GeekBrains

Backend-разработка на Go. Модуль 2

Запуск приложений в Kubernetes. Конфигурирование. Мониторинг. Отладка

На этом уроке

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

Оглавление

Теория урока

- Kubernetes
- Основные ресурсы
  - Deployment
  - Service
  - Ingress
- Пробы (Probes)
  - Readiness probe
  - Liveness probe

Практическая часть

- Написание go-приложения
  - Простейший сервис
  - Версионирование
  - Добавим продвинутый роутинг
  - Добавим конфиг-файл
  - Добавим heartbeat/version-хендлеры
  - Makefile
- Docker-контейнер
  - Создание Dockerfile
  - Dockerhub
- Работа с Kubernetes
  - Deployment
  - Service
  - Ingress

Потенциальные ошибки

Заключение

Практическое задание

Глоссарий

Дополнительные материалы

Используемые источники

Теория урока

Kubernetes уже изучался в рамках курса «Микросервисная архитектура и контейнеризация». Поэтому сейчас мы быстро пройдёмся по основным понятиям, освежим в памяти архитектуру платформы и приступим к практической части занятия.

Kubernetes

Начнём с официального определения из документации.

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

Простыми словами, k8s — оркестратор контейнеров или система управления контейнерами.

Kubernetes предоставляет:

- мониторинг сервисов и распределение нагрузки;
- оркестрацию хранилища;
- автоматические развёртывания и откаты;
- автоматическое распределение нагрузки;
- самоконтроль;
- управление конфиденциальной информацией и конфигурацией.

При развёртывании k8s взаимодействие происходит с кластером. Кластер состоит из набора машин, которые также называются узлами или нодами (nodes). У кластера всегда есть как минимум один рабочий узел. В рабочих узлах располагаются поды (pods), представляющие собой компоненты приложения. Поды включают в себя один или несколько запущенных контейнеров. Плоскость управления (control plane) управляет рабочими нодами и подами в кластере. Компоненты плоскости управления (control plane):

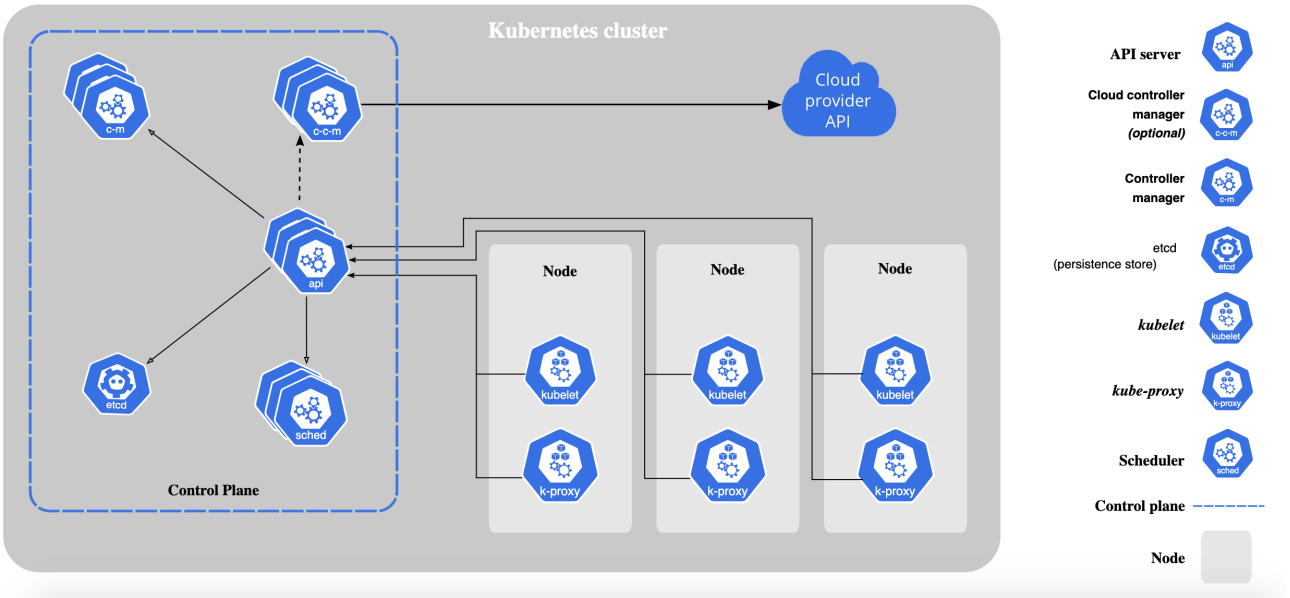
1. **API Server** — клиентская часть панели управления (control plane).
2. **etcd** — распределённое и высоконадёжное хранилище данных в формате key-value, которое используется как основное хранилище всех данных кластера в Kubernetes.

3. **kube-scheduler** — отслеживает поды, не привязанные к конкретным нодам, и выбирает узел, на котором они работают.
4. **kube-controller-manager** — объект, запускающий процессы контроллеров. Контроллер — это управляющий цикл, который проверяет состояние кластера и вносит изменения, пытаясь привести текущее состояние кластера к желаемому.
5. **cloud-controller-manager** — запускает контроллеры, которые взаимодействуют с облачными провайдерами.

Эти компоненты работают на всех узлах, поддерживая работу подов:

1. **kubelet** — следит за тем, чтобы контейнеры запускались в поде.
2. **kube-proxy** — сетевой прокси. Конфигурирует правила сети на узлах. Разрешает сетевые подключения к подам.

Кластер с основными компонентами указан на рисунке ниже.



Основные ресурсы

Составление конфигураций для всех ресурсов мы произведём 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 этого ресурса описывается желаемое состояние системы, на основе которого контроллеры пытаются изменить текущее состояние системы, чтобы достичь желаемого состояния.

Service

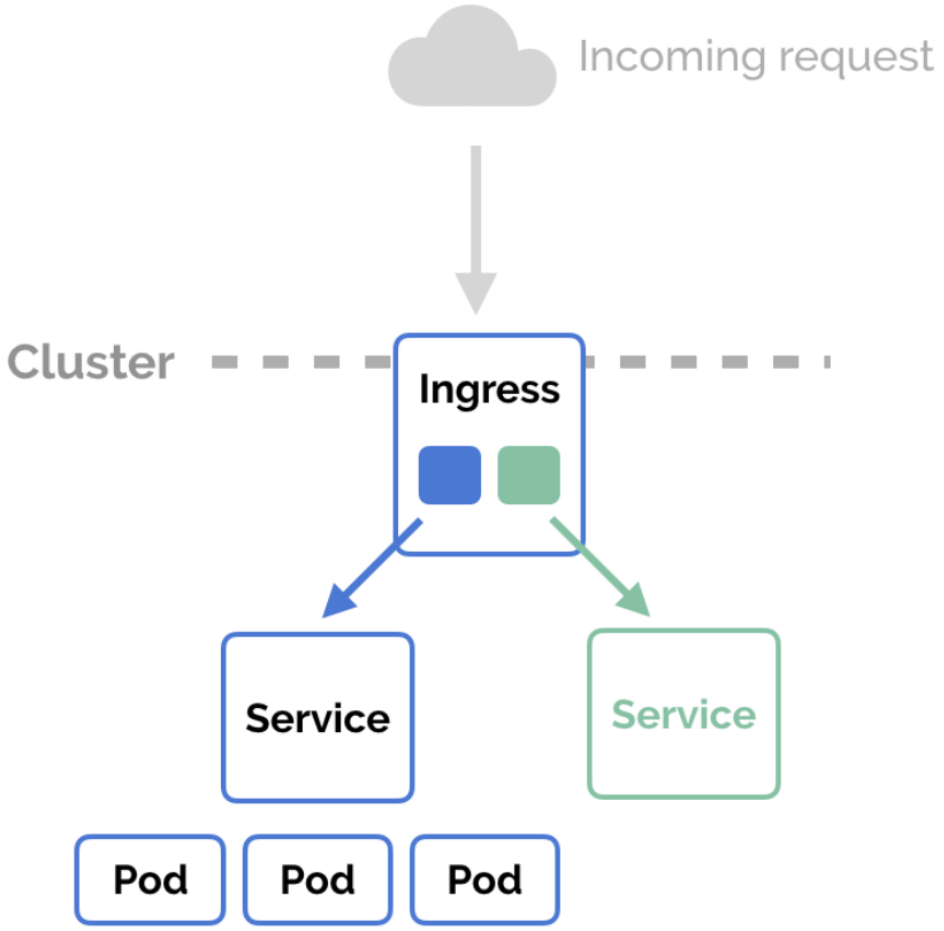
В k8s **Service** — это абстракция, которая определяет логическое множество подов и правила для получения доступа к ним. Цель этого ресурса — обеспечить доступ к подам, даже если поды изменились. Каждый под получает собственный IP-адрес. Но если под перестаёт быть рабочим, и на смену ему приходит новый под, то у последнего будет уже другой IP-адрес. Service решает эту проблему, так как знает новый IP-адрес пода и обращается уже по-новому IP. Для клиента, который взаимодействует с Service, ничего не меняется. Точка доступа к Service осталась прежней.

Есть несколько типов сервисов:

1. **ClusterIP** — доступен только внутри кластера, тип по умолчанию.
2. **NodePort** — даёт каждому поду внешний IP, который доступен извне кластера. С этим типом можно обратиться к сервису по **NodeIP:NodePort.** Компонент kube-proxy принимает запросы на этом порту и перенаправляет трафик на подходящий порт.
3. **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 всей «жизни» подов.

Readiness probe

Это проверка готовности пода принимать трафик. Если проверка провалилась, то под отключается от сервиса k8s, и трафик на него не поступает до тех пор, пока очередная проверка не завершится успешно.

Liveness probe

Это проверка «живучести» пода, которая 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:

1. Написать простое go-приложение, у которого реализуется несколько хендлеров.
2. Написать Dockerfile, чтобы поднять go-приложение в контейнере и отправить контейнер в docker hub на свой аккаунт.
3. Написать необходимые файлы для взаимодействия с абстракциями Kubernetes и, используя minikube и kubectl, которые зачастую устанавливаются вместе, поднять Kubernetes локально.

Написание go-приложения

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

Простейший сервис

Сформируем директорию **k8s-go-app**, где создадим **main.go**-файл:

```
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 запускается командой **go run main.go,** а командой **curl http://localhost:8080/** проверяется его работоспособность.

Версионирование

Добавим версионирование в наше приложение. Создадим файлик version/version.go:

```
package version

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

```
)
```

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

Добавим продвинутый роутинг

Для приложений, которые пишутся в продакшен, иногда требуется более продвинутый роутинг. Добавим в наш простой echo или любой другой сервис: gorilla/mux, httprouter и т. д. Затем добавим роутинг таким образом, чтобы перейти с одного фреймворка на другой стало проще.

Создадим рядом с **main.go** 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. Сейчас в структуре **Server** только одно поле — **port**. В дальнейшем эту структуру можно расширить, когда потребуется внедрить в сервис другие зависимости — базу данных, другой логгер и т. д.

В метод **Serve** передаётся **ctx**, который блочит выполнение, используя **<-ctx.Done()**. Сервис закончит работу, когда из канала **Done()** мы что-то прочитаем. С предыдущей реализации остался **handler**, который отрабатывает на «**/**», поменяв сигнатуру для **echo**. А ещё на всех других **path** сервис станет выдавать **404**. В **server/echo.go** мы добили код, поэтому теперь надо поменять старый код, чтобы всё работало. Для этого немного изменим 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()
}
```

Мы начали использовать пакет **server** и его функции. Добавили context-отмены, чтобы тушить сервис по сигналам OS. Теперь надо 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**, чтобы команда **curl http://localhost:8080** отработала, как ожидаем: получила в ответ **Hello, World! Welcome to GeekBrains!**

Добавим конфиг-файл

Сейчас у нас есть только один варьируемый параметр — **port.** Но даже его менять не очень удобно, надо лезть в код, искать подходящую строчку и менять значение. А если таких параметров больше, то сделать это становится сложнее. Более гибкий подход — хранить все конфиги в одном месте без привязки к языку программирования. В этом сервисе мы реализуем конфиги через переменные окружения, также можно использовать любой другой подход: хранить их в формате JSON/TOML/YAML/etc.

Создадим рядом с **main.go** файл-директорию **config/** с 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
}
```

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
```

Сейчас в файлике с конфигом есть 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()
}
```

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** передаём **cfg.Port.** Хороший шаг — логирование конфиг-параметров, это позволит быстрее найти ошибки.

Добавим heartbeat/version-хендлеры

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

heartbeat — это специальный маркер или сигнал, который показывает работоспособность нашего сервиса. Если сервис не отвечает на эту ручку, значит, есть проблемы.

version — ручка, которая отдаёт параметры сервиса. Зачастую это версия сервиса, номер билда, номер коммита и т. д.

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 — заглушка на будущее. **commit** можно уже отображать через команды:

```
git rev-parse HEAD
```

Подробнее — в следующем пункте.

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 почти разобрались. Для удобства осталось добавить Makefile. Makefile — это файл с командами alias. Чтобы постоянно не вводить длинных команд, создадим что-то наподобие alias, а затем будем вызывать команды через **make**. В нынешнем варианте добавим 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)
```

Запустим наш сервис через **make run**. А используя команду **curl localhost:8080/__version__,** получим примерно следующее:

```
{"build":"","commit":"245531c43ec628dbfa3ea86d461a95d277223466","version":"lates
t"}
```

Важно!

*Применение **version latest** в продакшене считается плохой практикой. Есть риск, что используется неправильная версия из «кеша» и возникнут проблемы, из-за которых вы не получите ожидаемого результата.*

Docker-контейнер

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

Создание Dockerfile

На предыдущем уроке мы рассмотрели процесс создания Dockerfile. Сегодня применим 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"]
```

Этот dockerfile мало чем отличается от dockerfile с предыдущего урока. Чтобы использовать конфиги, мы добавили проброс файлов ***.env** в контейнер, а сейчас прокидываем переменные окружения **GIT_COMMIT, VERSION, PROJECT** и используем при команде **go install** для нашей ручки **__version__**. 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 сделаем **make build_container** и **make run_container**, а затем зайдём в ручку **__version__**. Всё должно работать, как ожидается, и отдавать версию:

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

Важно!

*Применение **version latest** в продакшене считается плохой практикой. Есть риск, что используется неправильная версия из «кеша» и возникнут проблемы, из-за которых вы не получите ожидаемого результата.*

Dockerhub

Теперь отправим наш контейнер в dockerhub, чтобы использовать его при работе с Kubernetes. Для этого зарегистрируемся на сайте Docker Hub.

После регистрации выполним в терминале команду **docker login** и введём необходимые данные. Затем отправим наш готовый контейнер в dockerhub. Добавим следующую команду в **Makefile**:

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

Если всё прошло успешно, то на docker.com вы найдёте информацию о контейнере. Вместо **$(...)** вставьте свои данные.

Работа с Kubernetes

Теперь перейдём к самой интересной части — к работе с Kubernetes. Чтобы работать с k8s, надо установить minikube и kubectl.Обратите внимание, что зачастую kubectl устанавливается вместе с minikube.

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

Важно!

*Если вы пользователь MacOS, то вам следует выполнить команду **minikube start --vm=true.** Это связано с тем, что утилита minikube на MacOS не работает при некоторых командах. Вот обсуждение на эту тему. В основном это касается работы с Ingress — с ним возникают проблемы. Ещё не совсем так ведёт себя команда **minikube ip**. Но это менее критично, так как результат можно использовать для дальнейшей работы. Если вы всё-таки выполнили установку без флага **vm**, выполните **minikube delete** и снова 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

Начнём с Kubernetes Deployments, о котором шла речь в теоретической части. Для создания напишем 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
```

Разберём, что здесь написано.

1. **kind** — показывает, какой тип ресурса мы описываем. Сейчас опишем Deployment, затем — другие типы.
2. **metadata/name** — название ресурса k8s.
3. **replicas** — свойство объекта, показывающее, сколько реплик (экземпляров) подов можно запустить. Мы запускаем две реплики.
4. **strategy/type** — описывает стратегию развёртывания 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. Это свойство **RollingUpdate**. В нашем варианте с двумя репликами значение этого свойства указывает на то, что после завершения работы одного пода один ещё будет выполняться. Это делает приложение доступным во время обновления.
6. **RollingUpdate/maxSurge** — описывает максимальное число подов, которое можно добавить в развёртывание. Это свойство **RollingUpdate**. В нашем случае его значение равно 1, следовательно, при переходе на новую версию программы мы можем добавить в кластер ещё один под. Поэтому у нас появляется возможность запустить одновременно три пода.
7. **selector/matchLabels/app** — показывает, что развёртывание применится к подам с таким лейблом.
8. **template** — задаёт шаблон пода, который ресурс Deployment станет использовать для создания новых подов по заданной конфигурации.
9. **spec/containers/image** — название образа для контейнера. В нашем случае мы берём 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 эта проба проваливается — приложение перезапускается.
13. **containers/readiness** — определяет правила проверки, готов ли под к трафику. Если проба проваливается, контейнер удаляется из балансировщиков нагрузки сервиса. Для проверки проб в своём простом приложении мы используем одну и ту же ручку **__heartbeat__**.

Теперь выполним следующую команду:

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

Этой командой мы через утилиту **kubectl** создали объект Deployment k8s.

Output:

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

Теперь посмотрим на наши **deployments**, применив команду:

```
kubectl get deployments
```

Output:

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

Всё 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
```

Здесь тоже всё работает. Но чтобы посмотреть, что происходит на каждом поде, выполним такую команду:

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

logs — выводит в терминал логи пода.
-f — этот флаг выводит логи в стриминговом формате, то есть 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-prob
e/1.19","status":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-prob
e/1.19","status":200,"error":"","latency":1053,"latency_human":"1.053µs","bytes_
in":0,"bytes_out":0}
...
```

Видно, что вывел наш сервис с момента запуска в этом поде. Мы напечатали, в каком окружении работаем, а также наш конфиг-файл при запуске. Затем идут логи сервиса. При написании go-приложения мы добавили логирование запросов на сервис. Выше в **output** видно, как выполняется наша **liveness**-проба — отправка запроса в хендлер **__heartbeat__**.

Чтобы узнать о команде **logs** и её опциях больше, надо выполнить команду **kubectl logs --help.**

«Сходим» в наш сервис. Если мы не знаем host:port, то обычно используется подход port-forward.

Возьмём **NAME** пода и замапим (сделаем map) между локальным портом и портом пода, применив команду:

```
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.**

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

Service

Поды перезапускаются по различным причинам:

- ошибка на liveliness или на пробе readiness;
- поды прибиваются, 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 файле **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
```

Пройдёмся по незнакомым полям:

1. **spec/type** — описание типа сервиса. В нашем случае это NodePort, который даёт 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, определяющее, для каких подов применим сервис.

Теперь выполним эту конфигурацию, выполнив команду:

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

Output:

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

Сервис сформировался, теперь проверим это, выполнив команду:

```
 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 надо получить **host:port**, чтобы узнать, куда обращаться, применяя эту команду:

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

В output появится **host:port**, по которому мы сможем сделать запрос **curl http://host:port.**

Ingress

Рассмотрим ещё 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** создаётся с типом **ClusterIP**, такой сервис принимает запросы только внутри кластера.

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

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

Требуется 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 получим:

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

Напишем файл **ingress.yaml**:

```
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 надо выполнить следующую команду, чтобы использовать ingress:

```
minikube addons enable ingress
```

Важно!

*Если вы пользователь MacOS, то запускали **minikube**, применяя **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. Но если вы всё-таки выполнили установку без флага **vm**, то проведите **minikube delete** и снова команду с флагом.*

Выполним команду **minikube ip**.

Output:

```
192.168.64.2
```

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

Теперь, при выполнении команды **curl k8s-go-app.host,** мы получим наш давно знакомый результат:

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

Добавим в завершении раздела такие 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 появился к нему доступ. В заключение рассмотрим ещё 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

Рассмотрим ситуацию, при которой у нас что-то не работает при конфигурировании. Для начала удалим всё, что сделали, и попытаемся сконфигурировать это сначала, но с ошибкой.

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

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** выдаёт 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
```

В столбце **READY** значение равно 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. Попытаемся получить детальную информацию по поду, выполнив команду для одного из подов:

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

В output появится 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 надо внимательно посмотреть на конфигурацию данных проб и убедиться, что порты проставлены верно.

Проблема с портами возникает и в файле **service.yaml**. Убедитесь, что **targetPort** стоит подходящий. Внимательно отнеситесь и к пункту **selector/app** этого 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 стали обращаться к нему извне.

Практическое задание

1. Запустите в k8s любое приложение несколькими ручками с ресурсами Deployment, Service, Ingress. Добавьте две пробы: readiness и liveness. Важно, чтобы приложение умело 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. *Поднимите два сервиса в k8s и наладьте между ними общение любым способом.

Глоссарий

1. **Kubernetes (k8s)** — портативная расширяемая платформа с открытым исходным кодом для управления контейнеризированными рабочими нагрузками и сервисами, которая облегчает как декларативную настройку, так и автоматизацию. Это оркестратор контейнеров или система управления контейнерами.
2. **API Server** — клиентская часть панели управления (control plane).

3. **etcd** — распределённое и высоконадёжное хранилище данных в формате key-value, которое используется как основное хранилище всех данных кластера в Kubernetes.

4. **kube-scheduler** — отслеживает поды, не привязанные к конкретным нодам, и выбирает узел, на котором они работают.

5. **kube-controller-manager** — объект, запускающий процессы контроллеров. Контроллер — это управляющий цикл, который проверяет состояние кластера и вносит изменения, пытаясь привести текущее состояние кластера к желаемому.

6. **cloud-controller-manager** — запускает контроллеры, которые взаимодействуют с облачными провайдерами.

7. **kubelet** — следит за тем, чтобы контейнеры запускались в поде.

8. **kube-proxy** — сетевой прокси. Конфигурирует правила сети на узлах. Разрешает сетевые подключения к подам.

9. **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.

10. **Service** — это абстракция, которая определяет логическое множество подов и правила для получения доступа к ним.

11. **ClusterIP** — доступен только внутри кластера, тип по умолчанию.

12. **NodePort** — даёт каждому поду внешний IP, который доступен извне кластера.

13. **LoadBalancer** — добавляет балансировщик нагрузки на основе облачного провайдера.

14. **Ingress** — объект, который предоставляет данные к Services по HTTP/HTTPS, как показано на схеме. Это API-объект, который управляет внешним доступом к сервисам в кластере.

15. **Readiness probe** — это проверка готовности пода принимать трафик.

16. **Liveness probe** — это проверка «живучести» пода, которая перезапускает под при неудачном завершении.

17. **heartbeat** — это специальный маркер или сигнал, который показывает работоспособность сервиса.

18. **version** — ручка, которая отдаёт параметры сервиса.

19. **kind** — показывает, какой тип ресурса мы описываем. Сейчас опишем Deployment, затем — другие типы.

20. **metadata/name** — название ресурса k8s.
21. **replicas** — свойство объекта, показывающее, сколько реплик (экземпляров) подов можно запустить.
22. **strategy/type** — описывает стратегию развёртывания при переходе с текущей версии на новую.
23. **RollingUpdate/maxUnavailable** — показывает максимальное количество недоступных подов при выполнении обновления системы.
24. **RollingUpdate/maxSurge** — описывает максимальное число подов, которое можно добавить в развёртывание.
25. **selector/matchLabels/app** — показывает, что развёртывание применится к подам с таким лейблом.
26. **template** — задаёт шаблон пода, который ресурс Deployment станет использовать для создания новых подов по заданной конфигурации.
27. **spec/containers/image** — название образа для контейнера.
28. **spec/Containers/ImagePullPolicy** — определяет порядок работы с образами.
29. **containers/ports/containerPort** — должен совпадать с портом, на котором запущено go-приложение.
30. **containers/liveness** — определяет правила проверки, жив ли под.
31. **containers/readiness** — определяет правила проверки, готов ли под к трафику.
32. **logs** — выводит в терминал логи пода.
33. **-f** — этот флаг выводит логи в стриминговом формате.
34. **spec/type** — описание типа сервиса.
35. **ports/port** — принимает входящие запросы на этот порт и перенаправляет их на targetPort.
36. **selector/app** — название, определяющее, для каких подов применим сервис.

Дополнительные материалы

1. ARG, ENV через build-arg для Docker. Для мультистеджевых сборок — решение.
2. YouTube. Микросервисы в продакшен. 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».