# 9. Непрерывное обновление в Kubernetes. Deployment

В предыдущей главе мы выяснили, что декларативная модель Kubernetes, и селекторы, работающие с обычными метками, настолько удачно взаимодействуют, что позволяют нам несколькими движениями руки и парой меток быстро и удобно реализовать такие мощные и непростые в других системах действия, как “канареечные” и “сине-зеленые” развертывания.

Но Kubernetes не стал бы так популярен и не получил бы своего практически культового статуса, если бы в каждой операции не предоставлял еще более простые, а главное, декларативные, способы реализовать бесперебойную работу распределенной системы из множества компонентов и микросервисов. Объект для развертывания контейнеров Deployment, который мы до сих пор использовали просто для запуска контейнеров определенной версии и конфигурации, способен превратить нашу систему микросервисов в практически неуязвимую к простоям (*robust*), постоянно доступную (*HA, highly available*) и автоматически масштабируемую по требованию (*autoscaling*).

## Непрерывное обновление (rolling update)

Давайте используем наш сервис по получению времени time-service еще раз, в этот раз для обновления его версии и функциональности без применения ручной работы с метками. У нас после работы в прошлых главах есть две версии - 0.1.0 и 0.2.0. Для успеха эксперимента необходимо, чтобы они отвечали на запрос /time в разном формате - образы image контейнера в моем репозитории образов Docker Hub ivanporty/time-service содержат эти версии, но если вы хотите использовать образы, собранные вами локально, или из своего собственного репозитория, поменяйте имя образа в манифестах YAML.

Перед запуском обновления, если у вас остались предыдущие варианты сервиса из прошлых глав, в том числе “канареечные” или “сине-зеленые”, нужно будет их удалить и запустить один, первый вариант сервиса, версии 0.1.0:

…  
 # непосредственно описание контейнера в отсеке   
 spec:  
 containers:  
 - image: ivanporty/time-service:0.1.0  
 name: time-service

$ kubectl apply -f k8s/  
deployment.apps/time-service created  
service/time-service created

С таким применением развертывания мы уже хорошо знакомы - простое описание в YAML, и Kubernetes разворачивает для нас отсеки pods и запускает контейнеры в них, и поддерживает это желаемое состояние.

Но вот появляется новая, без сомнения лучшая версия сервиса 0.2.0, мы хотим перевести всю систему на новую версию, конечно же без остановки ее работы и перерыва в обслуживании драгоценных пользователей. Нет ничего проще - просто обновим версию сервиса, и передадим новое описание развертывания Deployment управляющей системе Kubernetes. Для наглядности скопируем описание YAML в папку update:

…  
 # обновленная версия контейнера в отсеке   
 spec:  
 containers:  
 - image: ivanporty/time-service:0.2.0  
 name: time-service

Проводим новое развертывание:

$ kubectl apply -f k8s/update/  
deployment.apps/time-service configured

Как мы видим, Kubernetes ответил нам, что развертывание time-service было сконфигурировано (configured), а не удалено и создано заново - обновление системы встроено в объекты Deployment, и останавливать и заново запускать отсеки и контейнеры не придется. Легко это проверить, как мы уже не однажды делали, с помощью обычного цикла curl к точке доступа сервиса прямо в терминале. Идеально это проверить на локальном кластере Docker Desktop с помощью прямого доступа к порту сервиса time-service (вы увидите его в листинге команды kubectl get services). В этом случае смена формата времени произойдет прямо на ваших глазах:

# только для Docker Desktop, запустите перед новым развертыванием  
$ while true; do curl localhost:[порт\_Docker\_Desktop]/time; sleep 1; done  
{"time":"2022-06-12 19:20:51.301901817 +0000 UTC m=+15.490737628"}  
{"time":"2022-06-12 19:20:51.425109475 +0000 UTC m=+15.613945291"}  
{"time":"12 Jun 2022"}  
{"time":"12 Jun 2022"}

В общем же случае, для любого кластера Kubernetes, мы всегда можем использовать переадресацию портов. Переадресация kubectl обычно оптимизирована и настроена напрямую на какой-либо отсек (pod), особенно в удаленном облаке, так что скорее всего после обновления переадресованный порт не ответит в новом формате или закончит работу с ошибкой. В этом случае для успеха нашего эксперимента при обновлении развертывания и его отсеков необходимо просто заново запустить переадресацию для сервиса time-service.

$ kubectl port-forward service/time-service 8080  
… в отдельном терминале, прямо перед новым развертыванием  
$ while true; do curl localhost:8080/time; sleep 1; done  
{"time":"2022-06-12 19:20:51.301901817 +0000 UTC m=+15.490737628"}  
…  
… новое развертывание, нужно перезапустить port-forward  
$ kubectl port-forward service/time-service 8080  
… тот же самый цикл curl переключается на новый формат  
{"time":"12 Jun 2022"}

Наш сервис, благодаря тому, что он по сути “нано”, а не “микро”, и написан на Go, запустился практически мгновенно, и как только управляющая система Kubernetes получила сигнал от отсека (pod), что процесс контейнера успешно запущен, она остановила предыдущую версию, а сервис Service остался неизменным - он по прежнему выбирает любые отсеки с метками app: time-service, и стал без всяких изменений со стороны пользователей сервиса передавать запросы к новой версии. Это и есть базовое непрерывное развертывание. Все что нам требуется - обновленная версия сервиса в новом образе контейнера с уникальной версией, и вставка этой версии в описание развертывания YAML!

## История обновлений. Откат к стабильным версиям.

Обновления для развертывания Deployment не только делают это по возможности непрерывно, еще они, как и следует надежной системе, легко позволяют нам отследить историю изменений и обновлений, и в случае проблемы с новыми версиями (проблемы с запуском, несовместимость с другими микросервисами), откатить микросервис на предыдущую, стабильную версию.

Получить историю обновлений можно командой rollout history:

$ kubectl rollout history deploy time-service  
deployment.extensions/time-service   
REVISION CHANGE-CAUSE  
1 <none>  
2 <none>

Как и ожидалось, у нас две “редакции” (revision), или версии нашего развертывания. Пока мы еще хорошо знаем, что это обновление версий образа контейнера, и какие это были версии, но как мы видим, Kubernetes не знает, что именно было причиной (change cause). Хорошей практикой является использование аннотации kubernetes.io/change-cause - именно ее значение и показано в списке редакций обновления. Если мы добавим её в описание YAML развертывания с новой версией (в директории update):

# Тип объекта  
…   
kind: Deployment  
# Метаданные нашего объекта, вложенный объект ObjectMeta  
metadata:  
 # список меток самого объекта Deployment  
 labels:  
 app: time-service  
 # аннотации объекта  
 annotations:  
 owner: ivan.porty@ipsoftware.ru  
 kubernetes.io/change-cause: Updated version to 0.2.0  
 name: time-service

Теперь список редакций нашего развертывания станет выглядеть намного приличнее:

$ kubectl rollout history deploy time-service  
deployment.extensions/time-service   
REVISION CHANGE-CAUSE  
1 <none>  
2 Updated version to 0.2.0

История обновлений хороша сама по себе, особенно в совокупности с декларативным управлением кластером, когда мы можем просто использовать систему контроля версий (Git) чтобы увидеть, когда, кто, и зачем обновлял и изменял систему и развертывание.

Но представим, что случилась ошибка. Как это часто бывает, один разработчик не понял другого, и третий программист решил, что сервис time-service выпущен в стабильной версии (1.0.0), и было бы просто глупо оставаться на старой версии. Обновление в Kubernetes - это быстро, просто и логично, файл с развертыванием был обновлен и передан в кластер:

…  
 # аннотации объекта  
 annotations:  
 ...  
 kubernetes.io/change-cause: Updated version to release 1.0!  
…  
 # обновленная версия контейнера в отсеке   
 spec:  
 containers:  
 - image: ivanporty/time-service:1.0.0  
 name: time-service

Сохраним обновленный объект обновления в директории broken, чтобы не забыть тот факт, что это “сломанная” версия:

$ kubectl apply -f k8s/update/broken  
deployment.apps/time-service configured

Если мы снова проверим работоспособность сервиса, то он продолжит работать - Kubernetes быстро поймет, что образа версии 1.0.0 просто нет, и не станет останавливать старый отсек с рабочей версией. Однако новая редакция развертывания будет теперь существовать в кластере, отчаянно пытаясь запустить новый отсек и контейнер из несуществующего образа (ошибка ImagePullBackOff), и достичь желаемого состояния:

$ kubectl get pods  
NAME READY STATUS RESTARTS AGE  
time-service-749f577cbc-mng5m 0/1 ImagePullBackOff 0 3m55s  
time-service-7ff577b7bd-kq7vb 1/1 Running 0 4m32s

Мы можем увидеть новую редакцию развертывания в истории изменений:

$ kubectl rollout history deploy time-service  
deployment.extensions/time-service   
REVISION CHANGE-CAUSE  
1 <none>  
2 Updated version to 0.2.0  
3 Updated version to release 1.0!

Ресурсы, особенно при использовании коммерческих провайдеров облака, дороги, и держать в кластере “сломанную” версию, несмотря на то что Kubernetes поддерживает общую работоспособность микросервиса, просто расточительно. Как поступить? Удалить нерабочий отсек pod? Развертывание Deployment снова и снова будет пытаться запустить его, так как мы передали это как желаемое состояние системы. Удаление развертывания будет совсем неудачной идеей - мы остановим непрерывную работу системы, пока будет запускаться новое развертывание.

Возможно, проблема с новой версией временная - например, нет доступа к репозиторию Docker Hub, и мы захотим вернуться к этой редакции позже. Развертывание Deployment способно помочь - мы можем вернуться к любой редакции, не удаляя никаких изменений! Давай посмотрим:

$ kubectl rollout undo deploy time-service  
deployment.extensions/time-service rolled back  
  
$ kubectl rollout history deploy time-service  
deployment.extensions/time-service   
REVISION CHANGE-CAUSE  
1 <none>  
3 Updated version to release 1.0!  
4 Updated version to 0.2.0  
  
$ kubectl get pods  
NAME READY STATUS RESTARTS AGE  
time-service-7ff577b7bd-kq7vb 1/1 Running 0 21m

Команда rollout undo откатывает изменения в развертывании к предыдущей версии. Кстати, если это необходимо, можно указать точный номер редакции, к которой необходимо откатиться (флаг --to-revision=...).

После отката к предыдущей редакции “сломанная” редакция 3 никуда не исчезла! Если мы исправим проблему, то всегда можем к ней вернуться. Что еще интереснее, редакция 2 стала теперь новым номером 4. Текущая редакция развертывания всегда является последней, и ее номер всегда увеличивается по порядку. В этом есть смысл - вы всегда знаете, сколько редакций было у обновления, и какие из них были заново использованы при откатах. Ну а сама редакция 2 исчезла из списка - это происходит в случае если к ней произошел откат, и она заняла более свежую строчку в истории развертываний.

Ну и наконец мы видим, что неработающий отсек pod исчез - предыдущей редакции развертывания он просто не нужен. Если же мы вернемся к редакции 3, он снова появится и попробует запустить контейнер с образом time-service:1.0.0 - Kubernetes будет приводить кластер к желаемому состоянию (desired state).

### Помощник развертывания - ReplicaSet

Kubernetes известен тем, что старается распределить обязанности по отдельным объектам, и сконцентрировать управление ими в одном месте - к примеру, мы управляем развертыванием с помощью объекта Deployment, однако непосредственно запуск контейнера и управление его ресурсами выполняет отсек Pod. Аналогично происходит с редакциями и масштабированием отсеков для развертываний - для каждой редакции развертывания создается свой объект ReplicaSet (набор экземпляров, или реплик, отсеков с контейнерами). В нашем случае с тремя редакциями развертывания мы получим три набора ReplicaSet:

$ kubectl get replicasets  
NAME DESIRED CURRENT READY AGE  
time-service-5f59fbf479 0 0 0 42m  
time-service-749f577cbc 0 0 0 41m  
time-service-7ff577b7bd 1 1 1 41m

Легко увидеть, какой их этих наборов является активным и управляет активными отсеками (pod). Именно суффикс объекта ReplicaSet используется для активным отсеков, иногда это полезно знать, в случае большого количества экземпляров. В общем случае вручную управлять наборами отсеков ReplicaSet нет особенного смысла - развертывания Deployment намного мощнее и удобнее, однако знать о них стоит - на каждую редакцию вашего развертывания будет создаваться новый набор отсеков, всегда готовый включиться в случае отката версий.

В общем и целом наборы ReplicaSet недороги и не требуют много ресурсов, однако если у вас много развертываний и микросервисов, и вы нечасто используете откат к предыдущим версиями (что в общем и целом возможно и просто с использованием предыдущей версии описания YAML из системы контроля версий, если используется декларативное управление), то можно ограничить их количество (и количество возможных откатов). По умолчанию история развертываний ограничена 10 редакциями, давайте уменьшим их до 5:

...  
# Описание собственно правил развертывания контейнера  
# Вложенный объект DeploymentSpec  
spec:  
 # Количество запущенных отсеков pods для масштабирования  
 replicas: 1  
 # максимальное количество редакций (revisions)  
 revisionHistoryLimit: 5  
...

## Стратегия непрерывного обновления

Все что мы использовали и только увидели, обновляя наше развертывание time-service, является результатом работы так называемой стратегии обновления (strategy), и по умолчанию используется непрерывное обновление (rolling update). Если мы укажем значение стратегии явно в нашем описании YAML, то получим следующее:

…  
# Вложенный объект DeploymentSpec  
spec:  
 # Количество запущенных отсеков pods для масштабирования  
 replicas: 1  
 # максимальное количество редакций (revisions)  
 revisionHistoryLimit: 5  
 # стратегия обновления  
 strategy:  
 type: RollingUpdate  
...

Стратегий наше развертывание поддерживает две:

* Recreate - развертывание заново. Удаляет все существующие отсеки (pods), и только потом создает новые. Как легко видеть, в случае проблем с новым развертыванием или контейнерами, микросервис сразу же перестает быть доступным. Эту стратегию конечно не стоит использовать в эксплуатации (production), но она может быть полезна, чтобы полностью стереть состояние предыдущих отсеков и контейнеров, например в кластере для разработки и отладки.
* RollingUpdate - знакомая нам стратегия непрерывного обновления по умолчанию. Запускает новые отсеки pods по одному, проверяет что они успешно запущены, и только после этого заканчивает работу отсеков с предыдущими версиями. Как мы уже видели, очень хороша для истории обновлений, легких откатов к прошлым версиям, и бесперебойной работы вашего сервиса.

Стратегия RollingUpdate имеет дополнительные настройки, крайне полезные в зависимости от типа микросервиса или приложения, и изменений в его логике и функциональности.

* maxSurge - насколько можно превысить желаемое (desired) количество отсеков pods. Если мы хотим, что наш сервис работал в 2 экземплярах, то значение
  + 1 - разрешает развертыванию создавать максимум три экземпляра, как правило два старой версии, один новой, потом два новой, один старой, и так далее.
  + 50% - делает все то же самое, но в случае автоматически масштабируемого развертывания учитывает, сколько отсеков работает на данный момент, вместо использования точного числа.
* maxUnavailable - дополняющая настройка к первой, имеющая такие же возможные абсолютные и процентные значения. Она указывает, насколько можно уменьшить количество желаемых отсеков, уменьшая доступность сервиса с целью экономии времени и ресурсов.

Проще всего все увидеть на примере с несколькими экземплярами микросервиса, например тремя. Давайте удалим все наши предыдущие развертывания (запустив команду kubectl delete -f k8s/ из директории примера time-service), и попробуем следующее развертывание для первой версии 0.1.0:

…  
spec:  
 # Количество запущенных отсеков pods для масштабирования  
 replicas: 3  
 # максимальное количество редакций (revisions)  
 revisionHistoryLimit: 5  
 # стратегия обновления  
 strategy:  
 type: RollingUpdate  
 rollingUpdate:  
 maxSurge: 1  
 maxUnavailable: 2  
 selector:  
 matchLabels:  
 app: time-service  
 # описание шаблона для создания новых отсеков   
 template:  
 metadata:  
 # список меток для нового отсека  
 labels:  
 app: time-service  
 # обновленная версия контейнера в отсеке   
 spec:  
 containers:  
 - image: ivanporty/time-service:0.1.0  
 name: time-service

Как видно, у нас будет три экземпляра нашего микросервиса, а непрерывное обновление может максимально превысить количество экземпляров на 1 (в общем 4), а уменьшить максимум на два (минимум 2 работающих отсека). Теперь передадим управляющему серверу Kubernetes такое же обновление, но с новой версией 0.2.0, и посмотрим что произойдет с нашими отсеками. Новые манифесты для развертываний для простоты и наглядности будут храниться в отдельных папках update/rolling-0.1.0/0.2.0. В реальных системах мы бы использовали метки (tag) контроля версий Git.

Чтобы увидеть картину обновления развертывания в действии, используем флаг -w[atch], доступный для многих команд kubectl, чтобы увидеть все изменения в непрерывном потоке по мере обновления отсеков:

# удаляем прошлые версии  
$ kubectl delete -f k8s/  
…  
$ kubectl apply -f k8s/update/rolling-0.1.0  
…  
$ kubectl apply -f k8s/update/rolling-0.2.0  
...  
$ kubectl get pods -w  
 - отсеки первого развертывания  
NAME READY STATUS RESTARTS AGE  
time-service-5f59fbf479-9v4vw 1/1 Running 0 25s  
time-service-5f59fbf479-chxvp 1/1 Running 0 43s  
time-service-5f59fbf479-sg995 1/1 Running 0 25s  
- момент начала второго обновления - создан 1 новый отсек,  
 и остановлена работы двух существующих отсеков.  
time-service-7ff577b7bd-78zv4 0/1 Pending 0 0s  
time-service-5f59fbf479-9v4vw 1/1 Terminating 0 6m47s  
time-service-5f59fbf479-sg995 1/1 Terminating 0 6m47s  
- создается еще два новых отсека, так как два остановлено  
time-service-7ff577b7bd-6td9f 0/1 Pending 0 0s  
time-service-7ff577b7bd-lg8db 0/1 Pending 0 0s  
- запуск контейнеров и процесса в трех новых отсеках  
 остается только один работающий отсек старой версии!  
time-service-7ff577b7bd-78zv4 0/1 ContainerCreating 0 0s  
time-service-7ff577b7bd-6td9f 0/1 ContainerCreating 0 0s  
time-service-7ff577b7bd-lg8db 0/1 ContainerCreating 0 0s  
- первый отсек с новой версией запущен  
time-service-7ff577b7bd-6td9f 1/1 Running 0 2s  
- в этот момент у нас 4 отсека, 1 старой версии, 1 новой  
 и два в процессе запуска контейнеров. Так как новая версия  
 запущена, остановлен последний отсек со старой версией  
time-service-5f59fbf479-chxvp 1/1 Terminating 0 7m7s  
- три отсека с новой версией - желаемое состояние!  
time-service-7ff577b7bd-78zv4 1/1 Running 0 3s  
time-service-7ff577b7bd-lg8db 1/1 Running 0 3s

Если мы посмотрим внимательно, то увидим, что непрерывное обновление развертывания никогда не превышает 4 экземпляров, указанных нами в качестве параметров для стратегии обновления. Однако, как мы видим, обновление очень быстро останавливает отсеки, как только отсеки с новой версией сообщают управляющему kubelet о том, что процесс запущен. Если в процессе обновления вы начнете запрашивать доступ к сервису (к точке доступа /time или /nanotime), при определенной удаче можно увидеть, что обновленный сервис не ответит.

Все дело в том, что нам не хватает проверки готовности сервиса к работе (readiness probe), которую мы уже добавили для микросервиса weekend-service. Добавим её для идеально непрерывного развертывания в описание развертывания версии 0.2.0. Для простоты мы не станем использовать новую точку доступа HTTP, а просто используем все тот же вызов /time:

…  
 # обновленная версия контейнера в отсеке, вместе с проверкой готовности  
 spec:  
 containers:  
 - image: ivanporty/time-service:0.2.0  
 name: time-service  
 readinessProbe:  
 httpGet:  
 path: /time  
 port: 8080  
 periodSeconds: 2  
 initialDelaySeconds: 5  
 failureThreshold: 2  
 successThreshold: 1

Снова проведя тот же эксперимент (удалите развертывание, разверните версию 0.1.0 из папки update, а затем 0.2.0), вы увидите, что отсеки останавливаются на мгновение позже, только в момент готовности новых к работе. Проверка готовности (readiness check) - совершенно обязательное условие безотказных, непрерывных обновлений Kubernetes. Любой микросервис или монолит, работающий в кластере, должен определять такую проверку.

Если нас больше заботит качество сервиса, а количество ресурсов достаточно велико, мы можем подстраховаться таким образом:

# стратегия обновления  
 strategy:  
 type: RollingUpdate  
 rollingUpdate:  
 maxSurge: 100%  
 maxUnavailable: 0

В этом случае мы говорим, что готовы удвоить количество отсеков (100%), и не разрешаем уменьшать количество работающих отсеков (maxUnavailable: 0).

## Автоматическое масштабирование

Как вы помните, еще в первой, обзорной главе про Kubernetes мы использовали автоматическое масштабирование нашего развертывания, впечатляющее оружие для автоматического управления кластером, способное сделать его действительно постоянно доступным и приспособленным к растущим нагрузкам, особенно если они случаются случайно, резко растут, а затем снижаются (например, просмотр видео через стриминг-сервис по вечерам, или выход новой версии популярной игры) - в таких случаях ручное масштабирование не настолько эффективно, и правильно рассчитать нагрузку заранее довольно сложно. Горизонтальное автоматическое масштабирование (horizontal pod autoscaling) считывает метрики ваших отсеков, их использование процессора и памяти, и при возрастании нагрузки увеличивает количество экземпляров сервиса под нагрузкой.

Автоматическое масштабирование не всегда доступно в любом кластере Kubernetes по умолчанию, хотя в коммерческих облаках, таких как Google Kubernetes Engine или Amazon EKS, оно почти всегда будет доступно. Если вы экспериментируете с Kubernetes на локальном кластере или запустили собственный кластер, нужно будет развернуть дополнительный сервис metrics-server, собирающий метрики кластера и предоставляющий их в едином виде для всех пользователей. Детали можно найти на сайте данного сервера в GitHub, ну а для локального кластера minikube метрики можно включить через расширение (addon) следующей командой:

$ minikube addons enable metrics-server

Проверить, что в кластере доступны метрики загрузки процессора и памяти, позволяет хорошо знакомая по Linux команда top, только теперь в составе kubectl. Она показывает текущий статус ресурсов для узлов кластера, например так:

$ kubectl top node  
NAME CPU(cores) CPU% MEMORY(bytes) MEMORY%   
minikube 301m 15% 1344Mi 69%

Ее же можно использовать и для просмотра загрузки отдельных отсеков (pods):

$ kubectl top pods  
NAME CPU(cores) MEMORY(bytes)   
time-service-68978c4db5-tn5nw 0m 5Mi

Теперь, зная что метрики загрузки кластера и работающих в отсеках приложений нам доступны, мы можем настроить горизонтальное автоматическое масштабирование. Как мы помним, в мире Kubernetes все представляет собой объект, и описание правил автоматического масштабирования - не исключение. Новый объект - это новый файл с неизвестным нам форматом YAML, но мы помним, что мы всегда можем прибегнуть к помощи команды kubectl с флагом --dry-run=client, чтобы получить заготовку описания нашего объекта в формате YAML или JSON. Второй вариант - использовать редактор со встроенной поддержкой схемы объектов Kubernetes, который подскажет вам правильные поля и их значения. Попробуем знакомый нам вызов:

$ kubectl autoscale deployment/time-service --min=1 --max=3 --cpu-percent=80 --dry-run=client -o yaml

Вот что мы получим в результате - объект HorizontalPodAutoscaler, созданный как точный аналог результата вызванной нами команды, только на этот раз в предпочтительном декларативном варианте, который всегда будет проще отследить через систему контроля версий и использовать, чтобы попросить Kubernetes привести систему в желаемое состояние:

apiVersion: autoscaling/v1  
kind: HorizontalPodAutoscaler  
metadata:  
 creationTimestamp: null  
 name: time-service  
spec:  
 maxReplicas: 3  
 minReplicas: 1  
 scaleTargetRef:  
 apiVersion: apps/v1  
 kind: Deployment  
 name: time-service  
 targetCPUUtilizationPercentage: 80

Описание нашего объекта содержит именно то, что мы и хотели получить вызовом команды kubectl autoscale. Можно задать нижний и верхний порог количества экземпляров, и описать, для каких ресурсов (scaleTargetRef), и как именно будет вычисляться необходимое количество экземпляров (targetCPUUtilizationPercentage) - в нашем случае мы указываем, что следим за использованием процессора для развертывания time-service, и если использование процессора одним из запущенных экземпляров превышает 80% от доступного максимума, необходимо добавить еще один экземпляр. Сервис Service для развертывания time-service незамедлительно включит новый экземпляр в свой список отсеков (после того, как будет запущен), и начнет отправлять ему запросы, снизив среднюю нагрузку. Обратное будет сделано при падении загрузки менее 80% - по прошествии некоторого времени ненужные экземпляры будут остановлены.

Остается только передать правила масштабирования управляющей системе Kubernetes:

$ kubectl apply -f k8s/autoscale/  
horizontalpodautoscaler.autoscaling/time-service created

Аналогичным образом можно настроить масштабирование при превышении использования памяти. Как проверить, что автоматическое масштабирование работает? На мощном кластере, и таком миниатюрном примере, как наш time-service, превысить нагрузку непросто. Это будет хорошим упражнением - например, можно указать низкий порог нагрузки (10% загрузки процессора), или придумать тестовый метод, дающий высокую нагрузку - например, считать некую сложную математическую формулу.

## Проверка жизнеспособности (liveness)

Мы только что добавили *проверку готовности* (readiness check) к нашему микросервису time-service, и выяснили, что эта проверка является абсолютно необходимой для создания системы микросервисов, способной обновляться и масштабироваться без простоев и с большой надежностью. В Kubernetes есть еще одна проверка, называемая *проверкой жизнеспособности* (liveness check). Давайте рассмотрим, в чем разница и зачем нужна проверка жизнеспособности.

* По умолчанию система управления Kubernetes считает, что микросервис, работающий в отсеке (pod) готов к работе и обработке запросов, как только получен образ (image) контейнера, контейнер развернут в отсеке, и процесс, указанный в этом образе (обычно командой CMD), успешно запущен. Конечно, для подавляющего большинства сервисов это не так - они запускают встроенные веб-сервера для обработки запросов, подключаются к другим сервисам, настраивают системы журналов logs, и могут производить большое количество еще более сложных шагов для инициализации системы. Обрабатывать запросы мгновенно они не могут.
* Таким образов, проверка готовности просто обязана существовать во всех микросервисах высоко доступной (*HA, highly available*), надежной системы. Только когда микросервис подтвердит готовность к обработке запросов, управление Kubernetes начнет направлять их к нему в контейнер (работающий в отсеке). В противном случае при обновлении и масштабировании системы неизбежны отказы в обслуживании, возможно каскадные (ошибка передается через многие микросервисы).
* Сложные микросервисы с большим количеством логики и потребностей в ресурсах могут продолжать работать (процесс Linux активен), однако на самом деле обрабатывать запросы уже не могут. Как тут не вспомнить большие Java-приложения с большими запросами к памяти - они могут “застыть” на десятки секунд, если нужно очистить “кучу” памяти (heap), и перестанут отвечать на запросы. Именно для таких сложных микросервисов нужна проверка жизнеспособности. В идеале она точно определяет, что приложение все еще активно работает, и делает это быстро.

Для наших микроскопических сервисов проверка жизнеспособности вряд ли актуальна, но представим, что мы переносим Java-монолит в кластер Kubernetes, постепенно разбивая его на микросервисы, начиная с очень больших частей. В этом варианте нам точно понадобится проверка жизнеспособности, к примеру, чтобы избежать проблем с памятью. Описывается она также в объекте Deployment, и точно таким же способом, как и проверка готовности, только с другим названием. Для примера добавим ее в наше развертывание для time-service:0.2.0:

spec:  
 containers:  
 - image: ivanporty/time-service:0.2.0  
 name: time-service  
 readinessProbe:  
 …  
 livenessProbe:  
 httpGet:  
 path: /time  
 port: 8080  
 periodSeconds: 2  
 initialDelaySeconds: 5  
 failureThreshold: 2  
 successThreshold: 1

Как мы видим, проверка жизнеспособности здесь идентична проверке готовности и объявлена точно так же, но в большинстве случаев она должна использовать другой алгоритм, чтобы определить, “жив” ли микросервис. Проверка готовности отвечает, способен ли микросервис обрабатывать запросы. Проверка жизнеспособности в идеале наблюдает за обработкой запросов со стороны, например, из другого потока (thread), или даже другого процесса, и выясняет, не возникла ли неисправимая ситуация, к примеру, критическая задержка обработки запросов из-за переполнения памяти или взаимная блокировка (deadlock), если используется параллельная обработка информации.

## Резюме

* Развертывания Deployment в Kubernetes обладают встроенной мощной поддержкой обновления версии ваших микросервисов без перерывов в обслуживании своих клиентов. В простейшем случае необходимо просто обновить версию или метку образа контейнера с микросервисом, и все будет сделано для нас автоматически.
* Развертывания поддерживают историю обновлений и легкий откат к предыдущим версиям в случае проблем с новой версией. За кулисами историю обновлений обеспечивают объекты ReplicaSet.
* Стратегия непрерывного обновления настраивается с помощью дополнительных параметров - мощное оружие для разнообразных типов обновлений, микросервисов и типов нагрузки.
* Масштабировать развертывания можно как вручную, просто указывая количество экземпляров работающего микросервиса или приложения, так и автоматически, указывая, при каких параметрах загрузки процессора и памяти необходимо добавить дополнительные экземпляры.
* Для создания высоко доступной (*HA, highly available*), надежной системы микросервисов все они должны реализовать и описать в своих объектах Deployment *проверку готовности* к приему запросов (readiness check).
* Для сложных микросервисов, зависящих от памяти, сложных данных, параллельных вычислений, в идеале должны существовать хорошо продуманная *проверка жизнеспособности* (liveness check).

В этой главе мы уже вплотную приближаемся к стороне управления, а не разработки, системы (operations, в концепции DevOps). Разрабатывая приложение как набор микросервисов под управлением Kubernetes, мы должны осознавать, что оно будет чрезвычайно динамично, работать во множестве экземпляров, в идеале не обладать состоянием (stateless), и мы видим как все это складывается в единую картину при развертываниях. Если же вы хотите еще сильнее погрузиться в сторону управления системы (или примерить на себя роль оператора), можно посоветовать книги о том, как это делается в больших компаниях. К примеру, книги о SRE (site reliability engineering) в компании Google доступны на русском языке:

* “Site Reliability Workbook. Практическое применение”
* “Site Reliability Engineering. Надежность и безотказность как в Google”