
Оглавление

Введение	15
Кому стоит прочесть эту книгу.....	15
Почему мы написали эту книгу	15
Структура книги	16
Условные обозначения	16
Использование примеров кода	17
Благодарности.....	18
От издательства	19
Глава 1. Создание простого сервиса	20
Обзор приложения	20
Управление конфигурационными файлами.....	20
Создание реплицированного сервиса с помощью ресурса Deployment	22
Практические рекомендации по управлению образами	23
Создание реплицированного приложения	23
Настройка внешнего доступа для HTTP-трафика	26
Конфигурация приложения с помощью ConfigMap	27
Управление аутентификацией с помощью объектов Secret	29
Stateful-развертывание простой базы данных	32
Создание балансировщика нагрузки для TCP с использованием Service.....	36
Направление трафика к серверу статических файлов с помощью Ingress.....	37
Параметризация приложения с помощью Helm.....	39
Рекомендации по развертыванию сервисов	41
Резюме.....	41

Глава 2. Процесс разработки	42
Цели.....	42
Построение кластера для разработки	44
Подготовка разделяемого кластера для нескольких разработчиков	45
Добавление новых пользователей.....	45
Создание и защита пространства имен.....	48
Управление пространствами имен	50
Сервисы уровня кластера	51
Рабочие процессы разработчика	52
Начальная подготовка	52
Активная разработка	53
Тестирование и отладка	54
Рекомендации по подготовке среды для разработки.....	55
Резюме	56
Глава 3. Мониторинг и ведение журнала в Kubernetes	57
Метрики и журнальные записи	57
Разновидности мониторинга	57
Методы мониторинга	58
Обзор метрик, доступных в Kubernetes	59
сAdvisor	60
Сервер метрик.....	60
kube-state-metrics	61
Какие метрики нужно отслеживать.....	62
Средства мониторинга	63
Мониторинг в Kubernetes с использованием Prometheus	65
Обзор журналирования	70
Инструменты для ведения журнала	72
Журналирование с использованием стека EFK.....	72
Уведомления	75
Рекомендации по мониторингу, журналированию и созданию уведомлений.....	77
Мониторинг	77

Журналирование	77
Создание уведомлений	78
Резюме	78
Глава 4. Конфигурация, Secrets и RBAC	79
Конфигурация с использованием объектов ConfigMap и Secret	79
Объекты ConfigMap	80
Объекты Secret	80
Общепринятые рекомендации по работе с API ConfigMap и Secret	81
RBAC	88
Основные концепции RBAC	89
Рекомендации по работе с RBAC	91
Резюме	93
Глава 5. Непрерывная интеграция, тестирование и развертывание	94
Управление версиями	95
Непрерывная интеграция	95
Тестирование	96
Сборка контейнеров	96
Назначение тегов образам контейнеров	97
Непрерывное развертывание	98
Стратегии развертывания	99
Тестирование в промышленных условиях	104
Подготовка процесса и проведение хаотического эксперимента	105
Подготовка CI	105
Подготовка CD	108
Выполнение плавающего обновления	109
Простой хаотический эксперимент	109
Рекомендации относительно CI/CD	110
Резюме	111
Глава 6. Версии, релизы и выкатывание обновлений	112
Ведение версий	113
Релизы	113
Развертывание обновлений	114

Полноценный пример	115
Рекомендации по ведению версий, созданию релизов и развертыванию обновлений	119
Резюме	120
Глава 7. Глобальное распределение приложений и промежуточное тестирование	121
Распределение вашего образа	122
Параметризация развертываний	123
Глобальное распределение трафика	124
Надежное развертывание программного обеспечения в глобальном масштабе	124
Проверка перед развертыванием	125
Канареечный регион	128
Разные типы регионов	129
Подготовка к глобальному развертыванию	130
Когда что-то идет не так	131
Рекомендации по глобальному развертыванию	133
Резюме	134
Глава 8. Управление ресурсами	135
Планировщик Kubernetes	135
Предикаты	135
Приоритеты	136
Продвинутые методики планирования	137
Принадлежность и непринадлежность pod	137
nodeSelector	138
Ограничения и допуски	139
Управление ресурсами pod	141
Запросы ресурсов	141
Лимиты на ресурсы и качество обслуживания	142
Объекты PodDisruptionBudget	144
Управление ресурсами с помощью пространств имен	146
ResourceQuota	147
LimitRange	149

Масштабирование кластера.....	150
Масштабирование приложений	151
Масштабирование с использованием HPA.....	152
HPA с применением пользовательских метрик.....	153
Масштабирование с использованием VPA	154
Рекомендации по управлению ресурсами	154
Резюме	155
Глава 9. Сетевые возможности, безопасность сети и межсервисное взаимодействие	156
Принципы работы с сетью в Kubernetes.....	156
Сетевые дополнения	159
Kubenet	160
Рекомендации по использованию Kubenet	160
Дополнение CNI	160
Рекомендации по использованию CNI.....	161
Сервисы в Kubernetes	162
Тип сервисов ClusterIP.....	163
Тип сервисов NodePort	164
Тип сервисов ExternalName	165
Тип сервисов LoadBalancer	166
Объекты и контроллеры Ingress	168
Рекомендации по использованию сервисов и контроллеров Ingress	169
Сетевые политики безопасности.....	170
Рекомендации по применению сетевых политик.....	173
Механизмы межсервисного взаимодействия	175
Рекомендации по применению механизмов межсервисного взаимодействия	177
Резюме.....	177
Глава 10. Безопасность pod и контейнеров	179
API PodSecurityPolicy	179
Включение PodSecurityPolicy	180
Принцип работы PodSecurityPolicy	181
Трудности при работе с PodSecurityPolicy	190

Рекомендации по использованию политики PodSecurityPolicy	191
PodSecurityPolicy: что дальше?.....	192
Изоляция рабочих заданий и RuntimeClass	192
Использование RuntimeClass	193
Реализации сред выполнения.....	194
Изоляция рабочих заданий и рекомендации по использованию RuntimeClass	194
Другие важные аспекты безопасности pod и контейнеров	195
Контроллеры доступа.....	195
Средства обнаружения вторжений и аномалий.....	195
Резюме.....	196
Глава 11. Политики и принципы управления кластером.....	197
Почему политики и принципы управления кластером имеют большое значение	197
В чем отличие от других политик	198
Облачно-ориентированная система политик.....	198
Введение в Gatekeeper	198
Примеры политик.....	199
Терминология проекта Gatekeeper.....	199
Определение шаблона ограничений.....	200
Определение ограничений	201
Репликация данных.....	203
Обратная связь	203
Аудит	204
Более тесное знакомство с Gatekeeper	205
Gatekeeper: что дальше?	205
Рекомендации относительно политик и принципов управления.....	206
Резюме.....	207
Глава 12. Управление несколькими кластерами	208
Зачем может понадобиться больше одного кластера.....	208
Проблемы многокластерной архитектуры	211
Развертывание в многокластерной архитектуре	213
Методики развертывания и администрирования.....	213

Администрирование кластера с помощью методики GitOps.....	216
Средства управления несколькими кластерами	218
Kubernetes Federation.....	219
Рекомендации по эксплуатации сразу нескольких кластеров	222
Резюме	223
Глава 13. Интеграция внешних сервисов с Kubernetes.....	224
Импорт сервисов в Kubernetes	224
Сервисы со стабильными IP-адресами без использования селекторов	225
Стабильные доменные имена сервисов на основе CNAME.....	226
Активный подход с применением контроллеров	228
Экспорт сервисов из Kubernetes.....	229
Экспорт сервисов с помощью внутреннего балансировщика нагрузки	229
Экспорт сервисов типа NodePort.....	230
Интеграция внешних серверов в Kubernetes	231
Разделение сервисов между кластерами Kubernetes	232
Сторонние инструменты	233
Рекомендации по соединению кластеров и внешних сервисов.....	234
Резюме	235
Глава 14. Машинное обучение и Kubernetes.....	236
Почему Kubernetes отлично подходит для машинного обучения	236
Рабочий процесс машинного обучения	237
Машинное обучение с точки зрения администраторов кластеров Kubernetes.....	238
Обучение модели в Kubernetes	239
Распределенное обучение в Kubernetes.....	241
Требования к ресурсам	242
Специализированное оборудование	242
Библиотеки, драйверы и модули ядра	244
Хранение.....	244
Организация сети.....	245
Узкоспециализированные протоколы	246

Машинное обучение с точки зрения специалистов по анализу данных	246
Рекомендации по машинному обучению в Kubernetes	247
Резюме	248
Глава 15. Построение высокоуровневых абстракций на базе Kubernetes	249
Разные подходы к разработке высокоуровневых абстракций	249
Расширение Kubernetes	250
Расширение кластеров Kubernetes	251
Расширение пользовательских аспектов Kubernetes	252
Архитектурные аспекты построения новых платформ	253
Поддержка экспорта в образ контейнера	253
Поддержка существующих механизмов для обнаружения сервисов и работы с ними	254
Рекомендации по созданию прикладных платформ	255
Резюме	256
Глава 16. Управление состоянием	257
Тома и их подключение	258
Рекомендации по обращению с томами	259
Хранение данных в Kubernetes	259
PersistentVolume	260
PersistentVolumeClaim	260
Классы хранилищ	262
Рекомендации по использованию хранилищ в Kubernetes	263
Приложения с сохранением состояния	264
Объекты StatefulSet	265
Проект Operator	267
Рекомендации по использованию StatefulSet и Operator	268
Резюме	270
Глава 17. Контроль доступа и авторизация	271
Контроль доступа	271
Что такое контроллеры доступа	272
Почему они важны	272

Типы контроллеров доступа	273
Конфигурация веб-хуков доступа	274
Рекомендации по использованию контроллеров доступа.....	276
Авторизация.....	278
Модули авторизации	279
Практические советы относительно авторизации	282
Резюме.....	282
Глава 18. В заключение	283
Об авторах	284
Об изображении на обложке	285

Благодаря этой конфигурации наше клиентское приложение имеет доступ к паролю, который позволяет ему войти в сервис Redis. Аналогичным образом использование пароля настраивается и в самом сервисе; мы подключаем секретный том к Redis pod и загружаем пароль из файла.

Stateful-развертывание простой базы данных

Развертывание stateful принципиально не отличается от развертывания клиентского приложения, которое мы рассматривали в предыдущих разделах, однако наличие состояния вносит дополнительные сложности. Прежде всего, планирование функционирования pod в Kubernetes зависит от ряда факторов, таких как работоспособность узла, обновление или перебалансировка. Если данные экземпляра Redis хранятся на каком-то конкретном сервере или в самом контейнере, то будут потеряны при миграции или перезапуске данного контейнера. Чтобы этого избежать, при выполнении в Kubernetes stateful-приложений нужно обязательно использовать удаленные *постоянные тома* (PersistentVolumes).

Kubernetes поддерживает различные реализации объекта PersistentVolume, но все они имеют общие свойства. Как и секретные тома, описанные ранее, они привязываются к pod и подключаются к контейнеру по определенному пути. Их особенностью является то, что они обычно представляют собой удаленные хранилища, которые подключаются по некоему сетевому протоколу: или файловому (как в случае с NFS и SMB), или блочному (как в случае с iSCSI, облачными дисками и т. д.). В целом для таких приложений, как базы данных, предпочтительны блочные диски, поскольку обеспечивают лучшую производительность. Но если скорость работы не настолько важна, то файловые диски могут быть более гибкими.



Управление состоянием, как в Kubernetes, так и в целом, — сложная задача. Если среда, в которой вы работаете, поддерживает сервисы с сохранением состояния (stateful) (например, MySQL или Redis), то обычно лучше использовать именно их. Сначала тарифы на SaaS (Software as a Service — программное обеспечение как услуга) могут показаться высокими, но если учесть все операционные требования к поддержанию состояния (резервное копирование, обеспечение локальности и избыточности данных и т. д.) и тот факт, что наличие состояния усложняет перемещение приложений между кластерами Kubernetes, то становится

очевидно, что в большинстве случаев высокая цена SaaS себя оправдывает. В средах с локальным размещением, где сервисы SaaS недоступны, имеет смысл организовать отдельную команду специалистов, которая будет предоставлять услугу хранения данных в рамках всей организации. Это, несомненно, лучше, чем позволять каждой команде выкатывать собственное решение.

Для развертывания сервиса Redis мы воспользуемся ресурсом `StatefulSet`. Это дополнение к `ReplicaSet`, которое появилось уже после выхода первой версии Kubernetes и предоставляет более строгие гарантии, такие как согласованные имена (никаких случайных хешей!) и определенный порядок увеличения и уменьшения количества pod (`scale-up`, `scale-down`). Это не так важно, когда развертывается одноэлементное приложение, но если вам нужно развернуть состояние с репликацией, то данные характеристики придутся очень кстати.

Чтобы запросить постоянный том для нашего сервиса Redis, мы воспользуемся `PersistentVolumeClaim`. Это своеобразный запрос ресурсов. Наш сервис объявляет, что ему нужно хранилище размером 50 Гбайт, а кластер Kubernetes 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

```

В результате будет развернут один экземпляр сервиса Redis. Но, допустим, вам нужно реплицировать кластер Redis, чтобы масштабировать запросы на чтение и повысить устойчивость к сбоям. Для этого, очевидно, следует довести количество реплик до трех, но в то же время сделать так, чтобы для выполнения записи новые реплики подключались к ведущему экземпляру Redis.

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
```

Создание балансировщика нагрузки для TCP с использованием Service

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`.

Направление трафика к серверу статических файлов с помощью Ingress

Заключительный компонент нашего приложения — *сервер статических файлов*, который отвечает за раздачу HTML-, CSS-, JavaScript-файлов и изображений. Отделение сервера статических файлов от нашего клиентского приложения, предоставляющего API, делает нашу работу более эффективной и целенаправленной. Для раздачи файлов можно воспользоваться готовым высокопроизводительным файловым сервером наподобие NGINX; при этом команда разработчиков может сосредоточиться на реализации нашего API.

К счастью, ресурс Ingress позволяет очень легко организовать такую архитектуру в стиле мини/микросервисов. Как и в случае с клиентским приложением, мы можем описать 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
```

Параметризация приложения с помощью Helm

Все, что мы обсуждали до сих пор, было направлено на развертывание одного экземпляра нашего сервиса в одном кластере. Но в реальности сервисы почти всегда приходится развертывать в нескольких разных средах (даже при условии, что они находятся в общем кластере). Вы можете быть разработчиком-одиночкой, который занимается всего одним приложением, но если хотите, чтобы внесение изменений не мешало пользователям работать, то вам понадобится как минимум две версии: отладочная и промышленная. И прибавив к этому интеграционное тестирование и CI/CD, мы получим следующее: даже при работе с одним сервисом и наличии лишь пары разработчиков нужно выполнять развертывание по меньшей мере в трех разных средах, и это далеко не предел, если приложение должно справляться со сбоями на уровне вычислительного центра.

Начальная стратегия для борьбы со сбоями у многих команд заключается в тривиальном копировании файлов из одного кластера в другой. Вместо одного каталога `frontend/` они используют два: `frontend-production/` и `frontend-development/`. Такой способ опасен, поскольку разработчикам приходится следить за тем, чтобы файлы оставались синхронизированными. Этого можно было бы легко добиться, если бы эти каталоги должны были быть идентичными. Но некоторые расхождения между отладочной и промышленной версиями нормальны, так как вы будете разрабатывать новые возможности; крайне важно, чтобы эти расхождения были намеренными и простыми в управлении.

Еще один подход состоит в использовании веток и системы контроля версий; центральный репозиторий разделяется на промышленную и отладочную ветки, разница между которыми видна невооруженным глазом. Это может быть хорошим вариантом для некоторых команд, но если вы хотите развертывать ПО сразу в нескольких средах (например, система CI/CD может выполнять развертывание в разных регионах облака), то переключение между ветками будет проблематичным.

В связи с этим большинство людей в итоге выбирают *систему шаблонов*. Идея в том, что централизованный каркас конфигурации приложения образуют шаблоны, которые *подставляются* для той или иной среды на основе параметров. Таким образом, вы можете иметь одну общую конфигурацию, при необходимости легко подгоняемую под определенные условия. Для Kubernetes есть множество разных систем шаблонов, но наиболее популярна, безусловно, Helm (`helm.sh`).

В Helm приложения распространяются в виде так называемых *чартов* с файлами внутри.

В основе чарта лежит файл `chart.yaml`, в котором определяются его метаданные:

```
apiVersion: v1
appVersion: "1.0"
description: A Helm chart for our frontend journal server.
name: frontend
version: 0.1.0
```

Этот файл размещается в корневом каталоге чарта (например, в `frontend/`). Там же находится каталог `templates`, внутри которого хранятся шаблоны. Шаблон, в сущности, представляет собой YAML-файл, похожий на приводимые в предыдущих примерах; разница лишь в том, что отдельные его значения заменены ссылками на параметры. Скажем, представьте, будто хотите параметризовать количество реплик в своем клиентском приложении. Вот что содержал наш исходный объект `Deployment`:

```
...
spec:
  replicas: 2
...
```

В файле шаблона (`frontend-deployment.tpl`) данный раздел выглядит следующим образом:

```
...
spec:
  replicas: {{ .replicaCount }}
...
```

Это значит, что при развертывании чарта для поля `replicas` будет подставлен подходящий параметр. Сами параметры определены в файле `values.yaml`, предназначенном для конкретной среды, в котором развертывается приложение. Для этого простого чарта файл `values.yaml` выглядел бы так:

```
replicaCount: 2
```

Теперь, чтобы собрать все указанное вместе, вы можете развернуть данный чарт с помощью утилиты `helm`, как показано ниже:

```
helm install path/to/chart --values path/to/environment/values.yaml
```

Эта команда параметризирует ваше приложение и развернет его в Kubernetes. Со временем параметризация будет расширяться, охватывая все разнообразие сред выполнения вашего приложения.

Рекомендации по развертыванию сервисов

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 параметризовать, чтобы сделать его конфигурацию более пригодной к использованию в разных средах. Для этого лучше всего подходят диспетчеры пакетов, такие как `Helm` (`helm.sh`).

Резюме

Несмотря на свою простоту, приложение, созданное нами в этой главе, охватывает практически все концепции, которые могут понадобиться вам в более крупных и сложных проектах. Понимание того, как сочетаются эти фундаментальные компоненты, и умение их использовать — залог успешного применения Kubernetes.

Использование системы контроля версий, аудита изменений кода и непрерывной доставки ваших сервисов позволит любым проектам, которые вы создаете, иметь прочный фундамент. Эта основополагающая информация пригодится вам при изучении более сложных тем, представленных в других главах.