Теоретическое введение

Прежде чем погружаться в подробности развертывания данного приложения в Kubernetes, следует поговорить о том, как управлять конфигурацией. В Kubernetes все представлено в декларативном виде. Это значит, что для определения всех аспектов своего приложения вы сначала описываете, каким должно быть его состояние в кластере (обычно в формате YAML или JSON).

Декларативный подход куда более предпочтителен по сравнению с императивным, в котором состояние кластера представляет собой совокупность внесенных в него изменений.

В случае императивной конфигурации очень сложно понять и воспроизвести состояние, в котором находится кластер. Это существенно затрудняет диагностику и исправление проблем, возникающих в приложении. Предпочтительный формат для объявления состояния приложения — YAML, хотя Kubernetes поддерживает и JSON. Дело в том, что YAML выглядит несколько более компактно и лучше подходит для редактирования вручную.

Однако стоит отметить: этот формат чувствителен к отступам, из-за чего многие ошибки в конфигурационных файлах связаны с некорректными пробельными символами. Если что-то идет не так, то имеет смысл проверить отступы. Поскольку декларативное состояние, хранящееся в этих YAML-файлах, служит источником истины, из которого ваше приложение черпает информацию, правильная работа с состоянием — залог успеха.

В ходе его редактирования вы должны иметь возможность управлять изменениями, проверять их корректность, проводить аудит их авторов и откатывать изменения в случае неполадок. К счастью, в контексте разработки ПО для всего этого уже есть подходящие инструменты. В частности, практические рекомендации, относящиеся к управлению версиями и аудиту изменений кода, можно смело использовать в работе с декларативным состоянием приложения.

Процесс сборки образов как таковой уязвим к «атакам на поставщиков». В ходе подобных атак злоумышленник внедряет код или двоичные файлы в одну из зависимостей, которая хранится в доверенном источнике и участвует в сборке вашего приложения. Поскольку это создает слишком высокий риск, в сборке должны участвовать только хорошо известные и доверенные провайдеры образов. В качестве альтернативы все образы можно собирать с нуля; для некоторых языков (например, Go) это не составляет труда, поскольку они могут создавать статические исполняемые файлы, но в интерпретируемых

языках, таких как Python, JavaScript и Ruby, это может быть большой проблемой.

Некоторые рекомендации касаются выбора имен для образов. Теоретически тег с версией образа контейнера в реестре можно изменить, но вы никогда не должны этого делать. Хороший пример системы именования — сочетание семантической версии и SHA-хеша фиксации кода, из которой собирается образ (например, v1.0.1-bfeda01f). Если версию не указать, то по умолчанию используется значение latest. Оно может быть удобно в процессе разработки, но в промышленных условиях данного значения лучше избегать, так как оно явно изменяется при создании каждого нового образа.

Рекомендуется следить за тем, чтобы содержимое кластера в точности соответствовало содержимому репозитория. Для этого лучше всего использовать методику GitOps и брать код для промышленной среды только из определенной ветки системы контроля версий. Данный процесс можно автоматизировать с помощью непрерывной интеграции (Continuous Integration, CI) и непрерывной доставки (Continuous Delivery, CD). Это позволяет гарантировать соответствие между репозиторием и промышленной системой. Для простого приложения полноценный процесс CI/CD может показаться избыточным, но автоматизация как таковая, даже если не брать во внимание повышение надежности, которое она обеспечивает, обычно стоит затраченных усилий. Внедрение CI/CD в уже существующий проект с императивным развертыванием — чрезвычайно сложная задача.

Kubernetes — эффективная система, которая может показаться сложной. Однако процесс развертывания обычного приложения легко упростить, если следовать общепринятым рекомендациям.

– Большинство сервисов нужно развертывать в виде ресурса Deployment. Объекты Deployment создают идентичные реплики для масштабирования и обеспечения избыточности.

– Для доступа к объектам Deployment можно использовать объект Service, который, в сущности, является балансировщиком нагрузки. Service может быть доступен как изнутри (по умолчанию), так и снаружи. Если вы хотите, чтобы к вашему HTTP-приложению можно было обращаться, то используйте контроллер Ingress для добавления таких возможностей, как маршрутизация запросов и SSL.

– Рано или поздно ваше приложение нужно будет параметризировать, чтобы сделать его конфигурацию более пригодной к использованию в разных средах. Для этого лучше всего подходят диспетчеры пакетов, такие как Helm (helm.sh).

Полезные ссылки

Kubernetes: лучшие практики. — СПб.: Питер, 2021. — 288 с.: ил. — (Серия «Для профессионалов»).

K8S для начинающих. Первая часть — Текст: электронный [сайт]. — URL: https://habr.com/ru/post/589415/

Kubernetes или с чего начать, чтобы понять что это и зачем он нужен — Текст: электронный [сайт]. — URL: https://habr.com/ru/company/otus/blog/537162/

Основы Kubernetes — Текст: электронный [сайт]. — URL: https://habr.com/ru/post/258443/

Практическая часть

В данной практической будет рассмотрена конфигурация развертывания систем.

Для организации компонентов приложения обычно стоит использовать структуру папок файлов системы. Сервис приложения обычно хранится в отдельном каталоге, а его компоненты — в подкаталогах.

В этом примере мы структурируем файлы таким образом:

journal/

frontend/

redis/

fileserver/

Внутри каждого каталога находятся конкретные YAML-файлы, необходимые для определения сервиса. Как вы позже сами увидите, по мере развертывания нашего приложения в разных регионах или кластерах эта структура каталогов будет все более усложняться.

В качестве этой практике будет выступать приложение для Node.js, написанное на языке TypeScript. Его код (https://github.com/brendandburns/kbp-sample) На порте 8080 работает HTTP-сервис, который обслуживает запросы к /api/* и использует сервер Redis для добавления, удаления и вывода актуальных записей журнала. Вы можете собрать это приложение в виде образа контейнера, используя включенный в его код файл Dockerfile, и загрузить его в собственный репозиторий образов. Затем вы сможете подставить его имя в YAML-файлы, приведенные ниже.

Наше клиентское приложение является stateless (не хранит свое состояние), делегируя данную функцию серверу Redis. Благодаря этому его можно реплицировать произвольным образом без воздействия на трафик. И хотя наш пример вряд ли будет испытывать серьезные нагрузки, все же неплохо использовать как минимум две реплики (копии): это позволяет справляться с неожиданными сбоями и выкатывать новые версии без простоя.

В Kubernetes есть объект ReplicaSet, отвечающий за репликацию контейнеризованных приложений, но его лучше не использовать напрямую. Для наших задач подойдет объект Deployment, который сочетает в себе возможности объекта ReplicaSet, систему управления версиями и поддержку поэтапного развертывания обновлений. Объект Deployment позволяет применять встроенные в Kubernetes механизмы для перехода от одной версии к другой.

Ресурс Deployment нашего приложения выглядит следующим образом:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    app: frontend
  name: frontend
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - image: my-repo/journal-server:v1-abcde
        imagePullPolicy: IfNotPresent
        name: frontend
        resources:
          requests:
            cpu: "1.0"
            memory: "1G"
          limits:
            cpu: "1.0"
            memory: "1G"
```

Так, для идентификации экземпляров ReplicaSet, объекта Deployment и создаваемых им pod используются метки (labels). Мы добавили метку layer: frontend для всех этих объектов, благодаря чему они теперь находятся в общем слое (layer) и их можно просматривать вместе с помощью одного запроса. Обратите внимание и на то, что для контейнеров в ресурсе Deployment установлены запросы ресурсов Request и Limit с одинаковыми значениями. Request гарантирует выделение определенного объема ресурсов на сервере, на котором запущено приложение. Limit — максимальное потребление ресурсов, разрешенное к использованию контейнером. в. Когда Request и Limit равны, ваше приложение не тратит слишком много процессорного времени и не потребляет лишние ресурсы при бездействии.

Для настройки внешнего доступа для HTTP-трафика мы воспользуемся двумя ресурсами. Первый — это Service, который распределяет (балансирует) трафик, поступающий по TCP или UDP. В нашем примере мы задействуем протокол TCP. Второй ресурс — объект Ingress, обеспечивающий балансировку нагрузки с гибкой маршрутизацией запросов в зависимости от доменных имен и HTTP-путей.

Прежде чем определять ресурс Ingress, следует создать Kubernetes Service, на который он будет указывать. А чтобы связать этот Service с pod,

созданными в предыдущем разделе, мы воспользуемся метками. Определение Service выглядит намного проще, чем ресурс Deployment:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: frontend
  name: frontend
  namespace: default
spec:
  ports:
  - port: 8080
    protocol: TCP
    targetPort: 8080
  selector:
    app: frontend
  type: ClusterIP
```

Теперь можно определить ресурс Ingress. Он, в отличие от Service, требует наличия в кластере контейнера с подходящим контроллером. Контроллеры бывают разные: одни из них предоставляются облачными провайдерами, а другие основываются на серверах с открытым исходным кодом. Если вы выбрали открытую реализацию Ingress, то для ее установки и обслуживания лучше использовать диспетчер пакетов Helm (helm.sh). Популярностью пользуются такие реализации, как nginx и haproxy

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: frontend-ingress
spec:
  rules:
  - http:
      paths:
      - path: /api
        backend:
          serviceName: frontend
          servicePort: 8080
```

Конфигурация позволяет быстро (и даже динамически) активизировать и деактивизировать возможности в зависимости от потребностей пользователей или программных сбоев. В Kubernetes такого рода конфигурация представлена в ConfigMap, в формате «ключ — значение». Эта информация предоставляется подам с помощью файлов или переменных среды. Для того, чтобы создать ConfigMap с переменной journalEntries нужно выполнить команду:

```
kubectl create configmap frontend-config --from-literal=journalEntries=10
```

Затем вы должны предоставить конфигурационную информацию в виде переменной среды в самом приложении. Для этого в раздел “containers” ресурса Deployment, который вы определили ранее, можно добавить следующий код:

```
containers:
  - name: frontend
    ...
    env:
    - name: JOURNAL_ENTRIES
      valueFrom:
        configMapKeyRef:
          name: frontend-config
          key: journalEntries
```

В любом реальном проекте соединение между сервисами должно быть защищенным. Для аутентификации в сервере Redis используется обычный пароль, который будет храниться в объекте Secret. Для того чтобы создать этот объект Secret воспользуемся командой:

```
kubectl create secret generic redis-passwd --from-literal=passwd=${RANDOM}
```

Сохранив пароль к Redis в виде объекта Secret, вы должны привязать его к приложению, которое развертывается в Kubernetes. Для этого можно использовать ресурс Volume (том). В случае с секретными данными том создается в виде файловой системы tmpfs в оперативной памяти, и затем подключается к контейнеру.

Чтобы добавить секретный том в объект Deployment, вам нужно указать в YAML-файле последнего два дополнительных раздела. Первый раздел, volumes, добавляет том в pod:

```
volumes:
- name: passwd-volume
  secret:
  secretName: redis-passwd
```

Затем том нужно подключить к определенному контейнеру. Для этого в описании контейнера следует указать поле volumeMounts:

```
volumeMounts:
- name: passwd-volume
  readOnly: true
  mountPath: "/etc/redis-passwd"
```

Благодаря этому том становится доступным для клиентского кода в каталоге redis-passwd. Итоговый объект Deployment будет выглядеть следующим образом:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    app: frontend
  name: frontend
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - image: my-repo/journal-server:v1-abcde
        imagePullPolicy: IfNotPresent
        name: frontend
        volumeMounts:
        - name: passwd-volume
          readOnly: true
          mountPath: "/etc/redis-passwd"
        resources:
          requests:
            cpu: "1.0"
            memory: "1G"
          limits:
            cpu: "1.0"
            memory: "1G"
      volumes:
        - name: passwd-volume
          secret:
            secretName: redis-passwd
```

Для развертывания сервиса Redis необходим StatefulSet. Это дополнение к ReplicaSet, которое предоставляет более строгие гарантии, такие как согласованные имена и определенный порядок увеличения и уменьшения количества pod (scale-up, scale-down).

Чтобы запросить постоянный том для нашего сервиса Redis, используется PersistentVolumeClaim. Это своеобразный запрос ресурсов. Сервис объявляет, что ему нужно хранилище размером 50 Гбайт, а кластер определяет, как выделить подходящий постоянный том. Данный механизм нужен по двум причинам. Во-первых, он позволяет создать ресурс StatefulSet, который можно переносить между разными облаками и размещать локально, не заботясь о конкретных физических дисках. Во-вторых, несмотря на то, что том типа PersistentVolume можно подключить лишь к одному pod, запрос тома позволяет написать шаблон, доступный для реплицирования, но при этом каждому pod будет назначен отдельный постоянный том.

Ниже показан пример ресурса StatefulSet для Redis с постоянными томами:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: redis
spec:
  serviceName: "redis"
  replicas: 1
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
      - name: redis
        image: redis:5-alpine
        ports:
        - containerPort: 6379
          name: redis
        volumeMounts:
        - name: data
          mountPath: /data
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 10Gi
```

Когда мы добавляем в объект StatefulSet новый неуправляемый (headless) сервис, для него автоматически создается DNS-запись redis-0.redis; это IP-адрес первой реплики. Вы можете воспользоваться этим для написания сценария, пригодного для запуска во всех контейнерах:

```
#!/bin/sh

PASSWORD=$(cat /etc/redis-passwd/passwd)

if [[ "${HOSTNAME}" == "redis-0" ]]; then
  redis-server --requirepass ${PASSWORD}
else
  redis-server --slaveof redis-0.redis 6379 --masterauth ${PASSWORD}
  --requirepass ${PASSWORD}
fi
```

Этот сценарий можно оформить в виде ConfigMap:

```
kubectl create configmap redis-config --from-file=launch.sh=launch.sh
```

Затем объект ConfigMap нужно добавить в StatefulSet и использовать его как команду для управления контейнером. Добавим также пароль для аутентификации, который создали ранее.

Полное определение сервиса Redis с тремя репликами выглядит следующим образом:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: redis
spec:
  serviceName: "redis"
  replicas: 3
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
      - name: redis
        image: redis:5-alpine
        ports:
        - containerPort: 6379
          name: redis
        volumeMounts:
        - name: data
          mountPath: /data
        - name: script
          mountPath: /script/launch.sh
          subPath: launch.sh
        - name: passwd-volume
          mountPath: /etc/redis-passwd
        command:
        - sh
        - -c
        - /script/launch.sh
      volumes:
      - name: script
        configMap:
          name: redis-config
          defaultMode: 0777
      - name: passwd-volume
        secret:
          secretName: redis-passwd
  volumeClaimTemplates:
  - metadata:
      name: data
  spec:
    accessModes: [ "ReadWriteOnce" ]
    resources:
      requests:
        storage: 10Gi
```

Итак, мы развернули stateful-сервис Redis; теперь его нужно сделать доступным для нашего клиентского приложения. Для этого создадим два разных Service Kubernetes. Первый будет читать данные из Redis. Поскольку они реплицируются между всеми тремя участниками StatefulSet, для нас несущественно, к какому из них будут направляться наши запросы на чтение. Следовательно, для этой задачи подойдет простой Service:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: redis
  name: redis
  namespace: default
spec:
  ports:
  - port: 6379
    protocol: TCP
    targetPort: 6379
  selector:
    app: redis
  sessionAffinity: None
  type: ClusterIP
```

Выполнение записи потребует обращения к ведущей реплике Redis (под номером 0). Создайте для этого неуправляемый (headless) Service. У него нет IP-адреса внутри кластера; вместо этого он задает отдельную DNS-запись для каждого pod в StatefulSet. То есть мы можем обратиться к нашей ведущей реплике по доменному имени redis-0.redis:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: redis-write
  name: redis-write
spec:
  clusterIP: None
  ports:
  - port: 6379
  selector:
    app: redis
```

Таким образом, если нам нужно подключиться к Redis для сохранения каких-либо данных или выполнения транзакции с чтением/записью, то мы можем собрать отдельный клиент, который будет подключаться к серверу redis-0.redis-write

Заключительный компонент нашего приложения — сервер статических файлов, который отвечает за раздачу HTML-, CSS-, JavaScript-файлов и изображений.

Ingress позволяет очень легко организовать такую архитектуру в стиле микросервисов. Как и в случае с клиентским приложением, можно реплицируемый сервер NGINX с помощью ресурса Deployment. Соберем статические образы в контейнер NGINX и развернем их в каждой реплике. Ресурс Deployment будет выглядеть следующим образом:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    app: fileserver
  name: fileserver
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: fileserver
  template:
    metadata:
      labels:
        app: fileserver
    spec:
      containers:
      - image: my-repo/static-files:v1-abcde
        imagePullPolicy: Always
        name: fileserver
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
        resources:
          requests:
            cpu: "1.0"
            memory: "1G"
          limits:
            cpu: "1.0"
            memory: "1G"
      dnsPolicy: ClusterFirst
      restartPolicy: Always
```

Теперь, запустив реплицируемый статический веб-сервер, вы можете аналогичным образом создать ресурс Service, который будет играть роль балансировщика нагрузки

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: fileserver
  name: fileserver
  namespace: default
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: fileserver
  sessionAffinity: None
  type: ClusterIP
```

Итак, у вас есть Service для сервера статических файлов. Добавим в ресурс Ingress новый путь. Необходимо отметить, что путь / должен идти после /api, иначе запросы API станут направляться серверу статических файлов. Обновленный ресурс Ingress будет выглядеть следующим образом:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: frontend-ingress
spec:
  rules:
  - http:
    paths:
    - path: /api
      backend:
        serviceName: frontend
        servicePort: 8080
    # Примечание: этот раздел должен идти после /api,
    # иначе он будет перехватывать запросы.
    - path: /
      backend:
        serviceName: fileserver
        servicePort: 80
```

В результате выполнения работы вам необходимо отобразить в отчете все этапы конфигурации программной системы и показать работоспособность системы, доступ к которой настроен с помощью Ingress, с помощью браузера.

Вопросы к практической работе

1. Для чего нужен ресурс Deployment?
2. Почему не стоит хранить пароли в ConfigMap?
3. Что необходимо для настройки внешнего доступа для HTTP-трафика? Назовите шаги.
4. Чем отличается развертывание stateful от развертывания клиентского приложения?
5. Где хранится том с секретными данными?
6. Как работает связка PersistentVolume и PersistentVolumeClaim?

Критерии оценки

За выполнение данной практической работы можно максимально получить 2 балла.

Критерии на выставление 2 баллов:

- Соблюдены общие требования выполнения практических работ, представленные в документе “Требования к выполнению практических работ”.
- В отчете отображены все этапы конфигурации системы
- Показаны конфигурации в Minikube
- Показана работоспособность frontend-сервисов
- Сделан отчет с описанием и скриншотами выполненных заданий
- Дан полный и развернутый ответ на все вопросы преподавателя, как по вопросам к практике, так и по дополнительным вопросам к выполненному заданию.

Критерии на выставление 1 балла:

- Соблюдены общие требования выполнения практических работ, представленные в документе “Требования к выполнению практических работ”.
- В отчете отображены все этапы конфигурации системы
- Показаны конфигурации в Minikube
- Не показана работоспособность frontend-сервисов
- Сделан отчет с описанием и скриншотами выполненных заданий
- Дан полный и развернутый ответ на все вопросы преподавателя, как по вопросам к практике, так и по дополнительным вопросам к выполненному заданию.

Критерии на выставление 0 баллов:

- Не соблюдены общие требования выполнения практических работ, представленные в документе “Требования к выполнению практических работ”.
- В отчете не отображены все этапы конфигурации системы
- Не показаны конфигурации в Minikube
- Не показана работоспособность frontend-сервисов
- Сделан отчет с описанием и скриншотами выполненных заданий.
- Студент не смог ответить ни на вопросы к практической работе, ни на вопросы к ходу выполнения работы.