# 8. Метки и аннотации Kubernetes. «Канарейки». Service Mesh

В прошлом разделе мы рассмотрели первые объекты Kubernetes, развертывания Deployment и сервисы Service. В них мы указали простейшие *метки* (labels), которые в нашем начальном случае совпадали для двух созданных объектов и позволяли каким-то образом соединить развертывания и сервисы в единую работающую систему.

Метки — это произвольный набор текстовых ключей и их значений, который позволяет производить поиск любого объекта среди всех объектов одного типа в кластере Kubernetes или объединять объекты по какому-либо признаку. Можно представить их в виде бирок на чемоданах в аэропорту — именно они позволяют грузовой службе отличить практически одинаковые чемоданы друг от друга и выбрать те, что направляются в один аэропорт.

Основная задача меток в кластере — обеспечить эффективную организацию объектов по категориям (которые в общем случае зависят от нужд вашего конкретного приложения) и обеспечить работу селекторов (selector), выбирающих набор объектов, например, для сервисов Kubernetes.

Обычно метки применяются для следующих целей:

* «канареечные» выпуски обновлений (canary releases). Красивое название связано с историей о том, как шахтеры брали с собой в забой чрезвычайно чувствительных к опасным газам канареек, чтобы узнать об опасности заранее. В случае же нового сервиса или обновления небольшая часть пользователей или сервисов-клиентов переключается на новую версию, чтобы в реальных условиях проверить работоспособность новых изменений. Очень эффективная техника, которую легко реализовать с помощью меток;
* безопасное переключение между версиями микросервиса в одном кластере. Такое переключение еще называется «сине-зеленым» развертыванием (blue-green deployment). В случае нашего примера time-service мы можем захотеть обновить его и добавить новую точку доступа и новый формат времени, и перейти на новую версию мы хотим сразу же, для всех клиентов и всех экземпляров микросервиса. С помощью меток мы можем создать новое развертывание с меткой version: green, и как только оно будет готово (все отсеки pods развернуты и запущены), переключить сервис Service на новые отсеки (pods) и контейнеры в них, имеющие метку green;
* разбиение вычислительных потоков или связанных с ними потоков данных (partitions). Если алгоритм данных подразумевает разбиение данных и параллельную обработку, вместо управления этим процессом в коде можно использовать одинаковые сервисы, а разбиение данных и их перенаправление устроить с помощью меток и сервисов.

Ваши конкретные нужды могут отличаться от этих основных случаев, но основная задача меток в кластере — гибко разделить одинаковые объекты на категории и управлять ими и их потоками данных, не усложняя код и конфигурацию микросервисов и приложений.

Одно из основных, самых фундаментальных применений меток — сервисы Service, с которым мы уже знакомы. Используя указанный при его создании набор меток, сервис Service выбирает набор отсеков и направляет к одному из них (обычно случайно выбранному) очередной запрос от клиента.

И еще — имя метки не должно превышать 63 символов, ее значение — 253. В общем случае лучше не использовать нестандартные символы. Детали легко найти в документации.

## Метки на практике. «Канареечное» развертывание

Давайте попробуем применить метки на практике. Запустим в своем кластере сразу две версии time-service, одну классическую, созданную нами в первых главах, а вторую обновленную. Представим, однако, что у нас сотни или тысячи существующих пользователей, чье доверие, как хорошо известно, очень тяжело завоевать и проще простого потерять. Цена ошибки велика, а найти все ошибки в сложной распределенной системе из микросервисов весьма непросто и дорого.

Идея «канареечного» развертывания заключается в том, чтобы в приложении одновременно сосуществовали две версии одной и той же функциональности. Как правило, проверенная, предыдущая версия преобладает, а новая версия обслуживает намного меньшее количество запросов. Классическим запуском приложений на выделенных серверах сделать подобное непросто, а управлять еще сложнее. Но, как оказывается, самые простые метки Kubernetes помогут нам развернуть первое элементарное «канареечное» развертывание.

Представим, что мы обновили сервис time-service следующим образом:

…  
  
func main() {  
 log.Print("Начало работы сервиса time-service")  
  
 http.HandleFunc("/time", serveTime)  
 http.HandleFunc("/nanotime", serveNanoTime)  
 log.Fatal(http.ListenAndServe("0.0.0.0:8080", nil))  
}  
  
func serveTime(w http.ResponseWriter, r \*http.Request) {  
 log.Print("Вызов функции serveTime()")  
 var serverTime Time  
 // используйте этот код, чтобы собрать версию сервиса 0.1.0  
 // serverTime.Time = time.Now().String()  
 // новый формат времени для версии 0.2.0  
 serverTime.Time = time.Now().Format("02 Jan 2006")  
 json.NewEncoder(w).Encode(serverTime)  
}  
  
...

По сути ничего не поменялось, только более привычный нам формат даты в методе Format(), которую мы отправляем при запросе ресурса /time вместо чрезмерно неудобного результата функции String(). Все шаги по построению приложения и упаковке его в образ контейнера будут неизменны, мы изменим лишь версию, увеличив ее на следующую минимальную 0.2.0:

$ docker build . -t {ваша\_учетная\_запись\_Docker}/time-service:0.2.0  
$ docker push {ваша\_учетная\_запись\_Docker}/time-service:0.2.0

К этому моменту в нашем репозитории Docker Hub хранятся две версии time-service. Давайте развернем в нашем кластере вторую версию, только укажем для наших объектов дополнительную метку, чтобы отличить отсеки от предыдущей, стабильной версии. Для «канареечных» версий часто используют метку release: canary.

*Внимание*: для того чтобы наш эксперимент удался, версии сервиса 0.1.0 и 0.2.0 должны возвращать разные строки при вызове адреса /time. Если вы случайно обновите версию 0.1.0 (соберете образ контейнера со старой меткой), закомментируйте новый формат и снова соберите версию 0.1.0. В реальной разработке мы, конечно, сохранили бы старый выпуск под определенным ярлыком (tag) в контроле версий Git, но для простоты и наглядности в нашем маленьком примере разница между версиями сохранена в комментариях.

Начнем с развертывания Deployment. Откроем описание развертывания в YAML и создадим новый вариант, назвав его k8s-time-deploy-canary.yaml и сохранив во вложенной директории k8s/canary:

# Версия программного интерфейса Kubernetes  
apiVersion: apps/v1  
# Тип объекта  
kind: Deployment  
# Метаданные нашего объекта, вложенный объект ObjectMeta  
metadata:  
 # список меток “канареечного” развертывания  
 labels:  
 app: time-service  
 release: canary  
 name: time-service-canary  
# Описание собственно правил развертывания контейнера  
# Вложенный объект DeploymentSpec  
spec:  
 # Количество запущенных отсеков pods для масштабирования  
 replicas: 1  
 selector:  
 matchLabels:  
 app: time-service  
 release: canary  
 # описание шаблона для создания новых отсеков   
 template:  
 metadata:  
 # список меток для “канареечных” отсеков pods  
 labels:  
 app: time-service  
 release: canary  
 # непосредственно описание контейнера в отсеке   
 spec:  
 containers:  
 - image: ivanporty/time-service:0.2.0  
 name: time-service

Единственное отличие нашего нового развертывания — новая метка в шаблоне spec.template и новая версия образа для запускаемых контейнеров — это означает, что всем создаваемым в этом развертывании отсекам pods будет присваиваться дополнительная метка release: canary.

Давайте развернем оба развертывания одновременно — конечно же, декларативно, просто указав директорию со всеми нашими объектами в YAML:

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

Мы видим, что теперь у нас два развертывания — стабильное, с проверенной версией образа контейнера и сервиса, и новое, «канареечное», для «притирки» и сбора информации о том, как обновленный микросервис работает в реальных условиях. Однако сервис Kubernetes у нас по-прежнему один, и он выбирает все отсеки с отметками app: time-service, то есть любые отсеки, неважно, есть ли у них метка release: canary или нет.

…  
 # по этим меткам идет поиск отсеков, куда отправляются запросы  
 selector:  
 app: time-service

Между прочим, это не что иное, как простое «канареечное» развертывание (canary deployment)! Доступ через сервис будет выбирать любые отсеки и контейнеры с метками app: time-service в случайном порядке, и мы получим возможность сравнить, как работает новая функциональность, полностью не отказываясь от старой проверенной версии. С другой стороны, мы имеем полный контроль над тем, сколько «канареечных» контейнеров работает в нашем кластере, и в любой момент можем изменить их количество (просто поменяв число экземпляров replicas) или, в случае обнаруженной ошибки, немедленно удалить «канареечное» развертывание и откатить систему в стабильное состояние.

Проверить, как это работает, весьма просто — давайте запрашивать time-service в простом цикле, и вот что мы увидим (здесь команда на основе кластера Kubernetes Docker Desktop):

$ while true; do curl localhost:[порт\_NodePort]/time; sleep 1; done  
{"time":"06 Jun 2022"}  
{"time":"2022-06-06 17:27:12.260209446 +0000 UTC m=+85.041021065"}  
{"time":"2022-06-06 17:27:12.384187859 +0000 UTC m=+85.164999497"}  
{"time":"06 Jun 2022"}  
{"time":"2022-06-06 17:27:12.638330589 +0000 UTC m=+85.419142202"}  
{"time":"2022-06-06 17:27:12.76055814 +0000 UTC m=+85.541369750"}  
…

Как мы видим, сервис случайным образом отвечает или стабильной существующей версией, или новым форматом данных «канареечного» выпуска. Кстати, доступ к сервису мы получили по номеру порта NodePort — напрямую, так как для развертывания был использован локальный кластер Docker Desktop.

В случае облачного кластера или других локальных кластеров (minikube, kind) мы получим доступ как обычно, с помощью перенаправленного порта (kubectl port-forward). Однако вы увидите только один из вариантов вывода сервиса time-service. Это не означает, что канареечное развертывание не работает. Дело будет в том, что сетевые управляющие компоненты Kubernetes откроют соединение к одному отсеку (pod), это значительно повышает эффективность сети. В этом случае, чтобы увидеть «канареечный» сервис в действии, придется открыть сетевые соединения к нему с разных машин, поэтому для простоты эксперимента лучше воспользоваться прямым вызовом порта локального кластера Docker Desktop.

Настраивая экземпляры replicas, мы можем уменьшить или увеличить долю «канареечной» версии в работе системы. Более профессиональный вариант применить «канареечное» развертывание — объект для входящего интернет-трафика Ingress, используя его, возможно назначить отдельные развертывания на выделенные маршруты HTTP. Вы сможете найти детали в документации этого объекта. Ну а самое мощное оружие для подобных развертываний — сетки микросервисов (service mesh); мы узнаем про них чуть позже в этой же главе.

## «Сине-зеленое» развертывание

Что еще можно легко сделать с метками? Представим себе, что мы хотим очень аккуратно перевести свой микросервис на новую версию, предварительно проверив, что все отсеки и контейнеры готовы к работе, и только потом переключить всю систему на новую версию. Запустим в кластере совершенно отдельную, новую версию time-service и для простоты назовем ее «зеленой». Все микросервисы для этой версии будем помечать довольно часто использующейся меткой version: green.

Создадим развертывание Deployment в файле k8s/blue-green/k8s-time-deploy-green.yaml примерно так же, как с «канареечным» развертыванием, только используем другой набор меток и назовем это развертывание time-service-green (мы помним, все объекты одного типа должны иметь уникальное название):

# Версия программного интерфейса Kubernetes  
apiVersion: apps/v1  
# Тип объекта  
kind: Deployment  
# Метаданные нашего объекта, вложенный объект ObjectMeta  
metadata:  
 # список меток “зеленого” развертывания Deployment  
 labels:  
 app: time-service  
 version: green  
 name: time-service-green  
# Описание собственно правил развертывания контейнера  
# Вложенный объект DeploymentSpec  
spec:  
 # количество запущенных отсеков pods для масштабирования  
 replicas: 1  
 selector:  
 matchLabels:  
 app: time-service  
 version: green  
 # описание шаблона для создания новых отсеков   
 template:  
 metadata:  
 # список меток “зеленых” отсеков  
 labels:  
 app: time-service  
 version: green  
 # новая версия 0.2.0   
 spec:  
 containers:  
 - image: ivanporty/time-service:0.2.0  
 name: time-service

Новые отсеки будут иметь метку version: green. Если мы развернем новую версию микросервиса, но не поменяем сервис Kubernetes, то вновь получим «канареечное» развертывание, так как отсеки и старого, и нового развертывания будут подходить по критерию поиска (app: time-service). Нам надо полностью отделить стабильную версию сервиса от новой (“зеленой»). Назовем стабильную версию «синей» и добавим «синие» метки в наше старое развертывание и сервис. Все изменения поместим для простоты в отдельные директории (k8s/blue-green):

# Версия программного интерфейса Kubernetes  
apiVersion: apps/v1  
# Тип объекта  
kind: Deployment  
# Метаданные нашего объекта, вложенный объект ObjectMeta  
metadata:  
 # список меток “синего” развертывания Deployment  
 labels:  
 app: time-service  
 version: blue  
 name: time-service  
# Описание собственно правил развертывания контейнера  
# Вложенный объект DeploymentSpec  
spec:  
 # количество запущенных отсеков pods для масштабирования  
 replicas: 1  
 selector:  
 matchLabels:  
 app: time-service  
 version: blue  
 # описание шаблона для создания новых отсеков   
 template:  
 metadata:  
 # список меток “синих” отсеков  
 labels:  
 app: time-service  
 version: blue  
 # старая версия 0.1.0   
 spec:  
 containers:  
 - image: ivanporty/time-service:0.1.0  
 name: time-service

Теперь наша стабильная версия развертывания имеет метки version: blue, поменяем наш сервис, чтобы он искал только «синие» отсеки:

# Версия программного интерфейса Kubernetes  
apiVersion: v1  
# Тип объекта  
kind: Service  
metadata:  
 labels:  
 app: time-service  
 version: blue  
 name: time-service  
spec:  
 # список портов. Дополнительно можно указать протокол  
 ports:  
 - port: 8080  
 # по этим меткам идет поиск “синих” отсеков  
 selector:  
 app: time-service  
 version: blue  
 # тип сервиса. В облаке можно использовать LoadBalancer  
 type: NodePort

Обновим наш кластер и сделаем стабильную, старую версию доступной только через «синие» метки. Внимание: при изменении меток объектов Kubernetes данные объекты сначала надо удалить, а только потом добавить в кластер заново! Мы так и поступим с нашим развертыванием и сервисом:

$ kubectl delete deploy/time-service  
$ kubectl delete svc/time-service  
…  
$ kubectl apply -f k8s/blue-green/blue/  
deployment.apps/time-service created  
service/time-service created

Теперь мы можем запустить обновленную «зеленую» версию развертывания, не опасаясь того, что к ней будут попадать реальные запросы:

$ kubectl apply -f k8s/blue-green/green/k8s-time-deploy-green.yaml   
deployment.apps/time-service-green created

Наш сервис теперь не будет обращать внимания на параллельно работающую «зеленую» версию, так как настроен на «синюю» версию. Проверим это классическим способом, перенаправив порт сервиса на свою локальную машину (в случае Docker Desktop можно обратиться напрямую к порту NodePort):

$ kubectl port-forward svc/time-service 8080  
… в новом терминале  
$ while true; do curl 0.0.0.0:8080/time; sleep 1; done  
{"time":"2022-06-07 22:26:40.438451912 +0000 UTC m=+87.838016061"}  
{"time":"2022-06-07 22:26:41.469423227 +0000 UTC m=+88.868987392"}  
...

Итак, у нас в кластере запущены полностью раздельные две версии микросервиса time-service. Мы в любой момент можем переключить сервис с «синих» на «зеленые» метки. Напишем YAML для «зеленого» варианта сервиса (он будет иметь то же имя, просто другой селектор, так что клиенты могут продолжать использовать этот сервис без каких-либо изменений по тому же DNS-имени):

# Версия программного интерфейса Kubernetes  
apiVersion: v1  
# Тип объекта  
kind: Service  
# Метаданные нашего объекта, вложенный объект ObjectMeta  
metadata:  
 labels:  
 app: time-service  
 version: green  
 name: time-service  
spec:  
 # список портов. Дополнительно можно указать протокол  
 ports:  
 - port: 8080  
 # список меток “зеленых” отсеков для отправки запросов  
 selector:  
 app: time-service  
 version: green  
 # тип сервиса. В облаке можно использовать LoadBalancer  
 type: NodePort

Торжественно (и немного волнуясь) переведем систему на «зеленую» версию и проверим, что сервис обновился (не забываем заново переадресовать порт сервиса — прежний вызов port-forward не сможет автоматически обновиться до нового сервиса, переадресацию надо остановить и перезапустить):

$ kubectl apply -f k8s/blue-green/green/k8s-time-svc-green.yaml   
service/time-service configured  
$ kubectl port-forward svc/time-service 8080  
… в новом терминале  
$ while true; do curl 0.0.0.0:[перенаправленный\_порт]/time; sleep 1; done  
{"time":"07 Jun 2022"}  
{"time":"07 Jun 2022"}  
...

Одной строкой мы мгновенно переключили time-service на новую версию, а клиенты совершенно незаметно для себя стали использовать ее через то же самое имя DNS.

Еще не время удалять «синюю» версию! В случае непредвиденных проблем и краха «зеленой» версии мы можем мгновенно откатиться на старую версию — ведь «синие» отсеки все еще работают в кластере (и снова — не забудьте заново переадресовать порт):

$ kubectl apply -f k8s/blue-green/blue/k8s-time-svc-blue.yaml   
service/time-service configured  
$ while true; do curl 0.0.0.0:[перенаправленный\_порт]/time; sleep 1; done  
{"time":"2022-06-07 22:32:28.216131391 +0000 UTC m=+435.879952166"}  
{"time":"2022-06-07 22:32:29.245915537 +0000 UTC m=+436.909736318"}  
...

Именно это и называется «сине-зеленым» развертыванием (blue-green deployment). У нас в кластере одновременно развернуты две версии одного и того же микросервиса, которые совершенно не знают о существовании друг друга. Сервис же один, и он может указывать или на «синий», или на «зеленый» сервис с помощью простейших меток. Каждый следующий цикл обновления может чередовать цвет — в следующее обновление в нашем примере уже «зеленая» версия будет стабильной, а «синяя» придет ей на замену.

Обратите внимание, насколько легко было организовать динамическое управление версиями и трафиком в кластере с помощью простейшей метки! В этом вся суть Kubernetes. Без оркестратора контейнеров организовать подобное вручную было бы непросто и требовало бы работы опытного администратора. Это еще не все — чуть позже мы увидим, как реализовать еще более удобное и простое непрерывное обновление с помощью того же объекта Deployment.

## Шаблоны YAML — Kustomize

И еще, как легко заметить, количество файлов YAML с различными минимальными отличиями стало быстро увеличиваться для каждой новой стратегии, реализуемой в кластере, будь это «канареечное» или «сине-зеленое» развертывание. Зачастую ядро системы, базовые ее настройки совершенно одинаковы, как в нашем примере, а отличие заключается лишь в номере версии образа контейнера и паре дополнительных меток labels.

Поддерживать большое количество одинаковых фрагментов YAML непросто и приводит к ошибкам, которые трудно отладить и понять. Здесь помогают различные инструменты для работы с шаблонами и многослойным построением итогового файла YAML объекта Kubernetes. Один из лучших вариантов — Kustomize. Его использование позволит в нашем случае оставить один главный объект Service и Deployment в формате YAML, а дополнительные мелкие настройки версий и меток перенести в дополнительные накладываемые на базовый файл фрагменты (overlays). Детали вы сможете легко найти в Интернете.

## Ручная отладка отсеков Pods и контейнеров с помощью меток

Метки можно задавать и удалять не только в декларациях объектов YAML, но и вручную, с помощью команды kubectl label. Казалось бы, это путь в никуда, ручное императивное управление объектами, как мы помним, через какое-то время приведет к системе, состояние которой было достигнуто потерянными и забытыми командами. Но в некоторых случаях это может быть очень полезно. Посмотрим на список отсеков (pods) развертывания time-service, у которых есть метки app, основная метка, по которой их выбирает сервис:

$ kubectl get pods --selector app=time-service  
NAME READY STATUS RESTARTS AGE  
time-service-564b4d479f-tjgx7 1/1 Running 0 22h  
time-service-canary-54ccf7c869-s8snk 1/1 Running 0 3d  
time-service-green-5c4497bbf9-m59sx 1/1 Running 0 22h

Как мы видим, можно указать селектор для команды kubectl вручную. Мы видим три отсека для текущего развертывания, «канареечного» и «зеленого». Представим, что наше «канареечное» развертывание работает в нескольких экземплярах и что-то пошло не так. Нам хотелось бы взять текущий экземпляр сервиса и контейнера time-service, но все они обслуживают реальные запросы. Нестандартное решение — убрать основную метку селектора отсека! Развертывание Deployment «потеряет» этот экземпляр микросервиса и запустит новый отсек, а сервис перестанет направлять к нему запросы.

$ kubectl label pod time-service-canary-54ccf7c869-s8snk debug=true app-  
pod/time-service-canary-54ccf7c869-s8snk labeled

Команда kubectl label позволяет одновременно добавлять и удалять метки. Здесь мы удалили нашу основную метку для селекторов app (добавив минус к ее имени) и добавили новую debug. Имя отсека (pod) скопируйте из списка kubectl get pods, так как уникальный идентификатор в конце имени отсека у вас будет отличаться.

Посмотрим, как управляющий цикл исправит потерю отсека:

$ kubectl get pods --selector app=time-service  
NAME READY STATUS RESTARTS AGE  
time-service-564b4d479f-tjgx7 1/1 Running 0 22h  
time-service-canary-54ccf7c869-n6zgb 1/1 Running 0 31s  
time-service-green-5c4497bbf9-m59sx 1/1 Running 0 22h

Как видим, для «канареечного» развертывания был запущен новый отсек, и система вернулась в желаемое состояние (desired state).

А мы получили желаемый экземпляр прежде рабочего микросервиса в свое распоряжение и можем изучить его состояние и понять, в чем может быть проблема:

$ kubectl get pods --selector debug=true,release=canary  
NAME READY STATUS RESTARTS AGE  
time-service-canary-54ccf7c869-s8snk 1/1 Running 0 3d

Эта нехитрая, но действенная техника может быть очень полезной для «горячей» отладки прямо на работающем в производственном режиме кластере, без потери функциональности и желаемого состояния кластера.

## Сетка микросервисов — Service Mesh

Мы уже смогли насладиться тем, как несколько меток позволили нам элементарно реализовать сложные (в мире, где нет помощи Kubernetes) техники «канареечного» и «сине-зеленого» развертывания. Однако все они работают на уровне перенаправления запросов к индивидуальным контейнерам, работающим в отсеках (pods). Чтобы изменить пропорцию запросов, отправляемых к «канареечной» версии, надо запускать новые отсеки, а это ресурсы, прежде всего процессора и памяти, они не бесконечны, а в случае коммерческого облака еще и весьма дороги.

Если ваш микросервис не такой уж и микро, например большой, полноценный сервер, реализовать такие развертывания будет дорого, а иногда и невозможно по причине нехватки ресурсов кластера. В этом случае оптимально было бы управлять запросами и сетевым трафиком на уровне данных, непосредственно идущих через сеть. К примеру, имея только два отсека, обычный и «канареечный», некий сетевой компонент мог бы гибко направлять запросы между ними в зависимости от указанной пропорции.

Именно это и делают сетки микросервисов (service mesh, *сёрвис меш*). Устоявшегося перевода этого термина нет, но назвать их сетью было бы неверно, так как сеть — это общее понятие передачи данных (network). Задача же сетки service mesh — объединить микросервисы в кластере Kubernetes в единое управляемое, наблюдаемое пространство, делать это упорядоченно — то есть распределить их в некую правильную, практически геометрическую, сетку.

Основные задачи сеток service mesh:

* гибкое управление сетевыми потоками данных и трафиком между микросервисами. Пример — то самое «канареечное» развертывание, но на этот раз уже на уровне сетевых потоков;
* тестирование A/Б. Вариант «канареечного» тестирования, только еще более тонкий. Обычно применяется для функциональности, видимой для конечного пользователя, например корзины покупок. С помощью триггера (обычно заголовок HTTP, флаг, или имя/тип пользователя) часть трафика направляется на новые компоненты системы;
* наблюдение и мониторинг кластера. Одно из важнейших преимуществ использования микросервисной сетки. Так как все сетевые потоки всех микросервисов находятся под управлением прокси-компонентов, можно отследить их взаимодействия в системе, подсчитать задержки ответов и сигнализировать при необычных отклонениях от ожидаемых, средних величин обслуживания (например, service level objective, SLO). Обычно результаты наблюдения выводятся на известные системы с открытым кодом Prometheus и Grafana.

Самые известные микросервисные сетки — Istio и Linkerd. Есть некоторый порог входа и обучения, установка и первоначальная настройка не так проста, требуется перенастройка всех контейнеров для работы с прокси-компонентами и установка немалого количества нестандартных сетевых компонентов, но потом вы получаете весомые преимущества. Некоторые провайдеры, например Google Kubernetes Engine, дают вам возможность сразу запускать кластер с работающей в нем сеткой Istio.

## Аннотации

Как мы увидели в этой главе, метки играют важнейшую роль в управлении кластером Kubernetes и поиске объектов в этой динамичной, сложной распределенной системе. Использовать метки стоит только со смыслом, для поиска и для важнейших параметров ваших развертываний в Kubernetes.

В большой команде разработчиков, в больших системах из многих компонентов зачастую необходимо предоставить больше информации — кому принадлежит микросервис или часть системы, к кому обратиться в случае проблемы, где находится исходный код или документация. В этом случае можно использовать аннотации (annotations) объектов Kubernetes. Это, по сути, те же самые метки, с теми же ограничениями на формат и размер, но не участвующие в поиске, и таким образом они могут содержать любую информацию любого формата и использовать одинаковые имена и значения без опасений повлиять на работу селекторов (например, для сервисов).

В качестве простого примера мы можем добавить аннотацию owner с именем автора или администратора определенного компонента. Например, для нашего развертывания time-service:

...  
# 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  
 name: time-service  
...

Аннотации часто используются в готовых решениях Kubernetes, таких как Google Kubernetes Engine или Amazon EKS, чтобы добавить к системным объектам дополнительную информацию.

## Резюме

* Основная задача меток (labels) Kubernetes — разбиение объектов, таких как отсеки (pods), развертывания (deployments) и сервисы (services), на группы и категории и легкий поиск по значениям меток. Метки — просто именованные произвольные строки. Особенно важны метки для работы сервисов. Именно по указанным меткам сервисы выбирают набор отсеков для направления к ним запросов.
* Несмотря на свою простоту, метки в совокупности с сервисами Service позволяют простым способом организовать сложную маршрутизацию трафика вашей системы — мы рассмотрели «канареечные» и «сине-зеленые» развертывания.
* В некоторых случаях полезно менять набор меток вручную, командой kubectl, вместо декларативного объявления — например, для вывода части отсеков из развертывания для отладки.
* Хорошей практикой является использование упорядоченного набора меток (выбранного и оговоренного заранее вашей командой) для всех микросервисов, работающих в кластере под управлением Kubernetes. В дополнение к меткам можно применять аннотации.
* Несмотря на свою мощь, метки ограничены работой на уровне объектов Kubernetes. Микросервисные сетки (service mesh) позволяют управлять трафиком и запросами более тонко, на уровне сетевых запросов и их параметров.