GeekBrains

Урок 2. Запуск приложений в Kubernetes. Конфигурирование. Мониторинг. Отладка

На этом уроке

1. Вспомним, что такое Kubernetes.
2. Напишем go-приложение и контейнеризируем его с помощью Docker.
3. Используя контейнеризованное приложение, запустим его с помощью Kubernetes.
4. Поговорим о “подводных камнях” при запуске приложения в Kubernetes.

Оглавление

Теория урока

Kubernetes уже изучался в рамках курса “Микросервисная архитектура и контейнеризация”. Поэтому, мы сейчас быстро пройдемся по основным понятия, освежим в памяти архитектуру и приступим к практической части занятия.

Kubernetes

Начнем с официального определения из документации:

Kubernetes (k8s) - это портативная расширяемая платформа с открытым исходным кодом для управления контейнеризованными рабочими нагрузками и сервисами, которая облегчает как декларативную настройку, так и автоматизацию.

Или простыми словами, k8s - оркестратор контейнеров или система управления контейнерами.

Kubernetes предоставляет:

- Мониторинг сервисов и распределение нагрузки
- Оркестрацию хранилища
- Автоматические развертывания и откаты
- Автоматическое распределение нагрузки
- Самоконтроль
- Управление конфиденциальной информацией и конфигурацией

При развертывании k8s взаимодействие происходит с кластером. Кластер состоит из набора машин, которые также называют узлами или нодами (nodes). У кластера всегда есть как минимум 1 рабочий узел. В рабочих узлах расположены поды (pods), которые являются компонентами приложения. Сами поды могут включать в себя один или несколько запущенных контейнеров. Плоскость управления (control plane) управляет рабочими нодами и подами в кластере. Компонентами плоскости управления (control plane) являются:

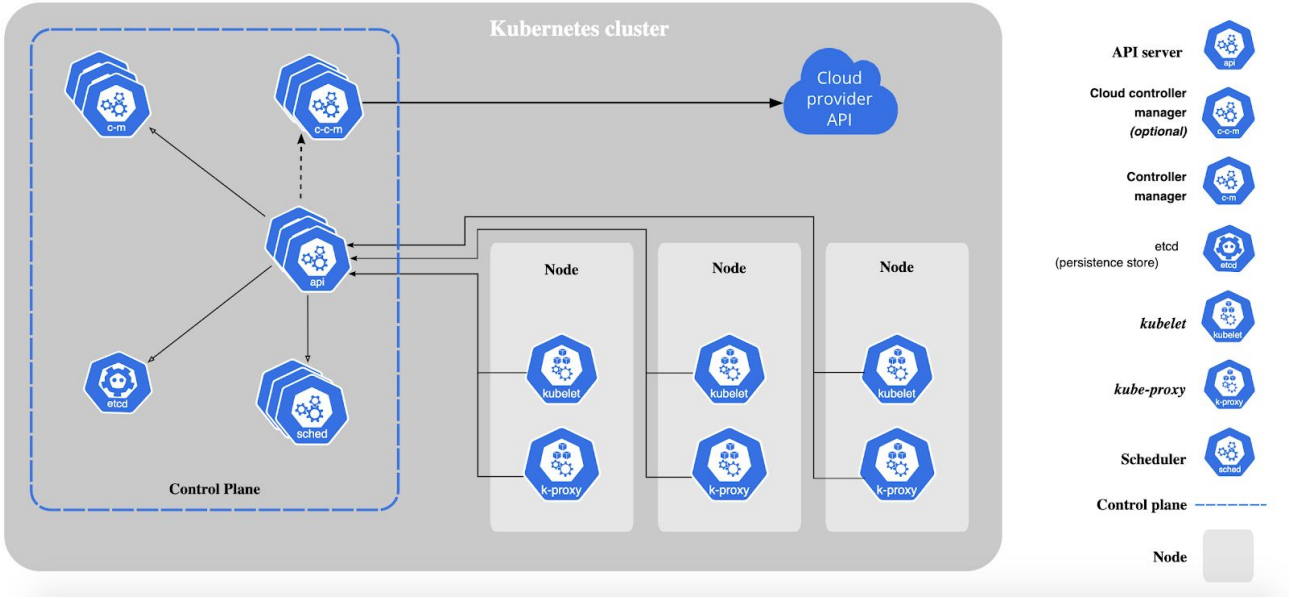
- **API Server** - клиентская часть панели управления (control plane)
- **etcd** - распределенное и высоконадежное хранилище данных в формате key-value, которое используется как основное хранилище всех данных кластера в Kubernetes
- **kube-scheduler** - отслеживает поды, которые не привязаны к конкретным нодам и выбирает узел, на котором они должны работать

- **kube-controller-manager** - объект, который запускает процессы контроллеров. Контроллер - это управляющий цикл, который проверяет состояние кластера и вносит изменения, пытаясь привести текущее состояние кластера к желаемому.
- **cloud-controller-manager** - объект, который запускает контроллеры, которые взаимодействуют с облачными провайдерами.

Эти компоненты работают на каждом из узлов, поддерживая работу подов:

- **kubelet** - следит за тем, чтобы контейнеры были запущены в поде.
- **kube-proxy** - является сетевым прокси. Конфигурирует правила сети на узлах. Разрешают сетевые подключения к подам.

Кластер с основными компонентами можно посмотреть на рисунке ниже.



Основные ресурсы

Составление конфигураций для каждого из ресурсов будет произведено в практической части.

Deployment

Deployment - это объект Kubernetes, который представляет работающее приложение в кластере, обеспечивает декларативные обновления для подов, а также позволяет автоматизировать переход от одной версии приложения к другой. Переход осуществляется без остановки работы системы. При описании данного ресурса описывается желаемое состояние системы, на основе которого контроллеры пытаются изменить текущее состояние системы, чтобы достичь желаемое состояние.

Service

В k8s, **Service** - это абстракция, которая определяет логическое множество подов и правила для получения доступа к ним. Цель данного ресурса - обеспечить доступ к подам, даже если поды были

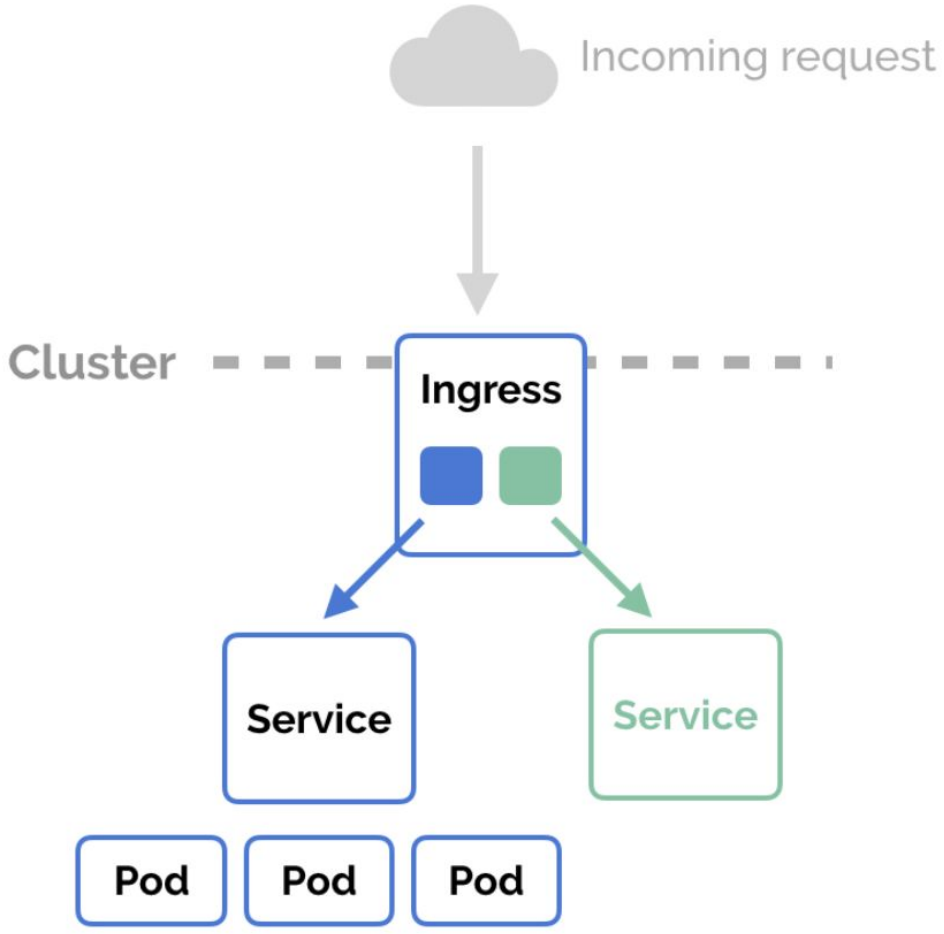
изменены. Каждый под получает свой собственный IP адрес. Но если под перестает быть рабочим и на смену к нему приходит новый под, то у этого нового пода уже другой IP адрес. Service решает эту проблему, так как под капотом он знает новый IP адрес пода и обращается уже по новому IP. А для клиента, который взаимодействует с Service, ничего не поменялось. Точка доступа к Service осталась прежней.

Существует несколько типов сервисов:

- **ClusterIP** - доступен только внутри кластера. Является типом по умолчанию.
- **NodePort** - дает каждому поду внешний IP, который доступен извне кластера. При данном типе можно обратиться к сервису по **NodeIP:NodePort.** Компонент kube-proxy принимает запросы на этом порту и перенаправляет трафик на нужный порт.
- **LoadBalancer -** добавляет балансировщик нагрузки на основе облачного провайдера.

Ingress

Ingress - объект, который предоставляет данные к Services по HTTP/HTTPS, как показано на схеме.



Пробы (Probes)

Поговорим немного о проверках, которые выполняются в течение всего времени жизни подов.

Readiness probe

Данная проверка узнает готовность пода принимать трафик. Если проверка провалена, то под отключается от сервиса k8s и трафик на него не поступает до тех пор, пока очередная проверка не завершится успешно.

Liveness probe

Данная проверка узнает “живучесть” пода и перезапускает под при неудачном завершении.

Практическая часть

Цель всей практической части состоит в том, чтобы запустить go-приложение в Kubernetes и начать с ним взаимодействовать. Для этого нам потребуется:

- Написать простое go-приложение, у которого будет реализовано несколько хэндлеров.
- Написать Dockerfile, чтобы можно было поднять go-приложение в контейнере и отправить контейнер в dockerhub на свой аккаунт.
- Написать необходимые файлы для взаимодействия с абстракциями Kubernetes и используя minikube и kubectl, которыезачастую устанавливаются вместе , поднять Kubernetes локально.

Написание go-приложения

Начнем с простого, постепенно напишем go-приложение, у которого будет несколько хэндлеров. В процессе написания мы будем потихоньку его усложнять, дополняя различными важными объектами.

Простейший сервис

Создадим директорию **k8s-go-app**, в которой создадим следующий **main.go** файл:

```
package main

import (
  "fmt"
  "log"
  "net/http"
)
```

```
func main() {
  http.HandleFunc("/", handler)

  port := "8080"
  log.Printf("start server on port: %s", port)
  log.Fatal(http.ListenAndServe(":"+port, nil))
}

func handler(w http.ResponseWriter, _ *http.Request) {
  _, _ = fmt.Fprint(w, "Hello, World! Welcome to GeekBrains!\n")
}
```

Здесь мы реализовали простой сервис, у которого реализован один хендлер на **/.** Сервис можно запустить командой **go run main.go,** а с помощью команды **curl http://localhost:8080/** проверить работоспособность.

Версионирование

Добавим версионионирование в наше приложение. Создадим файлик **version/version.go**:

```
package version

var (
  Version = "unset"
  Build   = "unset"
  Commit  = "unset"
)
```

Данные значения мы будем выставлять позже с помощью **ldflags** во время **go install**.

Добавим продвинутый роутинг

Потенциально, для приложений, которые пишутся в продакшн может понадобиться более продвинутый роутинг. Попробуем добавить в наш простой сервис echo, также можно использовать любой другой: gorilla/mux, httprouter и т.п. Попробуем добавить роутинг таким образом, чтобы перейти с одного фреймворка на другой было проще.

Создадим рядом с **main.go** следующую директорию с файлом **server/echo.go**:

```
package server

import (
  "context"
  "net/http"
```

```
  "github.com/labstack/echo/v4"
  "github.com/labstack/echo/v4/middleware"
)

type Server struct {
	VersionInfo

	port string
}

type VersionInfo struct {
	Version string
	Commit  string
	Build   string
}

func New(info VersionInfo, port string) *Server {
	return &Server{
		VersionInfo: info,
		port:        port,
	}
}


func (s Server) Serve(ctx context.Context) error {
  e := echo.New()
  e.HideBanner = true
  e.Use(middleware.Recover())
  e.Use(middleware.Recover())
  s.initHandlers(e)

  go func() {
     e.Logger.Infof("start server on port: %s", s.port)
     err := e.Start(":" + s.port)
     if err != nil {
        e.Logger.Errorf("start server error: %v", err)
     }
  }()

  <-ctx.Done()

  return e.Shutdown(ctx)
}

func (s Server) initHandlers(e *echo.Echo) {
  e.GET("/", handler)

  e.Any("/*", func(c echo.Context) error {
     return c.NoContent(http.StatusNotFound)
  })
}

func handler(c echo.Context) error {
```

```
  return c.String(http.StatusOK, "Hello, World! Welcome to GeekBrains!\n")
}
```

Здесь мы реализовали структуру **Server** и несколько методов этой структуры. В структуре **Server** на данный момент только одно поле - **port**. В дальнейшем, эту структуру можно расширить, когда нужно будет внедрить в сервис другие зависимости (базу данных, другой логгер и т.п.). В метод **Serve** передается **ctx**, который блочит выполнение с помощью **<-ctx.Done()**. Сервис закончит работу, когда из канала **Done()** мы что-то прочитаем. Мы оставили с предыдущей реализации **handler**, который отрабатывает на **/**, поменяв сигнатуру для **echo**. Также добавили, что на всех других **path**, сервис будет отдавать **404**. Мы добавили выше код в **server/echo.go** и теперь нужно поменять старый код, чтобы все работало. Для этого необходимо немного изменить функцию main в файле **main.go:**

```
func main() {
  port := "8080"

  info := server.VersionInfo{
            Version: version.Version,
            Commit:  version.Commit,
            Build:   version.Build,
      }

  srv := server.New(info, port)
  ctx, cancel := context.WithCancel(context.Background())
  go func() {
     err := srv.Serve(ctx)
     if err != nil {
        log.Println(fmt.Errorf("serve: %w", err))
        return
     }
  }()

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

  <-osSigChan
  log.Println("OS interrupting signal has received")

  cancel()
}
```

Здесь мы начали использовать наш пакет **server** с его функциями. Добавили context-отмены, чтобы было можно тушить сервис по сигналам OS. Теперь необходимо проверить, что ничего не сломалось. Выполним команды в директории с **main.go**: **go mod init** и **go mod tidy** и запустим наш сервис с **go run main.go.** Команда **curl http://localhost:8080** должна отработать как ожидаем: получить в ответ **Hello, World! Welcome to GeekBrains!**.

Добавим конфиг-файл

На данный момент, у нас есть только один варьируемый параметр - **port.** Но даже его менять не очень удобно, необходимо лезть в код, находить нужную строчку и менять значение. А если таких параметров больше - это становится еще сложнее. Более гибкий подход - хранить все конфиги в одном месте без привязки к языку программирования. В данном сервисе мы реализуем конфиги через переменные окружения, также можно использовать любой другой подход: хранить их в JSON/TOML/YAML/etc формате.

Создадим рядом с **main.go** файл директорию **config/** со следующими файлами **config.go** и **local.env**:

config.go

```
package config

import (
  "fmt"
  "path/filepath"

  "github.com/joho/godotenv"
  "github.com/kelseyhightower/envconfig"
)

type LaunchMode string

const (
  LocalEnv LaunchMode = "local"
  ProdEnv  LaunchMode = "prod"
)

type Config struct {
  Port string `envconfig:"PORT" default:"8080"`
}

func Load(launchMode LaunchMode, path string) (*Config, error) {
  switch launchMode {
  case LocalEnv:
     cfgPath := filepath.Join(path, fmt.Sprintf("%s.env", launchMode))

     err := godotenv.Load(cfgPath)
     if err != nil {
        return nil, fmt.Errorf("load .env config file: %w", err)
     }
  case ProdEnv:
     // all settings should be provided as env variables
  default:
     return nil, fmt.Errorf("unexpected LAUNCH_MODE: [%s]", launchMode)
  }

  config := new(Config)
```

```
  err := envconfig.Process("", config)
  if err != nil {
     return nil, fmt.Errorf("get config from env: %w", err)
  }

  return config, nil
}
```

Здесь мы используем внешнюю библиотеку для работы с переменными окружения и проверяем, в каком окружении мы подняли наш сервис: **local/prod.** В зависимости от того, в каком окружении поднят сервис, происходит чтение файла с конфигом и его обработка. Здесь показано, что prod конфиг желательно не хранить в файле, а задавать при запуске.

local.env:

```
PORT=8080
```

В файлике с конфигом на данный момент только одно значение - **PORT**.

Файл **main.go** изменился незначительно:

```
func main() {
  launchMode := config.LaunchMode(os.Getenv("LAUNCH_MODE"))
  if len(launchMode) == 0 {
     launchMode = config.LocalEnv
  }
  log.Printf("LAUNCH MODE: %v", launchMode)

  cfg, err := config.Load(launchMode, "./config")
  if err != nil {
     log.Fatal(err)
  }
  log.Printf("CONFIG: %+v", cfg)

  info := server.VersionInfo{
            Version: version.Version,
            Commit:  version.Commit,
            Build:   version.Build,
      }

  srv := server.New(info, cfg.Port)
  ctx, cancel := context.WithCancel(context.Background())
  go func() {
     err := srv.Serve(ctx)
     if err != nil {
        log.Println(fmt.Errorf("serve: %w", err))
        return
```

```
    }
  }()

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

  <-osSigChan
  log.Println("OS interrupting signal has received")

  cancel()
}
```

Здесь мы читаем переменную окружения **LAUNCH_MODE.** Если оно пустое, то сервис запускается в окружении **local**. Затем вызываем нашу библиотеку config для чтения конфиг параметров и при создании структуры **Server** передаем **cfg.Port.** Также хорошим шагом является логирование конфиг параметров, это поможет быстрее найти ошибки.

Добавим heartbeat/version хендлеры

В сервисы можно добавить еще две ручки: **heartbeat** и **version.**

heartbeat - это специальный маркер/сигнал, который показывает работоспособность нашего сервиса. Если сервис не отвечает на данную ручку , то это может говорить о наличии проблем.

version - ручка, которая отдает параметры сервиса. Зачастую это сама версия сервиса, номер билда, номер коммита etc.

Реализацию хендлеров добавим в файл **server/echo.go**:

```
func heartbeatHandler(c echo.Context) error {
  return c.NoContent(http.StatusOK)
}

func (s Server) versionHandler(c echo.Context) error {
  return c.JSON(
     http.StatusOK,
     map[string]string{
        "version": s.VersionInfo.Version,
        "commit":  s.VersionInfo.Commit,
        "build":   s.VersionInfo.Build,
     },
  )
}
```

build по факту заглушка на будущее. **commit** можно уже отображать с помощью команды

```
git rev-parse HEAD
```

Чуть подробнее будет в следующем пункте.

И поменяем в том же файле функцию **initHandlers:**

```
func (s Server) initHandlers(e *echo.Echo) {
  e.GET("/", handler)
  e.GET("/__heartbeat__", heartbeatHandler)
  e.GET("/__version__", s.versionHandler)

  e.Any("/*", func(c echo.Context) error {
     return c.NoContent(http.StatusNotFound)
  })
}
```

Makefile

С go-приложением практически разобрались. Для удобства осталось добавить Makefile. Makefile - это файл с командами alias. Для того, чтобы не вводить постоянно длинные команды, можно создать что-то наподобие alias и вызывать команды с помощью **make**. В нынешнем варианте можно добавить следующее:

```
#write here path for your project
PROJECT :=
GIT_COMMIT := $(shell git rev-parse HEAD)
VERSION := latest
APP_NAME := k8s-go-app

all: run

run:
  go install -ldflags="-X '$(PROJECT)/version.Version=$(VERSION)' \
  -X '$(PROJECT)/version.Commit=$(GIT_COMMIT)'" && $(APP_NAME)
```

И в терминале с помощью **make run** мы можем запустить наш сервис. А с помощью команды **curl localhost:8080/__version__** мы получим примерно следующее:

```
{"build":"","commit":"245531c43ec628dbfa3ea86d461a95d277223466","version":"latest"}
```

ВНИМАНИЕ!

Использование **version latest** в продакшене является плохой практикой, так как может быть использована неправильная версия из “кэша” и возникнут проблемы, при которых вы будете получать не то, что ожидаете.

Docker-контейнер

Продолжаем наше взрывное шоу. Теперь нам необходимо создать контейнер с нашим приложением и отправить его на dockerhub.

Создание Dockerfile

На предыдущем уроке был рассмотрен процесс создания Dockerfile. Мы будем использовать мультистейджевую сборку:

```
ARG GIT_COMMIT
ARG VERSION
ARG PROJECT

FROM golang:1.15.1 as builder
ARG GIT_COMMIT
ENV GIT_COMMIT=$GIT_COMMIT

ARG VERSION
ENV VERSION=$VERSION

ARG PROJECT
ENV PROJECT=$PROJECT

ENV GOSUMDB=off
ENV GO111MODULE=on
ENV WORKDIR=${GOPATH}/src/k8s-go-app

COPY . ${WORKDIR}
WORKDIR ${WORKDIR}

RUN set -xe ;\
    go install -ldflags="-X ${PROJECT}/version.Version=${VERSION} -X
${PROJECT}/version.Commit=${GIT_COMMIT}" ;\
    ls -lhtr /go/bin/

FROM golang:1.15.1

EXPOSE 8080

WORKDIR /go/bin

COPY --from=builder /go/bin/k8s-go-app .
COPY --from=builder ${GOPATH}/src/k8s-go-app/config/*.env ./config/
```

```
ENTRYPOINT ["/go/bin/k8s-go-app"]
```

Данный dockerfile мало чем отличается от dockerfile с предыдущего урока. Чтобы была возможность использовать конфиги мы добавили проброс ***.env** файлов в контейнер, также мы прокидываем переменные окружения **GIT_COMMIT, VERSION, PROJECT** и используем при команде **go install** для нашей ручки **__version__.** Мы слегка поменяем наш **Makefile** для удобства:

```
#write here your username
USERNAME :=
APP_NAME := k8s-go-app
VERSION := latest

#write here path for your project
PROJECT :=
GIT_COMMIT := $(shell git rev-parse HEAD)


all: run

run:
  go install -ldflags="-X '$(PROJECT)/version.Version=$(VERSION)' \
  -X '$(PROJECT)/version.Commit=$(GIT_COMMIT)'" && $(APP_NAME)

build_container:
  docker build --build-arg=GIT_COMMIT=$(GIT_COMMIT) --build-arg=VERSION=$(VERSION)
--build-arg=PROJECT=$(PROJECT)\
  -t docker.io/$(USERNAME)/$(APP_NAME):$(VERSION) .
```

Теперь попробуем сделать **make build_container** и **make run_container**. А затем сходить в ручку **__version__.** Все должно работать как ожидается и отдавать версию:

```
{"build":"","commit":"245531c43ec628dbfa3ea86d461a95d277223466","version":"latest"}
```

ВНИМАНИЕ!

Использование **version latest** в продакшене является плохой практикой, так как может быть использована неправильная версия из “кэша” и возникнут проблемы, при которых вы будете получать не то, что ожидаете.

Dockerhub

Теперь нам нужно отправить наш контейнер в dockerhub, чтобы использовать его при работе с Kubernetes. Для этого необходимо зарегистрироваться на сайте Docker Hub.

После регистрации необходимо выполнить в терминале команду **docker login** и ввести необходимые данные. Затем, нам нужно отправить наш готовенький контейнер в dockerhub. Добавим следующую команду в **Makefile**:

```
push_container:
  docker push  docker.io/$(USERNAME)/$(APP_NAME):$(VERSION)
```

Если все прошло успешно, то здесь https://hub.docker.com/repository/docker/$(USERNAME)/$(APP_NAME) вы найдете информацию о контейнере. Вместо **$(...)** нужно вставить ваши данные.

Работа с Kubernetes

Теперь перейдем к самой интересной части - работа с Kubernetes. Для того, чтобы работать с k8s, необходимо установить minikube и kubectl.Обратите внимание, что зачастую kubectl устанавливается вместе с minikube.

Перед следующими шагами нужно выполнить команду **minikube start.**

ВНИМАНИЕ!

Если вы пользователь MacOS, то вам следует выполнить команду **minikube start --vm=true.** Это связано с тем, что утилитка minikube на MacOS не работает при некоторых командах. Вот обсуждение на эту тему. В основном, это касается работы с Ingress - с ним возникают проблемы. Также не совсем так ведет себя команда **minikube ip**. Обсуждение проблемы. Но это менее критично, так как результат можно использовать для дальнейшей работы. Если вы все-таки установили без флага **vm**, выполните **minikube delete** и заново команду с флагом.

Output:

```
✨  Using the docker driver based on existing profile
👍  Starting control plane node minikube in cluster minikube
🏃  Updating the running docker "minikube" container ...
🐳  Preparing Kubernetes v1.19.2 on Docker 19.03.8 ...
🔎  Verifying Kubernetes components...
🌟  Enabled addons: storage-provisioner, default-storageclass, dashboard

❗  /usr/local/bin/kubectl is version 1.15.5, which may have incompatibilites
with Kubernetes 1.19.2.
💡  Want kubectl v1.19.2? Try 'minikube kubectl -- get pods -A'
```

```
🏄  Done! kubectl is now configured to use "minikube" by default
```

Deployment

Начнем мы с Kubernetes Deployments (о нем шла речь в теоретической части). Для создания необходимо написать следующую конфигурацию **deployment.yaml**:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: k8s-go-app
spec:
 replicas: 2
 strategy:
   type: RollingUpdate
   rollingUpdate:
     maxUnavailable: 1
     maxSurge: 1
 selector:
   matchLabels:
     app: k8s-go-app
 template:
   metadata:
     labels:
       app: k8s-go-app
   spec:
     containers:
       - name: k8s-go-app
         #IMPORTANT: provide your username here
         image: docker.io/$(USERNAME)/k8s-go-app:latest
         imagePullPolicy: Always
         ports:
           - containerPort: 8080
         livenessProbe:
           httpGet:
             path: /__heartbeat__
             port: 8080
             scheme: HTTP
           initialDelaySeconds: 5
           periodSeconds: 15
           timeoutSeconds: 5
         readinessProbe:
           httpGet:
             path: /__heartbeat__
             port: 8080
             scheme: HTTP
           initialDelaySeconds: 5
           timeoutSeconds: 1
```

Давайте по порядку разберем что здесь написано.

1. **kind** - показывает, какой тип ресурса мы описываем. Сейчас описывается Deployment, затем будем описывать другие типы.
2. **metadata/name** - название ресурса k8s.
3. **replicas** - свойство объекта, показывающее сколько реплик (экземпляров) подов можно запустить. Мы запускаем 2 реплики.
4. **strategy/type** - описывает стратегию развертывания при переходе с текущей версии на новую. **RollingUpdate** обеспечивает нулевое время простоя системы.
5. **RollingUpdate/maxUnavailable** - показывает максимальное количество недоступных подов при выполнении обновления системы. Является свойством **RollingUpdate**. В нашем варианте с двумя репликами значение этого свойства указывает на то, что после завершения работы одного пода ещё один будет выполняться, что делает приложение доступным в ходе обновления.
6. **RollingUpdate/maxSurge** - описывает максимальное число подов, которое можно добавить в развертывание. Является свойством **RollingUpdate**. В нашем случае его значение 1, следовательно, при переходе на новую версию программы, мы можем добавить в кластер ещё один под. Поэтому, у нас может быть одновременно 3 запущенных пода.
7. **selector/matchLabels/app** - показывает, что развертывание будет применимо к подам с таким лейблом.
8. **template** - задаёт шаблон пода, который ресурс Deployment будет использовать для создания новых подов по заданной конфигурации.
9. **spec/containers/image** - название образа для контейнера. В нашем случае мы берем его с dockerhub по username и по названию приложения.
10. **spec/Containers/ImagePullPolicy** - определяет порядок работы с образами. В нашем случае это Always - всегда загружаем образ из репозитория.
11. **containers/ports/containerPort** - должен совпадать с портом, на котором запущено go приложение.
12. **containers/liveness** - определяет правила проверки, жив ли под. Если данная проба провалена - приложение перезапустится.
13. **containers/readiness** - определяет правила проверки, готов ли под к трафику. Если проба провалена - контейнер будет удален из балансировщиков нагрузки сервиса. В нашем простом приложении мы используем одну и ту же ручку **__heartbeat__** для проверки данных проб.

Теперь выполним следующую команду:

```
kubectl apply -f deployment.yaml
```

Данной командой мы, с помощью утилиты **kubectl,** создали объект Deployment k8s.

Output:

```
deployment.apps/k8s-go-app created
```

Теперь посмотрим на наши **deployments** с помощью команды:

```
kubectl get deployments
```

Output:

```
NAME              READY   UP-TO-DATE   AVAILABLE   AGE
k8s-go-app        2/2     2            2           38s
```

Как можно видеть, у нас все работает и при запуске проблем не возникло. Теперь посмотрим на наши поды:

```
kubectl get pods
```

Output:

```
NAME                              READY   STATUS    RESTARTS   AGE
k8s-go-app-58f6d66886-8ztt7       1/1     Running   0          2m1s
k8s-go-app-58f6d66886-xzwpc       1/1     Running   0          2m1s
```

И здесь тоже все замечательно, все работает. Но это еще не все. Допустим, мы хотим посмотреть, что происходит на каждом поде. Попробуем выполнить следующую команду:

```
kubectl logs -f k8s-go-app-58f6d66886-xzwpc
```

logs - выводит в терминал логи пода.
-f - данный флаг выводит логи в стриминговом формате, т.е. они будут выводиться в терминал, как только появятся.

Output:

```
2020/11/14 07:19:22 LAUNCH MODE: local
2020/11/14 07:19:22 CONFIG: &{Port:8080}
```

```
⇨ http server started on [::]:8080
{"time":"2020-11-14T07:19:33.863739543Z","id":"","remote_ip":"172.17.0.1","host":"172
.17.0.7:8080","method":"GET","uri":"/__heartbeat__","user_agent":"kube-probe/1.19","s
tatus":200,"error":"","latency":983,"latency_human":"983ns","bytes_in":0,"bytes_out":
0}
{"time":"2020-11-14T07:19:37.098137241Z","id":"","remote_ip":"172.17.0.1","host":"172
.17.0.7:8080","method":"GET","uri":"/__heartbeat__","user_agent":"kube-probe/1.19","s
tatus":200,"error":"","latency":1053,"latency_human":"1.053µs","bytes_in":0,"bytes_ou
t":0}
...
```

Здесь можно увидеть, что вывел наш сервис с момента запуска в данном поде. Мы напечатали в каком окружении работаем и наш конфиг файл при запуске. Затем идут логи сервиса. При написании go-приложения, мы добавили логирование запросов на сервис. В **output** выше мы видим как выполняется наша **liveness** проба (отправка запроса в хэндлер **__heartbeat__**).

Чуть больше про команду **logs** и ее опции можно узнать выполнив команду **kubectl logs --help.**

Давайте попробуем сходить в наш сервис. В случае, когда мы не знаем host:port, можно использовать несколько подходов:

port-forward подход

Можно взять **NAME** пода и замапить (сделать map) между локальным портом и портом пода с помощью команды:

```
kubectl port-forward k8s-go-app-58f6d66886-xzwpc 8080:8080
```

Output:

```
Forwarding from 127.0.0.1:8080 -> 8080
Forwarding from [::1]:8080 -> 8080
```

Выполнив в другом терминале **curl http://localhost:8080** мы получим наш знакомый текст **Hello, World! Welcome to GeekBrains!**, а в терминале с командой port-forward **Handling connection for 8080.**

Данный подход хорош только для локального тестирования и мелких проверок. Рассмотрим чуть более сложный подход.

Service

Поды могут перезапускаться по различным причинам: ошибка на liveliness или readiness пробе, также поды могут быть прибиты, если нода на которой они запущены - отключилась. IP адреса подов

меняются и k8s обеспечивает стабильные точки доступа для объектов типа **Service**. Напишем конфигурацию для объекта k8s типа **Service** в файлике **service.yaml**:

```
apiVersion: v1
kind: Service
metadata:
 name: k8s-go-app-srv
spec:
 type: NodePort
 ports:
   - name: http
     port: 9090
     targetPort: 8080
 selector:
   app: k8s-go-app
```

Пройдемся быстренько по незнакомым полям:

1. **spec/type** - описание типа сервиса. В нашем случае это NodePort, который дает каждой ноде внешний IP для обработки запросов со стороны.
2. **ports/port** - принимает входящие запросы на этот порт и перенаправляет их на targetPort.
3. **selector/app** - название, определяющие для каких подов применим сервис.

Теперь выполним данную конфигурацию с помощью команды:

```
kubectl apply -f service.yaml
```

Output:

```
service/k8s-go-app-srv created
```

Сервис создан, теперь проверим это с помощью команды:

```
 kubectl get service
```

Output:

```
NAME                     TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
k8s-go-app-srv           NodePort    10.107.177.78   <none>        9090:32564/TCP   83s
```

Теперь нам нужно получить **host:port**, чтобы узнать куда мы можем обращаться с помощью команды:

```
minikube service k8s-go-app-srv  --url
```

В output команды будет представлен **host:port**, по которому мы можем сделать запрос **curl http://host:port.**

Ingress

Рассмотрим еще один объект k8s. **Ingress** - это API объект, который управляет внешним доступом к сервисам в кластере. На предыдущем степе мы рассмотрели объект k8s **Service** и прописали, что его тип **NodePort:** ***type: NodePort.*** По дефолту, если не указывать **type** в конфигурации, то **Service** создается с типом **ClusterIP**, такой сервис может принимать только запросы внутри кластера.

Попробуйте закомментировать строку с типом в конфигурации **service.yaml**. Если выполнить команду **kubectl apply -f service.yaml**， то вернется ошибка:

```
The Service "k8s-go-app-srv" is invalid: spec.ports[0].nodePort: Forbidden: may
not be used when `type` is 'ClusterIP'
```

Необходимо удалить **k8s-go-app-srv** и перезапустить заново с закомментированной строчкой. И теперь, если мы выполним команду для получения host:port для нашего сервиса,

```
minikube service k8s-go-app-srv  --url
```

то мы получим:

```
😿  service default/k8s-go-app-srv has no node port
```

Попробуем написать **ingress.yaml** файл:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
 annotations:
   kubernetes.io/ingress.class: nginx
   ingress.kubernetes.io/rewrite-target: /
 labels:
   app: k8s-go-app-ing
 name: k8s-go-app-ing
spec:
 backend:
   serviceName: k8s-go-app-srv
   servicePort: 8080
```

```
rules:
  - host: k8s-go-app.host
    http:
      paths:
        - path: /
          backend:
            serviceName: k8s-go-app-srv
            servicePort: 8080
```

Теперь нам необходимо выполнить следующую команду, чтобы можно было использовать ingress:

```
minikube addons enable ingress
```

ВНИМАНИЕ!

Если вы пользователь MacOS, то вы должны были запускать **minikube** с помощью **minikube start --vm=true.** Это связано с тем, что утилита minikube на MacOS не работает при этой команде - вы получите ошибку. Вот обсуждение на эту тему. Если вы все-таки установили без флага **vm**, выполните **minikube delete** и заново команду с флагом.

Выполним команду **minikube ip**.

Output:

```
192.168.64.2
```

Добавим в конец файла **/etc/host** следующую строчку **192.168.64.2 k8s-go-app.host.**

И теперь, при выполнении команды **curl k8s-go-app.host** мы получаем наш давно знакомый результат:

```
Hello, World! Welcome to GeekBrains!
```

Добавим в завершении раздела следующие строчки в **Makefile**:

```
apply_deploy:
  kubectl apply -f deployment.yaml

apply_service:
  kubectl apply -f service.yaml

apply_ingress:
  kubectl apply -f ingress.yaml
```

Теперь наш сервис может подниматься в k8s и у нас есть к нему доступ. В заключении можно рассмотреть еще несколько команд:

```
minikube dashboard
```

Команда откроет в браузере вкладку с информацией о кластерах.

Команда:

```
kubectl get events
```

покажет нам все события, которые происходили в кластере.

Потенциальные ошибки

Попробуем рассмотреть ситуацию, при которой у нас что-то не работает при конфигурировании. Для начала удалим все, что мы сделали и попробуем с начала все сконфигурировать, но с ошибкой.

```
kubectl delete deployment k8s-go-app
kubectl delete service k8s-go-app-srv
```

Теперь в файле **deployment.yaml** сменим порты у проб (liveness, readiness) с **8080** на **8090** и сконфигурируем наш Deployment **make apply_deploy.** Теперь команда **kubectl get pods** выдает следующее:

```
NAME                              READY   STATUS    RESTARTS   AGE
k8s-go-app-8664c8c65f-7544l       0/1     Running   1          84s
k8s-go-app-8664c8c65f-z8sv7       0/1     Running   1          84s
```

Можно заметить, что в столбце **READY** значение 0/1, а количество **RESTARTS** увеличивается и принимает ненулевое значение. Со временем значение в столбце **STATUS** сменится на **CrashLoopBackOff.** То есть под пытался подняться подряд несколько раз по backoff, но ничего не вышло. Можно попробовать получить подробную информацию по поду с помощью команды для одного из подов:

```
 kubectl describe pod k8s-go-app-8664c8c65f-7544l
```

В output будет представлена различная информация по конфигурации пода и что у него происходит.

Наше внимание привлекают следующие строчки:

```
Liveness probe failed: Get "http://172.17.0.11:8090/__heartbeat__": dial tcp
172.17.0.11:8090: connect: connection refused
Readiness probe failed: Get "http://172.17.0.11:8090/__heartbeat__": dial tcp
172.17.0.11:8090: connect: connection refused
```

В таких случаях стоит внимательно посмотреть на конфигурацию данных проб и убедиться, что порты проставлены верно.

Проблему с портами можно поймать и в файлике **service.yaml**, стоит убедиться, что **targetPort** стоит тот, который вам нужен. Внимательно стоит отнестись и к пункту **selector/app** данного файла, он должен совпадать с именем **metadata/name** файла **deployment.yaml.**

Заключение

Итак, на этом уроке:

1. Мы вспомнили теоретический материал по **Kubernetes**.
2. Написали приложение на go, подняли его в **Kubernetes** и смогли обращаться к нему извне.

Практическое Задание

1. Запустить в k8s любое приложение несколькими ручками с ресурсами Deployment, Service, Ingress. Добавить две пробы: readiness, liveness. Приложение должно уметь принимать запросы как через http с проверкой через curl, так и через grpc, для этого проще всего написать простенький клиент на go, который сможет общаться по grpc с вашим сервисом.
2. *Поднять два сервиса в k8s и наладить между ними общение любым способом.

Дополнительные материалы

1. ARG,ENV через build-arg для Docker. Для мультистеджевых сборок решение.
2. YouTube: Микросервисы в продакшн. От коммита до релиза: полная автоматизация в Kubernetes.
3. Habr: 10 типичных ошибок при использовании Kubernetes.
4. Книга: Kubernetes in Action.

Используемые источники

1. Официальная документация по Kubernetes: Deployments.
2. Официальная документация по Kubernetes: Service.
3. Официальная документация по Kubernetes: Ingress.
4. Официальная документация по Kubernetes: Probes.
5. Статья Deploying a containerized Go app on Kubernetes.
6. Статья Write a Kubernetes-ready service from zero step-by-step
7. Статья Руководство по k8s.