config)
  err := envconfig.Process("", config)
  if err != nil {
     log.Fatalf("Can't process config: %v", err)
  }

  fs := http.FileServer(http.Dir(config.StaticsPath))
  http.Handle("/", fs)

  err = http.ListenAndServe(":"+config.Port, nil)
  if err != nil {
     log.Fatalf("Error while serving: %v", err)
  }
}
```

Поскольку для конфигурации мы использовали стороннюю библиотеку envconfig, зададим управление зависимостями через gomod:

```
go mod init statics
go mod tidy
go mod download
```

Подготовим Docker-образ для запуска сервиса. Для того, чтобы в приложении был смысл, нам нужна хоть какая-то статика, поэтому на одном из этапов задания контейнера создадим файл index.html с текстом “Hello, World!” в качестве примера того, что может отдавать реализуемый нами сервер:

```
FROM golang:1.15 as modules

ADD go.mod go.sum /m/
RUN cd /m && go mod download

FROM golang:1.15 as builder

COPY --from=modules /go/pkg /go/pkg

RUN mkdir -p /src
ADD . /src
WORKDIR /src

RUN useradd -u 10001 myapp

RUN GOOS=linux GOARCH=amd64 CGO_ENABLED=0 \
  go build -o /myapp ./

# Готовим пробный файл статики
RUN mkdir -p /test_static && touch /test_static/index.html
RUN echo "Hello, world!" > /test_static/index.html

FROM busybox

ENV PORT 8080
ENV STATICS_PATH /test_static

COPY --from=builder /test_static /test_static

COPY --from=builder /etc/passwd /etc/passwd
USER myapp

COPY --from=builder /myapp /myapp
COPY --from=builder /etc/ssl/certs/ /etc/ssl/certs/

CMD ["/myapp"]
```

Проверим, что всё работает. Подготовим и запустим контейнер. Вместо myuser используйте ваш аккаунт от DockerHub:

```
docker build -f Dockerfile -t myuser/statics .
docker push myuser/statics
docker run -p8080:8080 myuser/statics
```

Теперь можно сделать запрос к http://127.0.0.1:8080, и мы должны увидеть строку “Hello, world!”

Контейнер готов.

Добавим минимальный набор манифестов для того, чтобы запустить приложение в Kubernetes. В описании образа укажите путь к вашему DockerHub. В данном случае мы зададим манифесты для деплоймента и сервиса, их можно описать в одном файле:

Файл manifests.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: statics
 labels:
   app: statics
spec:
 replicas: 3
 strategy:
   type: RollingUpdate
   rollingUpdate:
     maxUnavailable: 50%
     maxSurge: 1
 template:
   metadata:
     labels:
       app: statics
   spec:
     containers:
       - name: statics
         image: docker.io/webdeva/statics:v1
         imagePullPolicy: Always
         ports:
           - containerPort: 8080
         livenessProbe:
           httpGet:
             path: /
             port: 8080
         readinessProbe:
           httpGet:
             path: /
             port: 8080
         resources:
           limits:
             cpu: 2m
             memory: 10Mi
           requests:
             cpu: 2m
             memory: 10Mi
 selector:
```

```
   matchLabels:
     app: statics
---
apiVersion: v1
kind: Service
metadata:
 name: statics
 labels:
   app: statics
spec:
 ports:
   - port: 8080
     targetPort: 8080
     protocol: TCP
     name: http
 selector:
   app: statics
```

Теперь отправим манифест в кластер и убедимся в том, что развертывание прошло успешно. Обратите внимание на параметр **-n** (неймспейс). Обязательно задайте здесь тот неймспейс, к которому у вас есть доступ.

```
$ kubectl -n myuser apply -f manifests.yaml
deployment.apps/statics created
service/statics created

$ kubectl -n myuser get deployments
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
statics   3/3     3            3           5m17s

$ kubectl -n myuser get service
NAME      TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
statics   ClusterIP   10.254.137.17   <none>        8080/TCP   5m23s

$ kubectl -n myuser get pods
NAME                       READY   STATUS    RESTARTS   AGE
statics-5bfd7df6b-g9ln5    1/1     Running   0          5m27s
statics-5bfd7df6b-ms8hv    1/1     Running   0          5m27s
statics-5bfd7df6b-xhh2f    1/1     Running   0          5m27s
```

Для того, чтобы получить доступ к сервису с локального компьютера, воспользуемся механизмом port-forward. В этом примере мы перенаправляем деплоймент с порта 8080 в кластере на порт 8000 в локальном окружении.

```
$ kubectl -n myuser port-forward deployment/statics 8000:8080
Forwarding from 127.0.0.1:8000 -> 8080
Forwarding from [::1]:8000 -> 8080
```

Теперь, делая запрос к http://127.0.0.1:8000, мы попадем на приложение, запущенное в удаленном кластере.

Попробуйте удалить и снова развернуть приложение. Запускайте команды kubectl, изученные в прошлом уроке (logs, events и другие), чтобы понаблюдать за поведением кластера.

Взаимодействие сервисов внутри одного кластера

DNS-сервер - одна из важных составляющих Kubernetes-кластера. С помощью DNS можно осуществить механизм обнаружения сервисов (service discovery). Каждый сервис, развернутый в Kubernetes, получает собственный DNS-адрес в формате **<service-name>.<namespace-name>.svc.cluster.local**. Пусть, например, в неймспейсе **default** у нас развернут сервис **hello**, доступный на порте 8080. DNS-адрес этого сервиса будет выглядеть как **hello.default.svc.cluster.local**. Если мы захотим сделать запрос к этому сервису из сервиса world в неймспейсе logic того же кластера, мы можем пренебречь суффиксом svc.cluster.local, и, например, базовый URL для http-запросов будет выглядеть как **http://hello.default:8080**. Более того, если сервис world тоже запущен в неймспейсе default, то и суффиксом-неймспейсом тоже можно принебречь, и URL превращается в **http://hello:8080.**

Помимо DNS, Kubernetes также предоставляет механизм обнаружения сервисов **через переменные окружения**. В момент запуска пода, в его переменные окружения прописывается информация обо всех сервисах, которые существуют в текущий момент. Эти переменные окружения задаются в формате **<SERVICE-NAME>_SERVICE_HOST** и **<SERVICE-NAME>_SERVICE_PORT**. Для примера из предыдущего абзаца мы можем получить HELLO_SERVICE_HOST=10.109.72.32 и HELLO_SERVICE_PORT=8080.

Давайте посмотрим, как работает обнаружение сервисов на практике. В своём неймспейсе запустите под, в котором есть контейнер с утилитами командной строки, и посмотрите на переменные окружения, доступные в этом контейнере:

1. Создаём под для экспериментов. В результате запуска этой команды мы окажемся внутри контейнера на основе образа busybox с наличием curl:

   ```
   kubectl -n myuser run curl --image=radial/busyboxplus:curl -i --tty --rm
   ```

2. Смотрим на доступные переменные окружения, для этого вызываем команду printenv. Среди списка переменных окружения найдем хост и порт сервиса statics из предыдущей части урока.

```
$ printenv

KUBERNETES_SERVICE_PORT=443
KUBERNETES_PORT=tcp://10.254.0.1:443
HOSTNAME=curl
SHLVL=1
HOME=/root
PS1=\[\033[40m\]\[\033[34m\][ \[\033[33m\]\u@\H:\[\033[32m\]\w\[\033[34m\]
]$\[\033[0m\]
ENV=/root/.bashrc
STATICS_SERVICE_PORT_HTTP=8080
STATICS_PORT_8080_TCP_ADDR=10.254.137.17
STATICS_SERVICE_HOST=10.254.137.17
TERM=xterm
KUBERNETES_PORT_443_TCP_ADDR=10.254.0.1
STATICS_PORT_8080_TCP_PORT=8080
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
STATICS_PORT_8080_TCP_PROTO=tcp
KUBERNETES_PORT_443_TCP_PORT=443
KUBERNETES_PORT_443_TCP_PROTO=tcp
LS_COLORS=di=34:ln=35:so=32:pi=33:ex=1;40:bd=34;40:cd=34;40:su=0;40:sg=0;40:tw=
0;40:ow=0;40:
STATICS_SERVICE_PORT=8080
STATICS_PORT=tcp://10.254.137.17:8080
STATICS_PORT_8080_TCP=tcp://10.254.137.17:8080
KUBERNETES_PORT_443_TCP=tcp://10.254.0.1:443
KUBERNETES_SERVICE_PORT_HTTPS=443
CLICOLOR=1
PWD=/
KUBERNETES_SERVICE_HOST=10.254.0.1
```

3. Попробуем из текущего пода (с curl) вызвать сервис statics двумя способами: через значения переменных окружения и с помощью DNS:

```
[ root@curl:/ ]$ curl -i http://10.254.137.17:8080
HTTP/1.1 200 OK
Accept-Ranges: bytes
Content-Length: 14
Content-Type: text/html; charset=utf-8
Last-Modified: Sun, 15 Nov 2020 19:43:56 GMT
Date: Sun, 15 Nov 2020 22:18:02 GMT
```

```
Hello, world!

[ root@curl:/ ]$ curl -i
http://${STATICS_SERVICE_HOST}:${STATICS_SERVICE_PORT}/
HTTP/1.1 200 OK
Accept-Ranges: bytes
Content-Length: 14
Content-Type: text/html; charset=utf-8
Last-Modified: Sun, 15 Nov 2020 19:43:56 GMT
Date: Sun, 15 Nov 2020 22:21:21 GMT

Hello, world!


[ root@curl:/ ]$ curl -i http://statics:8080
HTTP/1.1 200 OK
Accept-Ranges: bytes
Content-Length: 14
Content-Type: text/html; charset=utf-8
Last-Modified: Sun, 15 Nov 2020 19:43:56 GMT
Date: Sun, 15 Nov 2020 22:18:09 GMT

Hello, world!

[ root@curl:/ ]$ curl -i http://statics.myuser:8080
HTTP/1.1 200 OK
Accept-Ranges: bytes
Content-Length: 14
Content-Type: text/html; charset=utf-8
Last-Modified: Sun, 15 Nov 2020 19:43:56 GMT
Date: Sun, 15 Nov 2020 22:18:13 GMT

Hello, world!
```

Аналогичным образом может быть построено взаимодействие любых сервисов в кластере.

Паттерны распределенных приложений

В прошлых уроках мы рассматривали особенности докеризации Go-приложений, вспоминали основы работы с Kubernetes и практиковались с развертыванием собственных приложений в Minikube. В этих примерах мы рассматривали простые приложения и простые развертывания, когда в одном kubernetes-поде у нас был один докер-контейнер. Но вы уже знаете, что под может содержать в себе несколько контейнеров. Чтобы понять, в каких случаях это необходимо, и как это работает, рассмотрим популярные паттерны распределенных приложений подробнее. Здесь мы поговорим о

том, какие проблемы решают эти паттерны, и какие варианты их реализации возможны в Kubernetes-кластере.

Контейнеры инициализации (Init Containers)

Часто при запуске сложных приложений отдельное внимание приходится уделять процессу инициализации. Например, при запуске приложения мы считываем конфигурацию и выполняем подключение к ресурсам. В некоторых случаях перед запуском контейнера необходимо прогреть кеш, или выполнить ещё какую-то одноразовую операцию, которую мы не хотим включать в основной образ контейнера приложения. Если мы хотим следовать “принципу единственной ответственности”, для реализации каждой маленькой задачи нужно использовать своё приложение (свой контейнер). На помощь в реализации этого подхода придёт паттерн “**Init Containers**”.

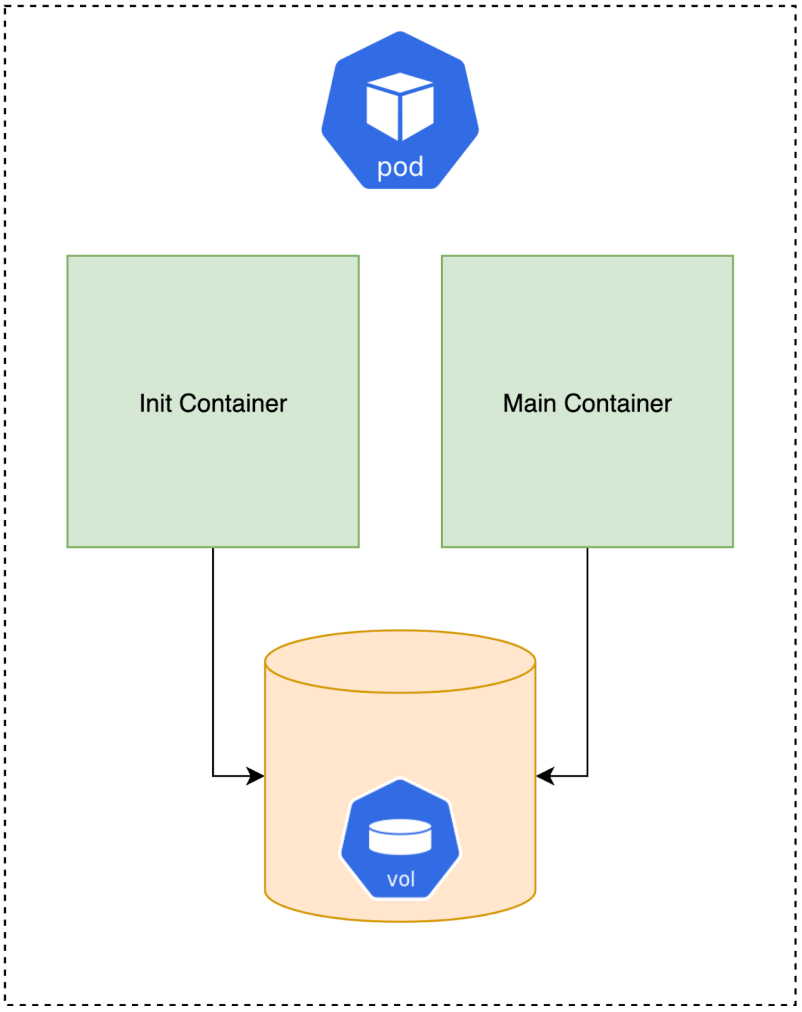
Контейнеры инициализации запускаются до старта контейнеров приложения и должны быть реализованы в виде небольших быстро отрабатывающих приложений. В случае, если такой контейнер отработал успешно, произойдет запуск контейнеров приложения. В случае провала контейнеры приложения запущены не будут, и весь *под* будет перезапущен,что, в свою очередь, снова вызовет запуск контейнеров инициализации.

Контейнеры инициализации работают по тем же принципам, что и обычные, но для них не предусмотрены readiness-пробы, так как эти контейнеры должны быть остановлены сразу после отработки своих задач.

Посмотрим на типичный манифест, который содержит контейнеры инициализации. По аналогии с тем, как мы описываем основные контейнеры в блоке “containers”, контейнеры инициализации должны быть описаны в блоке “initContainers”. В этом примере мы запускаем основной контейнер и два контейнера инициализации:

```
apiVersion: v1
kind: Pod
metadata:
  name: trying-initcont-pod
  labels:
    app: trying-initcont
spec:
  containers:
  - name: myapp-container
    image: busybox:latest
   command: ['sh', '-c', 'Running the main container && sleep 7200']
  initContainers:
  - name: init-myservice
    image: busybox:latest
    command: ['sh', '-c', 'Running init container 1']
  - name: init-myservice
    image: busybox:latest
    command: ['sh', '-c', 'Running init container 2']
```

В качестве более практического примера доработаем сервис отдачи статики, рассмотренный ранее. При создании образа контейнера мы генерировали “пробный” файл index.html. Давайте теперь представим, что при запуске контейнера в кластере нам нужно добавить подготовленную заранее статику. Решение такой задачи как раз можно возложить на контейнер инициализации. В качестве примера статического сайта возьмём https://github.com/mdn/beginner-html-site-scripted. В рамках работы контейнера инициализации будем клонировать этот репозиторий. Для того, чтобы основной контейнер приложения смог получить доступ к файлам, которые мы склонировали в контейнер инициализации, нужно будет добавить volume и сделать его общим для обоих контейнеров:



Изменим манифест деплоймента, чтобы достичь такого результата:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: statics
 labels:
   app: statics
spec:
 replicas: 3
```

```yaml
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxUnavailable: 50%
    maxSurge: 1
template:
  metadata:
    labels:
      app: statics
  spec:
    volumes:
      - name: src
        emptyDir: { }
    initContainers:
      - name: prepare-statics
        image: alpine/git
        command:
        - git
        - clone
        - https://github.com/mdn/beginner-html-site-scripted
        - /static
        volumeMounts:
        - mountPath: /static
          name: src
    containers:
      - name: statics
        image: docker.io/webdeva/statics:v1
        imagePullPolicy: Always
        volumeMounts:
          - mountPath: /static
            name: src
        env:
          - name: PORT
            value: "8080"
          - name: STATICS_PATH
            value: "/static"
        ports:
          - containerPort: 8080
        livenessProbe:
          httpGet:
            path: /
            port: 8080
        readinessProbe:
          httpGet:
            path: /
            port: 8080
        resources:
          limits:
            cpu: 2m
            memory: 10Mi
          requests:
            cpu: 2m
            memory: 10Mi
```

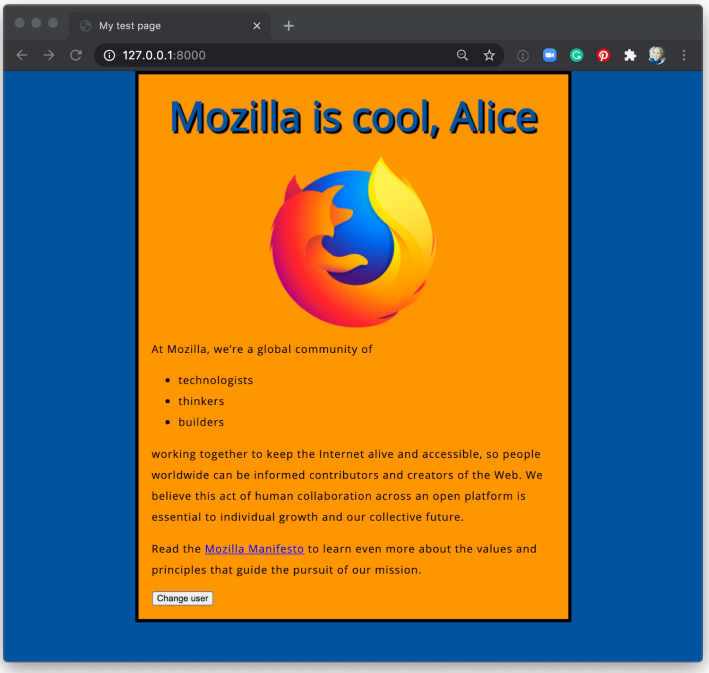
```
selector:
  matchLabels:
    app: statics
```

Обратите внимание на то, что для основного контейнера мы перезаписываем переменную окружения **STATICS_PATH,** чтобы приложение определило, откуда читать новую статику.

Снова запускаем port-forward, чтобы сервис был доступен с локального окружения:

```
$ kubectl -n myuser port-forward deployment/statics 8000:8080
Forwarding from 127.0.0.1:8000 -> 8080
Forwarding from [::1]:8000 -> 8080
```

Откроем в браузере http://127.0.0.1:8000. Если всё было сделано правильно, вы увидите приглашение ввести свое имя и затем такой пример сайта:



Сайдкар (Sidecar)

Сайдкар - это контейнер, запущенный в дополнение к главному контейнеру с приложением. В отличие от контейнеров инициализации, сайдкары могут оставаться запущенными в течение всего периода жизни приложения “параллельно” основному контейнеру. Как и контейнеры инициализации, сайдкары позволяют реализовать “принцип единственной ответственности”. Например, мы можем вынести в сайдкары реализацию вспомогательных процессов.

Если бы в предыдущем примере с контейнером инициализации мы хотели бы уметь обновлять статику без перезапуска основного контейнера, на помощь пришел бы именно сайдкар.

Более конкретными примерами реализации паттерна сайдкар являются паттерны **адаптер** и **амбассадор**, мы рассмотрим их далее.

Адаптер (Adapter)

Паттерн **“Адаптер”** используется для унификации интерфейса доступа к главному контейнеру пода извне. Например, если мы хотим ограничить доступ других сервисов кластера к текущему, нам понадобится поставить мини-роутер перед текущим сервисом. На уровне этого роутера будут производиться проверки возможности доступа. Приложение A, обратившееся к текущему сервису S, сначала попадет на роутер. Если конфигурация запрещает делать запросы из приложения A, оно получит информацию о том, что доступ ограничен. Если же доступ для приложения A настроен, роутер перенаправит запрос в сервис. Для реализации такого роутера нам и потребуется сайдкар типа “Адаптер”. Также мы можем использовать “Адаптер”, когда хотим преобразовать данные, полученные от основного контейнера приложения, в какой-то специализированный формат или сделать еще какой-то дополнительный процессинг.

Амбассадор (Ambassador)

Паттерн “**Амбассадор”** является противоположным паттерну “Адаптер”. Представьте, что теперь нам нужно реализовать прокси, который будет решать, как запросы должны строиться изнутри пода наружу. Например, мы хотим определить свой способ для обнаружения сервисов или хотим реализовать преобразование запросов из одного формата в другой. Здесь на помощь придёт сайдкар типа “Амбассадор”.

Управление жизненным циклом пода

Жизненный цикл приложений, запущенных в контейнерах, как правило, управляется самой инфраструктурой. Не исключением является и Kubernetes. В предыдущем уроке мы уже познакомились с понятием проб и хелсчеков - проверок, которые выполняются в течение всего жизненного цикла подов. Давайте теперь подробнее поговорим о таких важных этапах жизненного цикла приложений, как запуск и остановка.

Graceful shutdown

Kubernetes решает остановить контейнер, если под нужно завершить, или если liveness-проба провалена. В этом случае контейнер получит сигнал SIGTERM. Как только этот сигнал получен,

приложение должно его обработать и завершиться. Одни приложения могут завершиться сразу, другие должны сначала обработать текущий запрос и фоновые задачи. Такая элегантная остановка называется “**Graceful Shutdown**”.

Если после получения сигнала SIGTERM контейнер не завершил работу через механизм Graceful Shutdown, он будет остановлен принудительно после получения сигнала SIGKILL. Время периода, через который отправляется SIGKILL можно настроить в поле **terminationGracePeriodSeconds** спецификации пода. Однако, это время может быть перезаписано, если под нужно остановить вручную незамедлительно.

Посмотрим на реализацию Graceful Shutdown на примере http-сервера. Нам понадобится:

- Перехватить сигналы операционной системы с помощью специального канала на основе os.Signal и функции signal.Notify
- Вызывать метод Shutdown для http-сервера

Перепишем сервер раздачи статики так, чтобы он был запущен в отдельной горутине. Главная горутина приложения будет слушать сигналы операционной системы, и при получении сигнала SIGTERM или SIGINT будет завершать сервер:

```
package main

import (
  "log"
  "net/http"
  "os"
  "os/signal"
  "syscall"

  "github.com/kelseyhightower/envconfig"
)

// Config задает параметры конфигурации приложения
type Config struct {
  Port        string `envconfig:"PORT" default:"8080"`
  StaticsPath string `envconfig:"STATICS_PATH" default:"./static"`
}

func main() {
  config := new(Config)
  err := envconfig.Process("", config)
  if err != nil {
     log.Fatalf("Can't process config: %v", err)
  }

  fs := http.FileServer(http.Dir(config.StaticsPath))
  http.Handle("/", fs)

  go func() {
```

```
    err = http.ListenAndServe(":"+config.Port, nil)
    if err != nil {
       log.Fatalf("Error while serving: %v", err)
    }
 }()

 interrupt := make(chan os.Signal, 1)
 signal.Notify(interrupt, os.Interrupt, syscall.SIGTERM)

 select {
 case killSignal := <-interrupt:
    switch killSignal {
    case os.Interrupt:
       log.Print("Got SIGINT...")
    case syscall.SIGTERM:
       log.Print("Got SIGTERM...")
    }
 }

}
```

Post Start

Мы уже знакомы с механизмом контейнеров инициализации, который позволяет провести вспомогательные действия до запуска основного контейнера приложения. В случае, если такие вспомогательные действия нужно провести сразу же после запуска основного контейнера, можно воспользоваться механизмом postStart-хук. Команда, описанная в postStart, будет выполнена сразу же, как только будет достигнута ENTRYPOINT контейнера.

Pre Stop

По аналогии с postStart существует также механизм preStop. Этот хук запускается на этапе завершения контейнера, когда контейнером получен сигнал SIGTERM. Как правило, и postStart, и preStop используются для реализации некритичных действий по подготовке или очистке контейнера. Примеры использования postStart и preHook можно найти здесь.

Стратегии развертывания в продакшн

Процессы развертывания и отката приложений - пожалуй, самые сложные среди всех операций, которые производятся над приложениями. Мы хотим релизить новые версии приложений без простоев для пользователей, а значит, нам нужно обеспечить такой механизм, который позволит всегда иметь стабильную запущенную версию приложения, на которую отправляются запросы

пользователей. Тем не менее, во время развертывания новой версии или отката на предыдущую, всегда есть вероятность, что что-то может пойти не так: например, если в новой версии обнаружен баг или новый API оказался несовместим с другими сервисами. Для того, чтобы минимизировать риск возникновения таких проблем, но при этом не составлять сложных планов по развертыванию приложений, можно воспользоваться одной из безопасных стратегий развертывания. Именно о них мы поговорим в этой части урока.

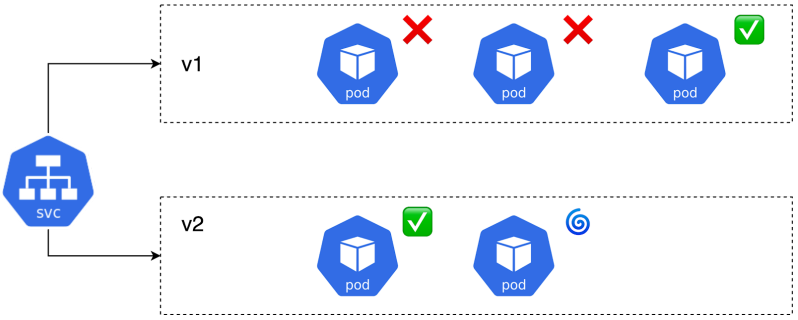
Из предыдущего урока нам понадобятся понятия deployment, strategy, replicas (replica set), selector.

Rolling Update

В прошлом уроке мы уже научились писать простые манифесты для развертывания приложений. При этом мы использовали стратегию **Rolling Update**, которая позволяет производить развертывания без простоя. В случае использования этой стратегии Kubernetes создает новый Replica Set с новыми версиями контейнеров, и по мере прохождения новыми контейнерами readiness-пробы, старые версии удаляются. Пример манифестов, описывающих такие развертывания, можно посмотреть в предыдущем уроке.

Важно!

Минусом подхода Rolling Update является то, что при его использовании у нас одновременно запущены и старая, и новая версии приложения. Чтобы при таком подходе не возникло проблем или сбоев, критически важно поддерживать обратную совместимость между новой и старой версиями приложения. Это касается как публичных API, так и внутренней логики работы приложения (например, при взаимодействии с полями таблиц в базах данных).



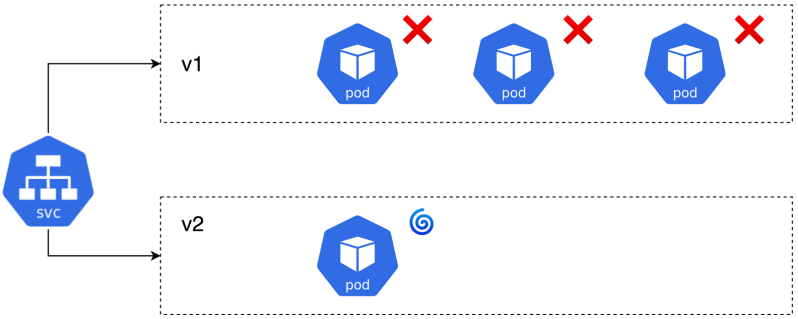
На схеме показан процесс Rolling Update:

- Два пода старой версии v1 уже остановлены (❌), один по-прежнему запущен (✔️)
- Уже запущен (✔️) один под новой версии v2, и еще один находится в процессе запуска (🌀)

По завершении процесса Rolling Update, все три пода старой версии будут остановлены, а три пода новой версии будут в запущенном состоянии.

Fixed Deployment

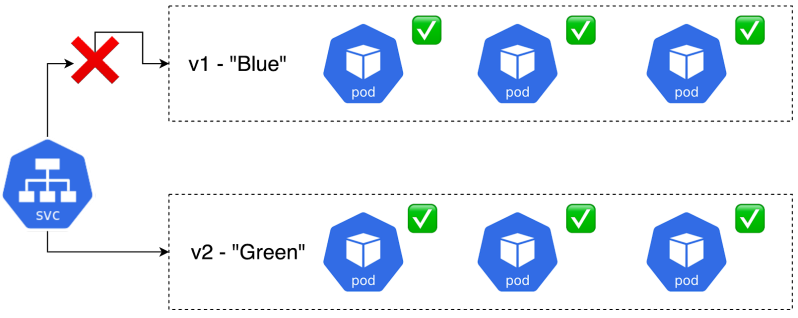
В случае, когда есть причина, по которой нельзя держать одновременно разные версии приложения, можно выполнить фиксированное развертывание (**Fixed Deployment**). Для этого достаточно выставить значение **maxUnavailalbe** таким же, как и значение **replicas**.



На схеме продемонстрирована ситуация, когда все три пода старой версии остановлены, и под новой версии находится в процессе запуска. При этом наблюдается простой - приложение в данный момент недоступно, так как нет ни одного пода в запущенном состоянии.

Blue-Green Deployment

В случае, если до момента, когда приложение становится доступным для пользователей, на продкашн-среде нужно провести дополнительное тестирование, можно использовать **Blue-Green Deployment -** сине-зелёные развертывания. При использовании этой стратегии продакшн-окружение делится на две среды: синюю и зелёную. В любой момент времени трафик пользователей направляется только на одну из сред - активную, например, зелёную. Во время проведения развертывания, новый код доставляется на неактивную (синюю) среду, где затем проводится тестирование. Когда мы уверены в том, что приложение на синей среде стабильно, мы можем перенаправить трафик пользователей на неё. Таким образом, неактивной становится уже зелёная среда. У такого подхода есть дополнительный плюс: в случае, если на синей среде будет зафиксирован сбой, мы всегда можем перенаправить трафик на зелёную стабильную среду. Таким образом, процесс отката при таком подходе максимально прост.



Посмотрим, как реализация таких “сред” может выглядеть в случае использования Kubernetes-кластера.

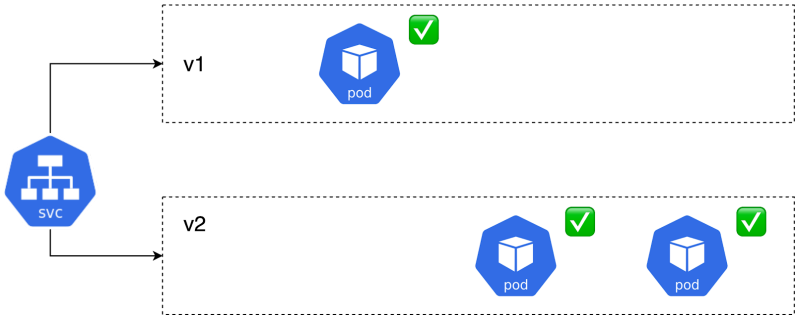
Представим, что у нас уже есть один развернутый Deployment, v1. В случае сине-зеленого развертывания вместо обновления контейнера и замены существующего развертывания, необходимо рядом подготовить ещё одно. Для этого нужно задать отдельный манифест с новой версией приложения v2. Селектор (selector) нового манифеста должен быть отличен от старого. После того, как новое развертывание готово, можно проверить работоспособность подов этого развертывания (например, с помощью запуска end-to-end тестов). Если развертывание произошло успешно, достаточно переключить сервис на новый Replica Set (для этого нужно поменять селектор сервиса).

Как правило, сине-зеленые развертывания автоматизируют с помощью процессов непрерывной интеграции и доставки, но для лучшего понимания их работы, имеет смысл попрактиковаться и с ручным развертыванием.

Canary Deployment

Ещё один популярный подход - **Canary Deployments** - канареечные развертывания, когда мы выкатываем новую версию, постепенно распространяя её на всё большее количество систем и пользователей. Термин “канареечные” пришел из шахтерской среды. Один из самых ранних способов обнаружения в шахтах рудничного газа заключался в использовании в качестве газоанализаторов канареек. Канарейки очень чувствительны к газам и гибнут даже от незначительной примеси его в воздухе. В прежнее время шахтеры часто брали клетку с канарейкой в шахту и во время работы следили за птицей. Если она внезапно начинала проявлять признаки беспокойства или падала, люди поспешно покидали выработку, пока слышалось пение птицы, можно было работать спокойно.

В случае с канареечных развертыванием ПО мы, конечно, не используем птиц. Мы производим релиз таким образом, чтобы он становился доступен пользователям постепенно. Например, сначала развертывание может быть произведено только на сотрудников компании. Затем, если мониторинг не зафиксировал неполадок, мы можем развернуть релиз доступный для небольшой доли конечных пользователей. И, убедившись в приемлемом качестве новой версии, мы наконец можем открыть релиз для всех пользователей, полностью свернув старую версию приложения.



Реализация такого подхода на практике похожа на предыдущий кейс: мы задаем два разных развертывания (Replica Set) с разными версиями приложения. При этом оба развертывания используют один и тот же селектор, и, соответственно, сервис будет отправлять трафик на их обоих.

Практические задания

1. В этом уроке мы рассмотрели простое приложение, отдающее статику. Реализуйте для него концепции, изученные в прошлом уроке. Включите в бинарник информацию о версии приложения. Создайте Makefile и добавьте в него все необходимые команды для компиляции, сборки и запуска контейнера. Выделите хелсчеки в отдельные пробы.
2. Выберите любой сервис, написанный вами ранее. Проверьте, соответствует ли он практикам, изученные в предыдущем и текущем уроках. Разверните его в удаленный кластер. Подготовьте манифесты для всех стратегий развертывания, изученных в этом уроке (Rolling Update, Fixed Deployment, Blue-Green Deployment, Canary Deployment).

Дополнительные материалы

1. Готовим приложение для Kubernetes-кластера шаг за шагом: https://habr.com/ru/post/345332/
2. Пространства имен: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/
3. Документация по DNS для Kubernetes: https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/
4. Сайдкар, амбассадор и адаптер: https://kubernetes.io/blog/2015/06/the-distributed-system-toolkit-patterns/
5. Примеры манифестов для сайдкара и адаптера: https://matthewpalmer.net/kubernetes-app-developer/articles/multi-container-pod-design-patterns.html
6. Шаблоны Kubernetes - примеры и бесплатная книга: https://k8spatterns.io/
7. Сине-зеленые развертывания: https://martinfowler.com/bliki/BlueGreenDeployment.html
8. Канареечные развертывания: https://martinfowler.com/bliki/CanaryRelease.html
9. Статья по стратегиям развертывания в Kubernetes (на русском): https://habr.com/ru/company/flant/blog/471620/