В книге рассмотрены механизмы, обеспечивающие работу контейнеров в GNU/Linux, основы работы с контейнерами при помощи Docker и Podman, а также система оркестрирования контейнеров Kubernetes. Помимо этого, книга знакомит с особенностями одного из самых популярных дистрибутивов Kubernetes – OpenShift (OKD).

Данная книга рассчитана на ИТ-специалистов, знакомых с GNU/Linux и желающих познакомиться с технологиями контейнеров и системой оркестрации Kubernetes.

Андрей Маркелов *имеет пятнадцатилетний опыт преподавания как авторских курсов, так и авторизированных курсов по ИТ-технологиям таких компаний, как Red Hat и Microsoft. В настоящее время работает в качестве старшего менеджера по архитектуре компании Ericsson, специализируясь на облачных технологиях. До этого работал в качестве старшего системного архитектора в компании Red Hat, а также в крупных системных интеграторах России. Имеет около шестидесяти публикаций в отечественных ИТ-журналах. Автор двух книг на русском и английском языках, посвященных OpenStack. Является сертифицированным архитектором Red Hat (RHCA) с 2009 г. Блог автора - http://markelov.blogspot.ru/*

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК «Галактика»
e-mail: books@alians-kniga.ru

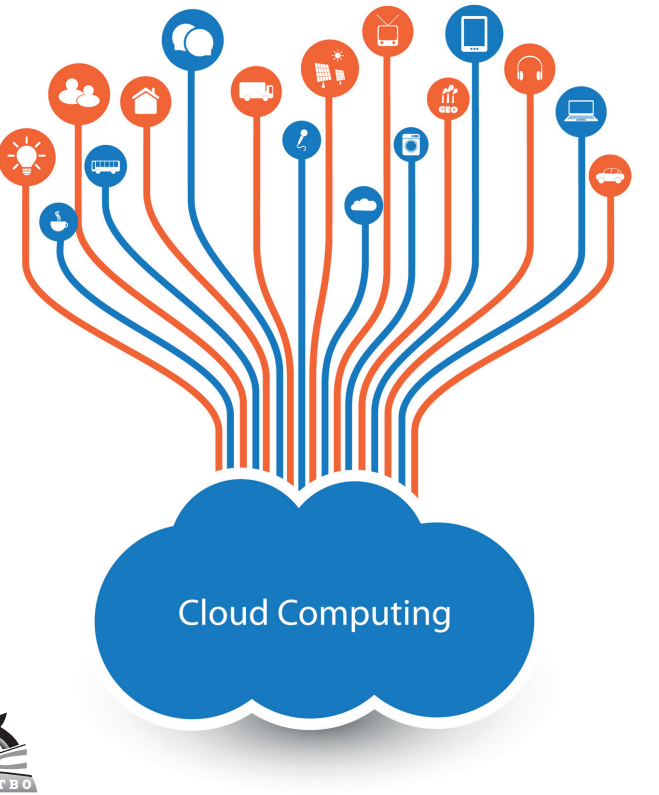
ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
www.дмк.рф

ISBN 978-5-97060-775-6
9 785970 607756 >

Введение в технологии контейнеров и Kubernetes

Андрей Маркелов

Введение в технологии контейнеров и Kubernetes



ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

ИНВЕНТА

Авторизованное обучение и сертификация Red Hat по технологиям контейнеров и Kubernetes в Учебном центре ИНВЕНТА!

Пройдите обучение и сертификацию Red Hat в ИНВЕНТЕ и получите сертификат международного образца от ведущего мирового поставщика решений с открытым исходным кодом – компании Red Hat.

Авторизованные курсы:

- DO180 Introduction to Containers, Kubernetes, and Red Hat OpenShift;
- DO280 Red Hat OpenShift Administration I;
- DO288 Red Hat OpenShift Development I: Containerizing Applications;
- DO380 Red Hat OpenShift Administration II: High Availability;
- DO425 Red Hat Security: Securing Containers and OpenShift.

Сертификационные экзамены:

- EX280 Red Hat Certified Specialist in OpenShift Administration exam;
- EX288 Red Hat Certified Specialist in OpenShift Application Development Exam;
- EX425 Red Hat Certified Specialist in Security: Containers and OpenShift Container Platform exam.

*При записи на курс или экзамен назовите промокод **OpenInventa** и получите скидку 10% и **подарок***

Срок действия промокода – 01 июня 2020 г.

Телефон: +7 (495) 775-87-77
e-mail: info@inventa.ru
г. Москва, ул. Большая Дмитровка д. 32, стр. 4

Андрей Маркелов

ВВЕДЕНИЕ В ТЕХНОЛОГИИ КОНТЕЙНЕРОВ И KUBERNETES

ДМК ПРЕСС
издательство
Москва, 2019

УДК 004.51Kubernetes
ББК 32.972.1
М25

М25 Маркелов А.А.
Введение в технологии контейнеров и Kubernetes. – М.: ДМК Пресс, 2019. – 194 с.: ил.

ISBN 978-5-97060-775-6

В книге рассмотрены механизмы, обеспечивающие работу контейнеров в GNU/ Linux, основы работы с контейнерами при помощи Docker и Podman, а также система оркестрирования контейнеров Kubernetes. Помимо этого, книга знакомит с особенностями одного из самых популярных дистрибутивов Kubernetes – OpenShift (OKD).

Данная книга рассчитана на ИТ-специалистов, знакомых с GNU/Linux и желающих познакомиться с технологиями контейнеров и системой оркестрации Kubernetes.

ISBN 978-5-97060-775-6

СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ

Благодарности

В первую очередь я хочу поблагодарить мою жену Лену за терпение и поддержку, проявленные во время написания этой книги, как и всех предыдущих.

Также традиционная большая благодарность рецензентам книги.

Об авторе

Андрей Маркелов имеет пятнадцатилетний опыт преподавания как авторских курсов, так и авторизированных курсов по ИТ-технологиям таких компаний, как Red Hat и Microsoft.

Последние годы автор работает в качестве старшего системного архитектора компании Ericsson, специализируясь на облачных технологиях, инфраструктуре виртуализации сетевых функций (NFV-I) и технологиях управления контейнерами. До этого работал в качестве старшего системного архитектора в компании Red Hat, а также в крупных системных интеграторах России, получив более чем десятилетний опыт продаж, проектирования и внедрения сетевых и инфраструктурных решений.

Около шестидесяти публикаций в российских ИТ-журналах («Системный администратор», «Linux Format», «PC Week» и др.). Автор книг «OpenStack. Практическое введение в облачную операционную систему» (ДМК Пресс, 2015, 2016, 2017, 2018 гг.), выдержавшей четыре издания, и вышедшей в 2016 году книги издательства Apress «Certified OpenStack Administrator Study Guide».

Андрей является сертифицированным архитектором Red Hat (RHCA) с 2009 года и имеет сертификаты Red Hat в таких технологиях, как OpenStack, OpenShift, RHV, CloudForms, Ansible, GlusterFS, настройка производительности, безопасности Linux-систем и др. Кроме того, автор имеет сертификаты Microsoft Certified System Engineer, Sun Certified System Administrator, Novell Certified Linux

Professional, Linux Professional Institute Certification (LPIC-1), Mirantis Certified OpenStack Administrator, OpenStack Foundation Certified OpenStack Administrator, Cisco Certified Network Associate.

Блог автора располагается по адресу: http://markelov.blogspot.com/. Его twitter-аккаунт @amarkelov.

Рецензенты

Артемий Кропачев (Artemii Kropachev) – технический директор (CTO) и главный ИТ-архитектор (Principal Architect) компании «Li9 Technology Solutions», расположенной в г. Финикс, США. Артемий имеет более 15 лет опыта в ИТ, который покрывает проектирование комплексных ИТ-систем, разработку программного обеспечения, SDN/NFV, облачные решения и системы автоматизации. Артемий занимается проектированием и внедрением проектов на базе технологий Red Hat. Основной фокус направлен на контейнеризацию приложений на базе Red Hat OpenShift Container Platform. Артемий является автором книг «Ansible for IT Experts», «Learn OpenShift», «Learn DevOps», «Learn Linux». Кроме того, Артемий на 2019 г. является обладателем самого высокого сертификационного статуса по инфраструктурным продуктам Red Hat в мире – Red Hat Certified Architect Level XX in Infrastructure.

Василий Ангапов (Vasiliy Angapov) – старший архитектор компании Bell Integrator, г. Москва. Василий имеет более 10 лет опыта в ИТ, долгое время занимался внедрением облачных решений компании Red Hat. В настоящее время успешно занимается проектированием и внедрением различных микросервисных приложений на базе Kubernetes в облачных платформах Google Cloud и Amazon Web Services.

Предполагаемая аудитория

Данная книга рассчитана на ИТ-специалистов, работающих с GNU/Linux и желающих познакомиться с технологиями контейнеров и системой оркестрации Kubernetes.

От читателя требуются базовые навыки работы с операционной системой GNU/Linux. Умение работать в командной строке и знание основных команд GNU/Linux обязательны.

О чем эта книга

Книга состоит из восьми глав и знакомит читателя с механизмами, обеспечивающими работу контейнеров в GNU/Linux, основами работы с контейнерами при помощи Docker и Podman, а также системой оркестрирования контейнеров Kubernetes. Помимо этого, книга знакомит с особенностями одного из самых популярных дистрибутивов Kubernetes – OpenShift (OKD).

Глава 1. Введение в контейнеры GNU/Linux

Первая глава вводит читателя в предметную область и историю технологий контейнеров в GNU/Linux. Также обсуждается архитектура «движка» Docker. В этой главе читатель создает лабораторный стенд и устанавливает Docker в GNU/Linux на примере дистрибутива CentOS 7.

Глава 2. Основы работы с контейнерами Docker

Во второй главе читатель познакомится с базовыми операциями и основными командами утилиты docker. Рассмотрены такие темы, как поиск образов контейнеров и теги, запуск контейнеров, управление состоянием контейнеров и их изоляция, работа с хранилищем, сетью и реестром.

Глава 3. Создание контейнеров при помощи Dockerfile

В этой главе обсуждается файл инструкций для сборки контейнеров Dockerfile и сам процесс создания новых образов.

Глава 4. Работа с контейнерами Docker без движка Docker

В данной главе рассматриваются инструменты, изначально разработанные для альтернативной среды исполнения контейнеров CRI-O: podman, buildah и skopeo. Все три утилиты не требуют наличия запущенного демона и независимы от Docker.

Глава 5. Введение в Kubernetes и установка кластера

В пятой главе читатель познакомится с архитектурой системы оркестрации контейнеров Kubernetes и создаст кластер, состоящий из трех узлов, для последующих экспериментов в процессе обучения.

Глава 6. Основы работы с Kubernetes

В данной главе описаны основные объекты Kubernetes. Читатель на практике познакомится с созданием pod-модулей, внедрений и их масштабированием. Рассмотрена работа с томами

и запросами на постоянные тома, а также секреты, карты конфигурации и сервисы Kubernetes.

Глава 7. Расширенные возможности Kubernetes

Седьмая глава знакомит с контроллерами StatefulSet, DaemonSet, Job и CronJob. Также рассмотрена работа с Ingress и Ingress Controller.

Глава 8. Знакомство с OpenShift и OKD

В данной главе читатель знакомится с одним из популярных дистрибутивов Kubernetes, разрабатываемых компанией Red Hat, – OpenShift/OKD. Дается сравнение OpenShift и Kubernetes, показана установка тестовой среды при помощи команды oc cluster up, и продемонстрирована работа с основными объектами OpenShift.

Глава 1.
ВВЕДЕНИЕ В КОНТЕЙНЕРЫ GNU/LINUX

В первой главе мы поговорим об истории технологий контейнеров в GNU/Linux и архитектуре «движка» Docker.

Отличия контейнеров от виртуализации

В отличие от «вертикального» абстрагирования в случае виртуализации, контейнеры, в частности контейнеры Docker, с рассмотрения которых мы начнем, обеспечивают «горизонтальное» разбиение операционной системы на отдельные изолированные окружения. За счет того, что в каждом контейнере, в отличие от виртуализации, обходятся без использования отдельного экземпляра операционной системы, значительно ниже накладные расходы. Минусом является тот факт, что вы не сможете на одном узле в контейнерах запускать разные операционные системы, например Windows и GNU/Linux.

Также необходимо отметить, что зачастую контейнеры используются поверх виртуальных машин, которые могут располагаться как в частном, так и в публичном облаке.

Итак, контейнеры обеспечивают часть преимуществ виртуальных машин, при этом используя меньше аппаратных ресурсов. Преимущества использования контейнеров, по сравнению с виртуальными машинами, следующие:

- горизонтальная изоляция приложений;
- меньшие накладные расходы на инфраструктуру;
- изоляция не на уровне операционной системы, а на уровне окружения приложения (необходимые библиотеки);
- скорость развертывания приложений и скорость перезапуска приложений;
- параллельная работа нескольких окружений одного приложения;

- «виртуализация приложений» – независимость приложения от операционной системы и библиотек, что упрощает обновления и тестирование;
- возможность переиспользования одного и того же контейнера для нескольких приложений на одном узле.

При использовании контейнеров, и контейнеров Docker в частности, нужно иметь в виду следующие соображения:

- Docker может являться основой для облака, работающего по сервисной модели PaaS, но сам по себе не является таким облаком. Примеры облачных платформ модели PaaS: Cloud Foundry, Red Hat OpenShift (OKD);
- хотя с технической точки зрения Docker всего лишь меняет способ упаковки приложений, это может повлечь за собой изменение архитектуры приложения;
- контейнеры – это не противопоставление виртуализации. Контейнеры могут и довольно часто запускаются внутри виртуальных машин. В качестве примера можно привести PaaS OpenShift Online, в котором контейнеры работают внутри инфраструктуры IaaS Amazon.

Все контейнеры на одном узле используют одно и то же ядро операционной системы. С точки зрения ядра операционной системы GNU/Linux, не существует такого понятия, как «контейнер». Контейнеры – это просто процессы GNU/Linux, использующие стандартные механизмы изоляции ядра.

Набор утилит, обеспечивающих работу с контейнерами, называется «движком» (container engine), а его часть, отвечающая за запуск и мониторинг контейнеров, – средой выполнения (container runtime). Мы начнем рассмотрение контейнеров с движка Docker, однако в дальнейшем, когда перейдем к Kubernetes, также познакомимся с альтернативой – CRI-O. Среда выполнения присваивает контейнерам собственные идентификаторы (ID), но этот уровень абстракции ничего не значит для ядра операционной системы. Движок контейнеров также предоставляет такие полезные вещи, как возможность диагностики и запуска дополнительных процессов в том же самом окружении контейнера, ссылаясь на его идентификатор.

История Docker

Готовые к промышленному применению контейнеры на открытых технологиях в GNU/Linux появились позже, чем механизм Jails во FreeBSD (2000 год) или Zones в Solaris (2005 год). Долгое время самой известной в GNU/Linux контейнерной технологией оставался коммерческий продукт Virtuozzo (с 2001 года) компании, в разные годы называвшейся Swsoft, Parallels и Odin. Часть кода Virtuozzo была открыта в 2005 году в виде проекта OpenVZ. Примерно с того же времени развивается проект Linux-VServer, требующий патчей к ядру Linux (нулевая версия появилась в 2001 году).

В 2006 году появился механизм cgroups (от англ. control groups – ресурсные, или контрольные, группы) как редизайн существующей технологии cpuset (подробнее о cpuset в блоге автора книги в заметке 2009 года [1]). В 2008 году появились пользовательские пространства имен, которые стали еще одной составной частью контейнеров GNU/Linux. Про контрольные группы и пространства имен мы поговорим далее в книге.

В 2008 году инженерами IBM был инициирован проект LXC, который долгое время в первую очередь ассоциировался с открытыми контейнерами GNU/Linux в противовес проприетарному Virtuozzo. Первая версия LXC вышла только в 2014 году, получив в том числе поддержку системы мандатного контроля доступа (MAC) SELinux.

В июле 2015 года Linux Foundation анонсировала запуск нового проекта Open Container Project (OCP) [21], который призван установить общие стандарты и обеспечить фундамент совместимости для продолжения развития контейнерных решений без дальнейшей фрагментации этого направления. Инициативу OCP поддержали многие крупные компании и организации, среди которых можно упомянуть Amazon Web Services, Apcera (в дальнейшем компания поглощена компанией Ericsson), Cisco, CoreOS (поглощена компанией Red Hat), EMC, Google, HP, IBM, Intel, Microsoft, Red Hat (теперь принадлежит IBM), VMware и др. В качестве основы Open Container Project выступили в значительной степени наработки проекта и компании Docker.

На настоящий момент OCI поддерживает две спецификации:

- спецификация среды выполнения (runtime-spec);
- спецификация образов контейнеров (image-spec).

В июле 2015 года была выпущена первая версия системы оркестрации контейнеров Kubrnetes, и краткий экскурс в историю мы продолжим в соответствующей главе.

Архитектура Docker

Docker использует клиент-серверную архитектуру и состоит из клиента – утилиты docker, которая обращается к серверу при помощи REST API, и демона в операционной системе GNU/Linux (dockerd). Хотя Docker работает и в отличных от GNU/Linux операционных системах, в этой книге они не рассматриваются.

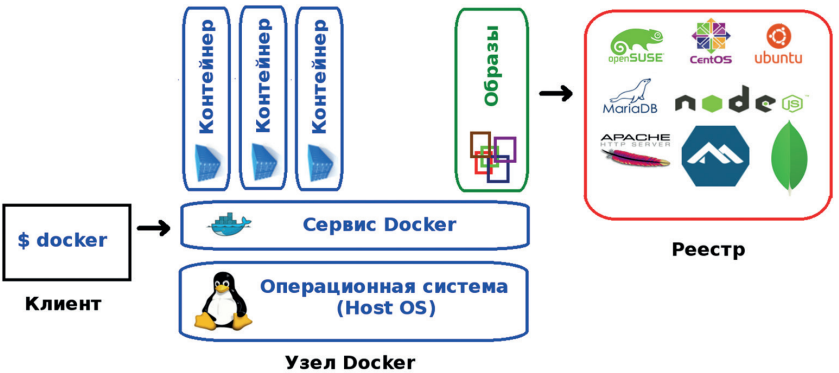


Рис. 1.1 ❖ Архитектура контейнеров Docker в GNU/Linux

Основные компоненты Docker:

- **контейнеры** – изолированные при помощи технологий операционной системы пользовательские окружения, в которых выполняются приложения. Проще всего дать определение контейнеру Docker как запущенному из образа приложению. Кстати, именно этим идеологически и отличается Docker, например, от LXC (Linux Containers), хотя они используют одни и те же технологии ядра Linux. Разработчики проекта Docker исповедуют принцип: один контейнер – это одно приложение;
- **образы** – доступные только для чтения шаблоны приложений. Поверх существующих образов могут добавляться новые уровни образов, которые совместно представляют

файловую систему, изменяя или дополняя предыдущий уровень. Обычно новый образ создается либо при помощи сохранения уже запущенного контейнера в новый образ поверх существующего, либо при помощи специальных инструкций для утилиты Dockerfile. Для разделения различных уровней контейнера на уровне файловой системы могут использоваться UnionFS, aufs, btrfs, vfs, OverlayFS и Device Mapper;

- **реестры** (registry), содержащие репозитории (repository) образов, – сетевые хранилища образов. Могут быть как приватными, так и общедоступными. Самым известным реестром является Docker Hub [3]. В книге также упоминаются репозитории, содержащие rpm-пакеты. С ними работают команды yum и dnf. Не путайте репозитории, служащие для установки пакетов операционной системы, и репозитории Docker.

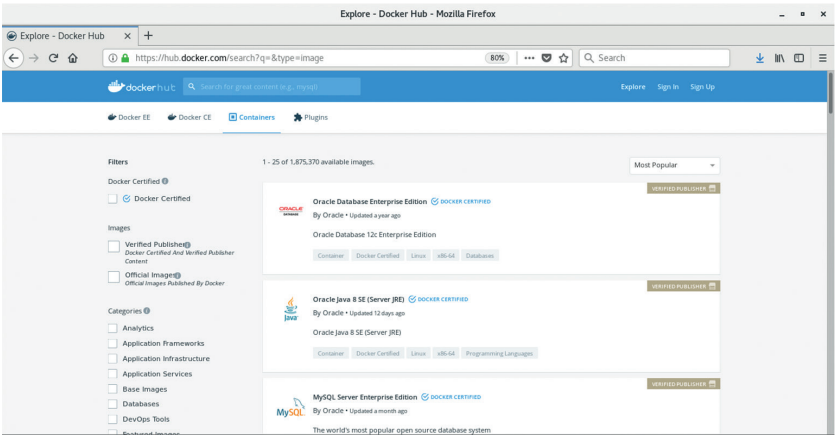


Рис. 1.2 ❖ Реестр образов Docker Hub

Для изоляции контейнеров и обеспечения безопасности в операционных системах GNU/Linux используются стандартные технологии ядра Linux, такие как:

- пространства имен (Linux Namespaces);
- контрольные группы (cgroups);
- средства управления привилегиями (Linux Capabilities);
- дополнительные, мандатные системы обеспечения безопасности, такие как AppArmor или SELinux.

Рассмотрим перечисленные технологии чуть более подробно.

Механизм контрольных групп (cgroups) предоставляет инструмент для тонкого контроля над распределением, приоритизацией и управлением системными ресурсами. Контрольные группы реализованы в ядре Linux. В современных дистрибутивах управление контрольными группами реализовано через systemd, однако сохраняется возможность управления при помощи библиотеки libcgroup и утилиты cgconfig. Основные иерархии контрольных групп (их также называют контроллерами) перечислены ниже:

- **blkio** – задает лимиты на операции ввода-вывода и на доступ к блочным устройствам;
- **cpu** – используя планировщик процессов, распределяет процессорное время между задачами;
- **cpuacct** – создает автоматические отчеты по использованию ресурсов центрального процессора. Работает совместно с контроллером cpu, описанным выше;
- **cpuset** – закрепляет за задачами определенные процессоры и узлы памяти;
- **devices** – регулирует доступ задач к определенным устройствам;
- **freezer** – приостанавливает или возобновляет задачи;
- **memory** – устанавливает лимиты и генерирует отчеты об использовании памяти задачами контрольной группы;
- **net_cls** – осуществляет тегирование сетевых пакетов идентификатором класса (classid). Это позволяет контроллеру трафика (команда tc) и брандмауэру (iptables) учитывать данные теги при обработке трафика;
- **perf_event** – позволяет производить мониторинг контрольных групп при помощи утилиты perf;
- **hugetlb** – дает возможность использовать виртуальные страницы памяти большого размера и применять к ним лимиты.

Пространства имен, в свою очередь, контролируют не распределение ресурсов, а доступ к структурам данных ядра. Фактически это означает изоляцию процессов друг от друга и возможность иметь параллельно «одинаковые», но не пересекающиеся друг с другом иерархии процессов, пользователей и сетевых интерфейсов. При желании разные сервисы могут иметь даже свои собственные loopback-интерфейсы.

Примеры пространств имен, используемых Docker:

- **PID, Process ID** – изоляция иерархии процессов;
- **User** – изоляция идентификаторов пользователей (UID) и групп (GID);
- **NET, Networking** – изоляция сетевых интерфейсов;
- **IPC, InterProcess Communication** – управление взаимодействием между процессами;
- **MNT, Mount** – управление точками монтирования;
- **UTS, Unix Timesharing System** – изоляция ядра, идентификаторов версии, имени хоста и доменного имени NIS.

Механизм под названием Capabilities позволяет разбить привилегии пользователя root на небольшие группы привилегий и назначать их по отдельности. Данный функционал в GNU/Linux появился начиная с версии ядра 2.2. Изначально контейнеры запускаются уже с ограниченным набором привилегий. При помощи опций команды docker вы можете разрешать и запрещать такие действия, как:

- операции монтирования;
- доступ к сокетам;
- выполнение части операций с файловой системой, например изменение атрибутов файлов или владельца.

Подробнее ознакомиться с привилегиями можно при помощи man-страницы руководства GNU/Linux CAPABILITIES(7).

Перейдем к последнему опциональному компоненту контейнерных технологий. SELinux (Security-Enhanced Linux) – это реализация системы мандатного контроля доступа (MAC), которая может работать (и работает по умолчанию) параллельно с классической дискреционной системой контроля доступа (DAC).

Оставаясь в рамках DAC, мы имеем фундаментальное ограничение в плане разделения доступа пользователей к ресурсам – доступ к ресурсам основывается на правах доступа пользователя. Это классические права RWX на трех уровнях – владелец, группа-владелец и остальные. Плюс к этому POSIX ACL, которые лишь расширяют число уровней, на которых можно определить права, но не более.

Таким образом, любое приложение, запущенное с правами user1, теоретически может сделать все, что угодно, со всеми данными, к которым имеет доступ user1. И не важно, например, что

данное приложение – это почтовая программа, которой нужно иметь доступ только к письмам user1. Эта программа будет иметь доступ и, например, к видеофайлам user1, и к картинкам user1. И мало того, что сама программа имеет доступ, но она же может и все эти данные сделать доступными остальным. Все становится гораздо хуже, когда user1 имеет UID, равный 0 (то есть это root). Мы утыкаемся в классическую «проблему суперпользователя». В данном случае мы получаем всего лишь два уровня доступа: root и обыкновенные пользователи. Иными словами, становится невозможным реализовать доступ с использованием минимально необходимых привилегий.

В MAC же права доступа определяются самой системой при помощи специально определенных политик. Если же говорить конкретно о рассматриваемой реализации – SELinux, то такие политики работают на уровне системных вызовов и применяются самим ядром.

SELinux действует после классической модели безопасности Unix. Иными словами, через SELinux нельзя разрешить то, что запрещено через права доступа пользователей/групп. Политики описываются при помощи специального гибкого языка описания правил доступа. В большинстве случаев правила SELinux «прозрачны» для приложений и не требуется никакой их модификации.

Установка Docker в GNU/Linux на примере CentOS 7

В данной книге для примеров мы будем использовать дистрибутив CentOS 7. Инструкции также должны без изменений работать на любых других производных от Red Hat Enterprise Linux 7. Единственное возможное отличие – это порядок настройки доступа к репозиториям RPM-пакетов.

Первое, что необходимо, – установить саму операционную систему. Предполагается, что это не должно вызвать затруднений у читателя. В качестве варианта установки можно выбрать Minimal или Server with GUI.

После установки операционной системы обновите все установленные пакеты командой

```
# yum -y update
```

При работе с CentOS у вас есть выбор: использовать последнюю версию из upstream или версию, собранную проектом CentOS, с дополнениями Red Hat. Описание изменений доступно на странице по ссылке [4]. В основном это обратное портирование исправлений из новых версий upstream и изменения, предложенные разработчиками Red Hat, но пока не принятые в основной код. Для целей первоначального знакомства с Docker мы будем использовать версию по умолчанию из стандартного репозитория CentOS:

```
[root@centos7 ~]# yum -y install docker
...
[root@centos7 ~]# yum info docker
Installed Packages
Name        : docker
Arch        : x86_64
Epoch       : 2
Version     : 1.13.1
Release     : 88.git07f3374.el7.centos
Size        : 65 M
Repo        : installed
From repo   : extras
Summary     : Automates deployment of containerized applications
URL         : https://github.com/docker/docker
License     : ASL 2.0
...
```

Настройки репозитория для установки upstream-версии, как и инструкции для инсталляции в других дистрибутивах и операционных системах, приведены в руководстве по инсталляции по ссылке [5].

Запускаем и включаем сервис:

```
[root@centos7 ~]# systemctl start docker.service
[root@centos7 ~]# systemctl enable docker.service
Created symlink from /etc/systemd/system/multi-user.target.wants/docker.
service to /usr/lib/systemd/system/docker.service.
```

Проверяем статус сервиса:

```
[root@centos7 ~]# systemctl status docker.service
• docker.service - Docker Application Container Engine
   Loaded: loaded (/usr/lib/systemd/system/docker.service; enabled; vendor
preset: disabled)
   Active: active (running) since Sun 2018-12-3013:04:04 EST; 5min ago
     Docs: http://docs.docker.com
 Main PID: 12844 (dockerd-current)
```

```
   CGroup: /system.slice/docker.service
           ├─12844 /usr/bin/dockerd-current --add-runtime docker-runc=/usr/
libexec/docker/docker-runc-current --default-runtime=docker-runc --exec-opt
native.cgroupdriver=systemd --userl...
           └─12848 /usr/bin/docker-containerd-current -l unix:///var/run/
docker/libcontainerd/docker-containerd.sock --metrics-interval=0 --start-
timeout 2m --state-dir /var/run/docker/l...
```

Также можно посмотреть системную информацию о Docker и окружении:

```
[root@centos7 ~]# docker info
Containers: 0
 Running: 0
 Paused: 0
 Stopped: 0
Images: 0
Server Version: 1.13.1
Storage Driver: overlay2
 Backing Filesystem: xfs
 Supports d_type: true
 Native Overlay Diff: true
Logging Driver: journald
Cgroup Driver: systemd
Plugins:
 Volume: local
 Network: bridge host macvlan null overlay
Swarm: inactive
Runtimes: docker-runc runc
Default Runtime: docker-runc
Init Binary: /usr/libexec/docker/docker-init-current
containerd version:  (expected: aa8187dbd3b7ad67d8e5e3a15115d3eef43a7ed1)
runc version: N/A (expected: 9df8b306d01f59d3a8029be411de015b7304dd8f)
init version: fec3683b971d9c3ef73f284f176672c44b448662 (expected:
949e6facb77383876aeff8a6944dde66b3089574)
Security Options:
 seccomp
  WARNING: You're not using the default seccomp profile
  Profile: /etc/docker/seccomp.json
 selinux
Kernel Version: 3.10.0-957.1.3.el7.x86_64
Operating System: CentOS Linux 7 (Core)
OSType: linux
Architecture: x86_64
Number of Docker Hooks: 3
CPUs: 1
Total Memory: 3.701 GiB
Name: centos7.test.local
```

```
ID: OHUV:5U3E:EOFY:LROQ:MCB5:55Q2:QSVK:DHLN:4AAZ:V22X:F7FV:B7NA
Docker Root Dir: /var/lib/docker
Debug Mode (client): false
Debug Mode (server): false
Registry: https://index.docker.io/v1/
Experimental: false
Insecure Registries:
 127.0.0.0/8
Live Restore Enabled: false
Registries: docker.io (secure)
```

Опции запуска демона, как это обычно бывает в CentOS, хранятся в файлах директории /etc/sysconfig/. В данном случае это файлы docker*:

```
[root@centos7 ~]# grep  'OPTIONS' /etc/sysconfig/docker*
/etc/sysconfig/docker:OPTIONS='--selinux-enabled --log-driver=journald
--signature-verification=false'
/etc/sysconfig/docker-network:DOCKER_NETWORK_OPTIONS=
/etc/sysconfig/docker-storage:DOCKER_STORAGE_OPTIONS="--storage-driver
overlay2 "
```

Как мы видим, Docker запускается с поддержкой SELinux, драйвером журналирования journald и драйвером хранилища overlay2.

Помимо показанной выше команды docker info, для того чтобы ознакомиться с версиями вашего окружения Docker, будет полезным и вывод docker version:

```
[root@centos7 ~]# docker version
Client:
 Version:         1.13.1
 API version:     1.26
 Package version: docker-1.13.1-88.git07f3374.el7.centos.x86_64
 Go version:      go1.9.4
 Git commit:      07f3374/1.13.1
 Built:           Fri Dec  716:13:512018
 OS/Arch:         linux/amd64

Server:
 Version:         1.13.1
 API version:     1.26 (minimum version 1.12)
 Package version: docker-1.13.1-88.git07f3374.el7.centos.x86_64
 Go version:      go1.9.4
 Git commit:      07f3374/1.13.1
 Built:           Fri Dec  716:13:512018
 OS/Arch:         linux/amd64
 Experimental:    false
```

В следующей главе мы научимся запускать контейнеры Docker и изучим базовые возможности движка контейнеров.

Вопросы для самопроверки

1. **Назовите отличия использования контейнеров по сравнению с виртуализацией (укажите все правильные ответы).**

A. Меньшие накладные расходы на инфраструктуру
B. Время старта приложений больше
C. Невозможность запуска GNU/Linux- и Windows-приложений на одном хосте
D. Обязательное использование гипервизора KVM

2. **Назовите основные компоненты Docker (укажите все правильные ответы).**

A. Гипервизор
B. Контейнеры
C. Образы виртуальных машин
D. Реестры

3. **Какие технологии используются для работы с контейнерами в GNU/Linux (укажите все правильные ответы)?**

A. Пространства имен (Linux Namespaces)
B. Подключаемые модули аутентификации (PAM)
C. Контрольные группы (cgroups)
D. Аппаратная поддержка виртуализации

Список ссылок

1. http://markelov.blogspot.se/2009/01/blog-post.html
2. https://www.opencontainers.org/
3. https://hub.docker.com/
4. http://www.projectatomic.io/docs/docker_patches/
5. https://docs.docker.com/install/linux/docker-ce/centos/

Глава 2.
ОСНОВЫ РАБОТЫ С КОНТЕЙНЕРАМИ DOCKER

В данной главе мы продолжим работу с нашим учебным стендом и познакомимся с основными командами утилиты docker.

Поиск образов контейнеров и теги

Прежде чем запустить контейнер, нам нужен образ файловой системы, который будет использован для запуска. Попробуем найти образ на Docker Hub. Это можно сделать при помощи команды docker search из командной строки. Мы получим примерно такой результат:

```
[root@centos7 ~]# docker search haproxy
INDEX       NAME                                            DESCRIPTION
STARS     OFFICIAL   AUTOMATED
docker.io   docker.io/haproxy                               HAProxy - The
Reliable, High Performance T...   1128      [OK]
docker.io   docker.io/dockercloud/haproxy                   HAProxy image
that balances between linked...   101
docker.io   docker.io/tutum/haproxy                         HAProxy image
that load balances between a...   47
```

В данном выводе мы получили список ряда образов HAProxy. В примере показаны первые три из полученных строк. Самый верхний элемент – это образ HAProxy из официального репозитория, о чем свидетельствует соответствующая колонка вывода. Такие образы отличаются тем, что в имени отсутствует символ «/», отделяющий имя репозитория пользователя от имени самого контейнера. В примере за официальным показаны два образа haproxy из открытых репозиториев dockercloud и tutum.

Образы, подобные двум нижним, вы можете создать сами, зарегистрировавшись на Docker Hub. Официальные образы поддерживаются специальной командой, спонсируемой Docker, Inc. Особенности официального репозитория:

- это рекомендованные к использованию образы, созданные с учетом лучших рекомендаций и практик;
- они представляют собой базовые образы, которые могут стать отправной точкой для более тонкой настройки. Например, базовые образы Ubuntu, CentOS или библиотек и сред разработки;
- содержат последние версии программного обеспечения с устраненными уязвимостями;
- это официальный канал распространения продуктов.

Чтобы искать только официальные образы, вы можете использовать опцию --filter "is-official=true" команды docker search:

```
[root@centos7 ~]# docker search haproxy --filter "is-official=true"
INDEX       NAME                DESCRIPTION
STARS     OFFICIAL   AUTOMATED
docker.io   docker.io/haproxy   HAProxy - The Reliable, High Performance T...
1128      [OK]
```

Число звезд в выводе команды docker search соответствует популярности образа. Это аналог кнопки Like в социальных сетях или закладок для других пользователей. Automated означает, что образ собирается автоматически из специального сценария Dockerfile средствами Docker Hub. Обычно следует отдавать предпочтение автоматически собираемым образам вследствие того, что его содержимое может быть проверено знакомством с соответствующим файлом Dockerfile.

Скачаем официальный образ HAProxy:

```
[root@centos7 ~]# docker pull haproxy
Using default tag: latest
Trying to pull repository docker.io/library/haproxy ...
latest: Pulling from docker.io/library/haproxy
177e7ef0df69: Pull complete
e6c26bdbfbe5: Pull complete
3dc4da27056d: Pull complete
Digest:
sha256:40c4ca7dd1dd713c2d0766a039f05a4f397bb52d275a9d11fe661e5342fd2afc
Status: Downloaded newer image for docker.io/haproxy:latest
```

Обратите внимание, что в выводе было указана загрузка образа с тегом latest. Использование тегов позволяет обращаться к конкретной версии программного обеспечения. На рис. 2.1 показано, какие теги и версии доступны, – обратите внимание на число 128.

Без указания версии скачивается образ с тегом latest. Помимо самого образа haproxy, был скачан базовый образ и все промежуточные, от которого зависит скачиваемый.

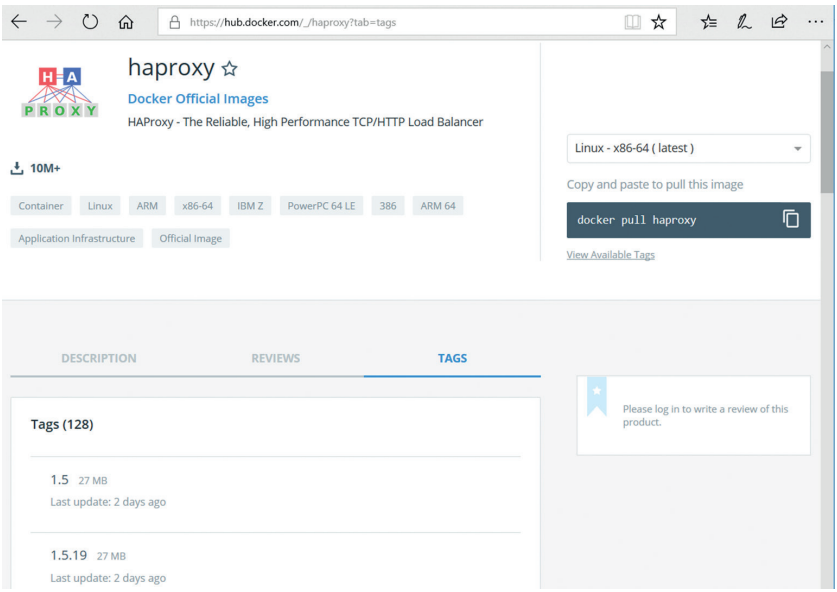


Рис. 2.1 ❖ Теги на странице HAProxy на Docker Hub

Можно скачать и конкретную версию, например:

```
[root@centos7 ~]# docker pull haproxy:1.5
Trying to pull repository docker.io/library/haproxy ...
1.5: Pulling from docker.io/library/haproxy
177e7ef0df69: Already exists
3c43d75807c7: Pull complete
e6f200fe61dc: Pull complete
Digest:
sha256:1a2ad587d3fecd940e553951096f4f4f86faa8c0de721fa2f4d2a05cc99264ec
Status: Downloaded newer image for docker.io/haproxy:1.5
```

Как вы видите, базовый образ с идентификатором 177e7ef0df69 повторно не скачивался, так как уже присутствует на нашем узле. Полное имя образа может выглядеть следующим образом:

```
[URI репозитория][имя пользователя]имя образа[:тег]
```

Просмотреть список скачанных образов можно командой docker images:

```
[root@centos7 ~]# docker images
REPOSITORY          TAG             IMAGE ID            CREATED
SIZE
docker.io/haproxy   1.5             83b17e901326        2 days ago
65.5 MB
docker.io/haproxy   latest          91783a64f7cc        2 days ago
71.9 MB
```

Команда docker images с опцией -q вернет только идентификаторы образов, что может быть полезным для использования идентификаторов в качестве входных значений для другой команды. Например, команды docker rmi, которая удаляет образ. Для того чтобы удалить все имеющиеся в локальном кеше образы, можно воспользоваться такой командой:

```
[root@centos7 ~]# docker rmi $(docker images -q)
```

Существует рекомендация не использовать только тег latest в производственной среде. Под этим тегом в разное время могут оказаться разные образы. Возможно, через день или месяц появится новый образ latest, который не обладает обратной совместимостью. Используйте теги, которые четко указывают на версию в дополнение к latest.

Вы также можете воспользоваться REST API, получив больше информации и возможностей для поиска образов, чем это позволяет клиент командной строки. Для демонстрации воспользуемся утилитой curl и декодером json.tool из стандартной библиотеки Python:

```
[user@centos7 ~]$ curl https://registry.hub.docker.com/v1/search?q=haproxy |
python -m json.tool
...
{
    "num_pages": 120,
    "num_results": 2986,
    "page": 1,
    "page_size": 25,
    "query": "haproxy",
    "results": [
        {
            "description": "HAProxy - The Reliable, High Performance TCP/HTTP
Load Balancer",
            "is_automated": false,
```

```
            "is_official": true,
            "is_trusted": false,
            "name": "haproxy",
            "star_count": 1128
        },
...
```

Следующий пример покажет список тегов образа:

```
[user@centos7 ~]$ curl https://registry.hub.docker.com/v1/repositories/
haproxy/tags | python -m json.tool
...
    {
        "layer": "",
        "name": "latest"
    },
    {
        "layer": "",
        "name": "1"
    },
    {
        "layer": "",
        "name": "1-alpine"
    },
    {
        "layer": "",
        "name": "1.4"
    },
    {
        "layer": "",
        "name": "1.4-alpine"
    },
...
```

Запуск контейнеров

Для запуска контейнера не обязательно предварительно скачивать образ. Если он доступен, то будет загружен автоматически. Давайте попробуем запустить контейнер с Ubuntu. Мы не будем указывать репозиторий, и будет скачан официальный образ, поддерживаемый Canonical. Кроме того, если не указывать тег, то будет скачан и запущен последний из LTS-релизов. Соответствующая команда:

```
[root@centos7 ~]# docker run -it ubuntu
Unable to find image 'ubuntu:latest' locally
Trying to pull repository docker.io/library/ubuntu ...
latest: Pulling from docker.io/library/ubuntu
```

```
84ed7d2f608f: Pull complete
be2bf1c4a48d: Pull complete
a5bdc6303093: Pull complete
e9055237d68d: Pull complete
Digest:
sha256:868fd30a0e47b8d8ac485df174795b5e2fe8a6c8f056cc707b232d65b8a1ab68
Status: Downloaded newer image for docker.io/ubuntu:latest
root@de7b469295b6:/#
```

Помимо команды run, мы указали две опции: -i – контейнер должен запуститься в интерактивном режиме и -t – должен быть выделен псевдотерминал. Как видно из вывода, в контейнере мы имеем привилегии пользователя root, а в качестве имени узла отображается идентификатор контейнера. Последнее может быть справедливо не для всех контейнеров и зависит от разработчика контейнера. Проверим, что это действительно окружение Ubuntu:

```
root@de7b469295b6:/# cat /etc/*release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=18.04
DISTRIB_CODENAME=bionic
DISTRIB_DESCRIPTION="Ubuntu 18.04.1 LTS"
NAME="Ubuntu"
VERSION="18.04.1 LTS (Bionic Beaver)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 18.04.1 LTS"
VERSION_ID="18.04"
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-
policy"
VERSION_CODENAME=bionic
UBUNTU_CODENAME=bionic
```

Команду uname -a для подобных целей использовать не получится, поскольку контейнер работает с ядром хоста.

В качестве одной из опций можно было бы задать уникальное имя контейнера, на которое можно для удобства ссылаться помимо ID контейнера. Оно задается как --name <имя>. В случае если опция опущена, имя генерируется автоматически.

Автоматически генерируемые имена контейнеров не несут смысловой нагрузки, однако как интересный факт можно отметить, что имена генерируются случайным образом из прилагательного и имени известного ученого, изобретателя или хакера.

В коде генератора для каждого имени можно найти краткое описание того, чем известен данный деятель.

Посмотреть список запущенных контейнеров можно командой docker ps. Для этого откроем второй терминал:

```
[root@centos7 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
de7b469295b6        ubuntu              "/bin/bash"         3 minutes ago
Up 3 minutes                            romantic_heisenberg
```

Романтичный Гейзенберг? Почему бы и нет.

В случае если необходимо запустить контейнер с процессом, не предполагающим интерактивное взаимодействие, например с демоном, используется опция -d. Так и поступим со следующим контейнером:

```
[root@centos7 ~]# docker run -d mysql
Unable to find image 'mysql:latest' locally
Trying to pull repository docker.io/library/mysql ...
latest: Pulling from docker.io/library/mysql
...
Digest:
sha256:196c04e1944c5e4ea3ab86ae5f78f697cf18ee43865f25e334a6ffb1dbea81e6
Status: Downloaded newer image for docker.io/mysql:latest
30f81da2f29b3d5acbf1d623e4ec5b38ae2ae1d8bf163f08d39f9e78a86c4353
```

Однако если отдать команду docker ps, то мы не обнаружим контейнера, созданного из образа mysql. Воспользуемся опцией -a, которая показывает все контейнеры, а не только запущенные:

```
[root@centos7 ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS                          PORTS               NAMES
30f81da2f29b        mysql               "docker-entrypoint.sh"   About a
minute ago   Exited (1) About a minute ago                       cocky_
ritchie
de7b469295b6        ubuntu              "/bin/bash"              7 minutes
ago        Exited (130) 2 minutes ago                           romantic_
heisenberg
```

В качестве статуса значится Exited. Для того чтобы разобраться с причиной, можно обратиться к журналу:

```
[root@centos7 ~]# docker logs cocky_ritchie
error: database is uninitialized and password option is not specified
  You need to specify one of MYSQL_ROOT_PASSWORD, MYSQL_ALLOW_EMPTY_PASSWORD
and MYSQL_RANDOM_ROOT_PASSWORD
```

Очевидно, что при запуске контейнера не были указанны обязательные параметры. Ознакомиться с описанием переменных среды, необходимых для запуска контейнера, можно, найдя официальный образ MySQL на Docker Hub. Повторим попытку, используя опцию -e, которая задает переменные окружения в контейнере:

```
[root@centos7 ~]# docker run --name mysql-test -e MYSQL_ROOT_PASSWORD=docker
-d mysql
4b5193027fc74f18f39eab0d6b3c2b7590ea5795c2aaaf55ef86307e47e98c1b
[root@centos7 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                 NAMES
9156f422a633        ubuntu               "/bin/bash"              About an
hour ago   Up About an hour                        romantic_heisenberg
4b5193027fc7        mysql               "docker-entrypoint..."   11 seconds
ago      Up 10 seconds       3306/tcp, 33060/tcp   mysql-test
```

При помощи следующей команды можно подключиться к работающему контейнеру:

```
[root@centos7 ~]# docker exec -it mysql-test bash
root@4b5193027fc7:/#
```

Последним параметром выступает команда, которую мы хотим исполнить внутри контейнера. В данном случае это командный интерпретатор Bash. Опции -it аналогичны по назначению использованным ранее в команде docker run.

Фактически после запуска этой команды в контейнер mysql-test добавляется еще один процесс – bash. Это можно наглядно увидеть при помощи команды pstree. Если вы получили сообщение, что команда не найдена, необходимо установить пакет psmisc:

```
[root@centos7 ~]# yum -y install psmisc
```

Сокращенный вывод до команды docker exec:

```
[root@centos7 ~]# pstree -p
systemd(1)─┬─NetworkManager(2783)─┬─dhclient(13182)
           ├─dockerd-current(3276)─┬─docker-containe(3326)─┬─docker-containe(
13559)─┬─mysqld(13573)─┬─{mysqld}(13728)
```

И после команды docker exec:

```
[root@centos7 ~]# pstree -p
systemd(1)─┬─NetworkManager(2783)─┬─dhclient(13182)
           ├─dockerd-current(3276)─┬─docker-containe(3326)─┬─docker-containe(
13559)─┬─mysqld(13573)─┬─{mysqld}(13728)
```

```
          |                    |                    ├─docker-
containe(13896)─┬─bash(13906)
```

Также можно воспользоваться командой docker top, поскольку из вывода pstree не очевидно, что процессы mysqld и bash принадлежат одному и тому же контейнеру:

```
[root@centos7 ~]# docker top mysql-test
UID                 PID                 PPID                C
STIME               TTY                 TIME                CMD
polkitd             13573               13559               0
05:21               ?                   00:00:02            mysqld
```

Вывод после запуска команды docker exec:

```
[root@centos7 ~]# docker top mysql-test
UID                 PID                 PPID                C
STIME               TTY                 TIME                CMD
polkitd             13573               13559               0
05:21               ?                   00:00:02            mysqld
root                13906               13896               0
05:28               pts/1               00:00:00            bash
```

Теперь посмотрим на вывод команды docker ps, которая покажет список запущенных контейнеров:

```
[root@centos7 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                 NAMES
4b5193027fc7        mysql               "docker-entrypoint.sh"   15 minutes
ago      Up 15 minutes       3306/tcp, 33060/tcp   mysql-test
```

Мы видим, что в качестве команды, запущенной при старте контейнера, использовался скрипт docker-entrypoint.sh. Это достаточно распространенный способ инициализации программного обеспечения в контейнере. Если бы мы хотели в отладочных целях запустить контейнер, но вместо скрипта просто получить приглашение командной строки, мы бы модифицировали команду запуска уже известным вам образом:

```
[root@centos7 ~]# docker run -it --name mysql-test2 -e MYSQL_ROOT_
PASSWORD=docker mysql /bin/bash
root@fadd0e0e631d:/#
```

После чего можно было бы изучить стартовый скрипт:

```
root@fadd0e0e631d:/# which docker-entrypoint.sh
/usr/local/bin/docker-entrypoint.sh
root@fadd0e0e631d:/# cat /usr/local/bin/docker-entrypoint.sh
```

Обратите внимание на одинаковый для двух контейнеров вывод в столбце COMMAND команды docker ps:

```
[root@centos7 ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS                          PORTS                  NAMES
fadd0e0e631d        mysql               "docker-entrypoint.sh"   About a
minute ago   Up About a minute             3306/tcp, 33060/tcp   mysql-test2
4b5193027fc7        mysql               "docker-entrypoint.sh"   18 minutes
ago       Up 18 minutes                  3306/tcp, 33060/tcp   mysql-test
```

Однако команда pstree, запущенная в контейнере, покажет нам разницу в запуске контейнеров. Для mysql-test:

```
root@4b5193027fc:/# pstree -p
mysqld(1)-+-{mysqld}(93)
          |-{mysqld}(94)
          |-{mysqld}(95)
          |-{mysqld}(96)
...
```

и для mysql-test2:

```
root@fadd0e0e631d:/# pstree -p
bash(1)---pstree(17)
```

Отсюда мы видим, что во втором случае база данных MySQL запущена не была. Посмотрим также на вывод docker top:

```
[root@centos7 ~]# docker top mysql-test
UID                 PID                 PPID                C
STIME               TTY                 TIME                CMD
polkitd             13573               13559               0
05:21               ?                   00:00:06            mysqld
[root@centos7 ~]# docker top mysql-test2
UID                 PID                 PPID                C
STIME               TTY                 TIME                CMD
root                13961               13952               0
05:38               pts/1               00:00:00            /bin/bash
```

Изоляция контейнеров

В предыдущей главе мы описали технологии изоляции контейнеров в GNU/Linux. Теперь, когда у нас есть тестовая система, посмотрим на них на практике. Запустим контейнер. В данном случае не принципиально, какой, пусть это будет контейнер с СУБД MySQL:

```
[root@centos7 ~]# docker run --name mysql-test -e MYSQL_ROOT_PASSWORD=docker
-d mysql
...
ae079446926a8d3ded3222c2984ea145216a412f153fdd213eb69c4cc47cfd21
```

При помощи команды docker inspect узнаем идентификатор процесса контейнера в операционной системе хоста:

```
[root@centos7 ~]# docker inspect  mysql-test
[
    {
        "Id":
"ae079446926a8d3ded3222c2984ea145216a412f153fdd213eb69c4cc47cfd21",
...
        "State": {
            "Status": "running",
            "Running": true,
...
            "Pid": 2264,
...
```

В данном примере это 2264. Для начала посмотрим на пространства имен контейнера. Это можно сделать при помощи команды lsns:

```
[root@centos7 ~]# lsns -p 2264
        NS TYPE  NPROCS   PID USER    COMMAND
4026531837 user     155     1 root    /usr/lib/systemd/systemd --switched-
root --system --deserialize 22
4026532349 mnt        1  2264 polkitd mysqld
4026532350 uts        1  2264 polkitd mysqld
4026532351 ipc        1  2264 polkitd mysqld
4026532352 pid        1  2264 polkitd mysqld
4026532354 net        1  2264 polkitd mysqld
```

По столбцу NPROCS (число процессов) видно, что данный процесс является единственным в пространствах имен mnt, uts, ipc, pid и net, а пространство имен user не используется – в нем все процессы системы. Второй способ проверить используемые пространства имен – это обратиться к виртуальной файловой системе /proc:

```
[root@centos7 ~]#  ls -l /proc/2264/ns
total 0
lrwxrwxrwx. 1 polkitd input 0 Mar  5 10:29 ipc -> ipc:[4026532351]
lrwxrwxrwx. 1 polkitd input 0 Mar  5 10:25 mnt -> mnt:[4026532349]
lrwxrwxrwx. 1 polkitd input 0 Mar  5 10:25 net -> net:[4026532354]
lrwxrwxrwx. 1 polkitd input 0 Mar  5 10:29 pid -> pid:[4026532352]
```

```
lrwxrwxrwx. 1 polkitd input 0 Mar  510:29 user -> user:[4026531837]
lrwxrwxrwx. 1 polkitd input 0 Mar  510:29 uts -> uts:[4026532350]
```

При помощи команды nsenter можно «зайти» в заданное пространство имен. Например, можно посмотреть IP-адрес контейнера командой ip addr в сетевом пространстве имен процесса 2264:

```
[root@centos7 ~]# nsenter -t 2264 --net ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
6: eth0@if7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
UP group default
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.17.0.2/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe11:2/64 scope link
       valid_lft forever preferred_lft forever
```

Полезным будет также знать команду unshare, которая позволяет запустить процесс в новых пространствах имен:

```
[root@centos7 ~]# unshare --ipc --uts --net --mount bash &
[2] 3063
```

Теперь перейдем к cgroup (контрольным группам). Как практически все вещи, связанные с ядром в GNU/Linux, информацию по cgroup можно обнаружить в файловой системе /proc:

```
[root@centos7 ~]# cat /proc/2264/cgroup
11:blkio:/system.slice/docker-ae079446926a8d3ded3222c2984ea145216a412f153fdd2
13eb69c4cc47cfd21.scope
10:pids:/system.slice/docker-ae079446926a8d3ded3222c2984ea145216a412f153fdd21
3eb69c4cc47cfd21.scope
9:net_prio,net_cls:/system.slice/docker-ae079446926a8d3ded3222c2984ea145216a4
12f153fdd213eb69c4cc47cfd21.scope
8:freezer:/system.slice/docker-ae079446926a8d3ded3222c2984ea145216a412f153fdd
213eb69c4cc47cfd21.scope
7:memory:/system.slice/docker-ae079446926a8d3ded3222c2984ea145216a412f153fdd2
13eb69c4cc47cfd21.scope
6:perf_event:/system.slice/docker-ae079446926a8d3ded3222c2984ea145216a412f153
fdd213eb69c4cc47cfd21.scope
5:devices:/system.slice/docker-ae079446926a8d3ded3222c2984ea145216a412f153fdd
213eb69c4cc47cfd21.scope
4:hugetlb:/system.slice/docker-ae079446926a8d3ded3222c2984ea145216a412f153fdd
```

```
213eb69c4cc47cfd21.scope
3:cpuacct,cpu:/system.slice/docker-ae079446926a8d3ded3222c2984ea145216a412f15
3fdd213eb69c4cc47cfd21.scope
2:cpuset:/system.slice/docker-ae079446926a8d3ded3222c2984ea145216a412f153fdd2
13eb69c4cc47cfd21.scope
1:name=systemd:/system.slice/docker-ae079446926a8d3ded3222c2984ea145216a412f1
53fdd213eb69c4cc47cfd21.scope
```

Либо более удобным способом, командой systemd-cgls:

```
[root@centos7 ~]# systemd-cgls
...
└─system.slice
  ├─docker-ae079446926a8d3ded3222c2984ea145216a412f153fdd213eb69c4cc47cfd21.
scope
  │ └─2264 mysqld
...
```

Параметры управления доступны в поддиректориях соответствующим контроллерам, например для памяти:

```
[root@centos7 ~]# ls /sys/fs/cgroup/memory/system.slice/docker-ae079446926a8d
3ded3222c2984ea145216a412f153fdd213eb69c4cc47cfd21.scope/
cgroup.clone_children  memory.kmem.failcnt             memory.kmem.tcp.
limit_in_bytes      memory.max_usage_in_bytes        memory.move_charge_at_
immigrate  memory.stat            tasks
cgroup.event_control   memory.kmem.limit_in_bytes      memory.kmem.tcp.
max_usage_in_bytes  memory.memsw.failcnt             memory.numa_stat
memory.swappiness
cgroup.procs           memory.kmem.max_usage_in_bytes  memory.kmem.tcp.
usage_in_bytes      memory.memsw.limit_in_bytes      memory.oom_control
memory.usage_in_bytes
memory.failcnt         memory.kmem.slabinfo            memory.kmem.usage_
in_bytes          memory.memsw.max_usage_in_bytes  memory.pressure_level
memory.use_hierarchy
memory.force_empty     memory.kmem.tcp.failcnt         memory.limit_in_bytes
memory.memsw.usage_in_bytes      memory.soft_limit_in_bytes       notify_on_
release
```

Последней системой, чье использование мы продемонстрируем для контейнеров, будет SELinux:

```
[root@centos7 ~]# ps auxZ | grep 2264
system_u:system_r:container_t:s0:c513,c827 polkitd 22640.4  2.1136093641648 8
? Ssl 10:25   0:27 mysqld
```

Как видно из вывода команды ps с ключом Z, контейнер запускается в домене container_t. Процессы этого домена имеют доступ только к файлам типа container_*_t:

```
[root@centos7 ~]# ls -Z -d /var/lib/docker/
drwx--x--x. root root system_u:object_r:container_var_lib_t:s0 /var/lib/
docker/
```

Если посмотреть на три числа после имени домена в выводе команды ps, то мы увидим, что задействована мультикатегорийная безопасность (Multi Category Security, MCS), являющаяся подмножеством многоуровневой безопасности (Multi Level Security, MLS). Так осуществляется защита между контейнерами.

Политика MLS базируется на формальной модели Bell-LaPadula. В терминах этой модели все субъекты (для простоты – процессы) и объекты (файлы) имеют свой уровень допуска. Субъект с определенным уровнем допуска имеет право читать и создавать (писать/обновлять) объекты с тем же уровнем допуска. Кроме того, он имеет право читать менее секретные объекты и создавать объекты с более высоким уровнем. Субъект никогда не сможет создавать объекты с уровнем допуска ниже, чем он сам имеет, а также прочесть объект более высокого уровня допуска. По-английски это формулируется намного короче: «write up, read down» и «no write down, no read up».

Для поддержки MLS традиционный контекст SELinux, состоящий из трех частей: пользователь, роль и тип, – был дополнен уровнем допуска и включает два уровня безопасности (security level):

```
user:role:type: sensitivity[:category,…][- sensitivity[:category,…]]
```

Уровень допуска состоит из диапазонов чувствительности данных (например, «секретно», «ДСП») и категорий данных («отдел кадров», «отдел финансовой отчетности»).

Первым указывается обязательный текущий уровень (low или current), затем через символ дефиса – наивысший разрешенный уровень (high или clearance). Каждый из двух возможных уровней безопасности включает в себя обязательную часть – чувствительность (sensitivity) данных и ноль или больше категорий (category). Sensitivity у нас всегда будет s0. Категории обозначаются как c0, c1 … c1024. С учетом использования двух категорий мы получаем лимит примерно в 500 000 контейнеров на хост.

Управление состоянием контейнеров

Команда docker ps показывает состояние запущенных и поставленных на паузу контейнеров:

```
[root@centos7 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                 NAMES
fadd0e0e631d        mysql               "docker-entrypoint..."   15 minutes
ago      Up 15 minutes       3306/tcp, 33060/tcp   mysql-test2
4b5193027fc7        mysql               "docker-entrypoint..."   32 minutes
ago      Up 32 minutes       3306/tcp, 33060/tcp   mysql-test
```

С ключом -a мы можем увидеть список всех контейнеров, вне зависимости от их текущего состояния:

```
[root@centos7 ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS                          PORTS                 NAMES
fadd0e0e631d        mysql               "docker-entrypoint..."   17 minutes
ago      Up 17 minutes                 3306/tcp, 33060/tcp   mysql-test2
9156f422a633        ubuntu              "/bin/bash"              32 minutes
ago      Exited (130) 29 minutes ago                            epic_spence
4b5193027fc7        mysql               "docker-entrypoint..."   33 minutes
ago      Up 33 minutes                 3306/tcp, 33060/tcp   mysql-test
30f81da2f29b        mysql               "docker-entrypoint..."   41 minutes
ago      Exited (1) 41 minutes ago                              cocky_ritchie
```

Контейнеры могут находиться в нескольких состояниях. При помощи опции --filter можно управлять выводом команды ps, фильтруя вывод списка контейнеров по их состоянию. В качестве значения опции можно передавать created, restarting, running, paused и exited. Например:

```
[root@centos7 ~]# docker ps --filter status=exited
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS                          PORTS               NAMES
9156f422a633        ubuntu              "/bin/bash"              34 minutes
ago      Exited (130) 31 minutes ago                          epic_spence
30f81da2f29b        mysql               "docker-entrypoint..."   42 minutes
ago      Exited (1) 42 minutes ago                            cocky_ritchie
```

Перечислим основные состояния контейнеров.

Исполняется (running) – контейнер работает. В выводе docker ps вы увидите статус «Up» и время, в течение которого он исполняется.

Создан (created) – контейнер создан, но в настоящий момент не выполняется. Такое состояние будет у контейнера после команды docker create. Еще один пример, когда контейнер оказывается в этом состоянии:

```
[root@centos7 ~]# docker run --name test alpine bash
```

```
...
/usr/bin/docker-current: Error response from daemon: oci runtime error:
container_linux.go:247: starting container process caused "exec: \"bash\":
executable file not found in $PATH".
[root@centos7 ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS                              PORTS                NAMES
cb9aaa0d0791        alpine              "bash"                   49 seconds
ago      Created                                                 test
...
```

Что мы увидели из вывода этих двух команд? Пользователь попытался создать контейнер test из образа Alpine Linux [1] и хотел в качестве запущенной команды получить командный интерпретатор bash. Данной команды в образе нет, соответственно, контейнер до запуска не дошел.

Завершил исполнение (exited) – контейнер завершил исполнение. Модифицируем наш пример:

```
[root@centos7 ~]# docker run --name test2 alpine busybox
BusyBox v1.28.4 (2018-12-0615:13:21 UTC) multi-call binary.
...
[root@centos7 ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS                          PORTS                  NAMES
5d562c1bd2ba        alpine              "busybox"                38 seconds
ago      Exited (0) 37 seconds ago                            test2
cb9aaa0d0791        alpine              "bash"                   3 minutes
ago       Created                                              test
...
```

В отличие от bash, команда busybox, в образе из которой запускался контейнер, присутствует. Успешно завершив выполнение команды busybox, контейнер завершил работу. Тот же результат мы получаем, если для работающего контейнера дадим команду docker stop. Второй способ остановить контейнер – docker kill. По умолчанию эта команда отправляет сигнал SIGKILL, однако при помощи опции -s можно отправить контейнеру и другие стандартные сигналы. Заново запустить остановленный контейнер можно командой docker start. В общем случае остановленный контейнер от созданного отличается тем, что первый уже когда-то запускался и на файловой ситеме остановленного контейнера могут быть уже произведены какие-то изменения. Файловая же система созданного контейнера аналогична образу, из которого создавался контейнер.

Поставлен на паузу (paused) – процесс контейнера остановлен, но существует. Поставить контейнер на паузу можно при помощи команды docker pause. При этом Docker использует контрольную группу freezer, для того чтобы «заморозить» процессы в контейнере.

Проведем простой эксперимент. Запустим контейнер и убедимся, что он работает:

```
[root@centos7 ~]# docker run -it --name ubuntu ubuntu /bin/bash
root@42382bf10d08:/#
```

Контейнер с идентификатором 42382bf10d08 запущен. Откроем на узле вторую консоль и проверим статус контейнера:

```
[root@centos7 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                 NAMES
42382bf10d08        ubuntu              "/bin/bash"              50 seconds
ago      Up 49 seconds                              ubuntu
```

Теперь поставим его на паузу:

```
[root@centos7 ~]# docker pause ubuntu
ubuntu
[root@centos7 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS                        PORTS                NAMES
42382bf10d08        ubuntu              "/bin/bash"              About a
minute ago   Up About a minute (Paused)                         ubuntu
```

Убедимся, что виртуальная файловая система для контроллера freezer смонтирована:

```
[root@centos7 ~]# mount | grep freezer
cgroup on /sys/fs/cgroup/freezer type cgroup
(rw,nosuid,nodev,noexec,relatime,seclabel,freezer)
```

Посмотрим содержимое поддиректории system.slice:

```
[root@centos7 ~]# ls /sys/fs/cgroup/freezer/system.slice/
cgroup.clone_children  cgroup.procs
freezer.parent_freezing  freezer.state      tasks
cgroup.event_control   docker-42382bf10d08b9160493a2b214ade03ed01d7cda3c416f1
107b79d8fe7a5c77a.scope  freezer.self_freezing    notify_on_release
```

Мы видим единственную виртуальную поддиректорию docker-*, соответствующую контейнеру для контроллера freezer. Очевидно, другие контейнеры не запущены. Как вы видите, идентификатор контейнера 42382bf10d08 также присутствует в имени виртуальной поддиректории. Проверим, что в настоящий момент контейнер «разморожен» (THAWED):

```
[root@centos7 ~]# docker unpause ubuntu
ubuntu
[root@centos7 ~]# cat /sys/fs/cgroup/freezer/system.slice/docker-42382bf10d08
b9160493a2b214ade03ed01d7cda3c416f1107b79d8fe7a5c77a.scope/freezer.state
THAWED
```

Теперь поставим контейнер на паузу и убедимся, что это реализовано средствами контрольных групп:

```
[root@centos7 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS                  PORTS               NAMES
42382bf10d08        ubuntu              "/bin/bash"         5 minutes ago
Up 5 minutes (Paused)                       ubuntu
[root@centos7 ~]# cat /sys/fs/cgroup/freezer/system.slice/docker-42382bf10d08
b9160493a2b214ade03ed01d7cda3c416f1107b79d8fe7a5c77a.scope/freezer.state
FROZEN
```

Перезапускается (restarting) – контейнер перезапускается. Остановленный контейнер можно рестартовать. Рестарт означает, что контейнер заново запустится с тем же идентификатором и настройками сети, что были до остановки. Однако это будет уже другой процесс с другим PID.

«Умер» (dead) – контейнер перестал функционировать вследствие сбоя.

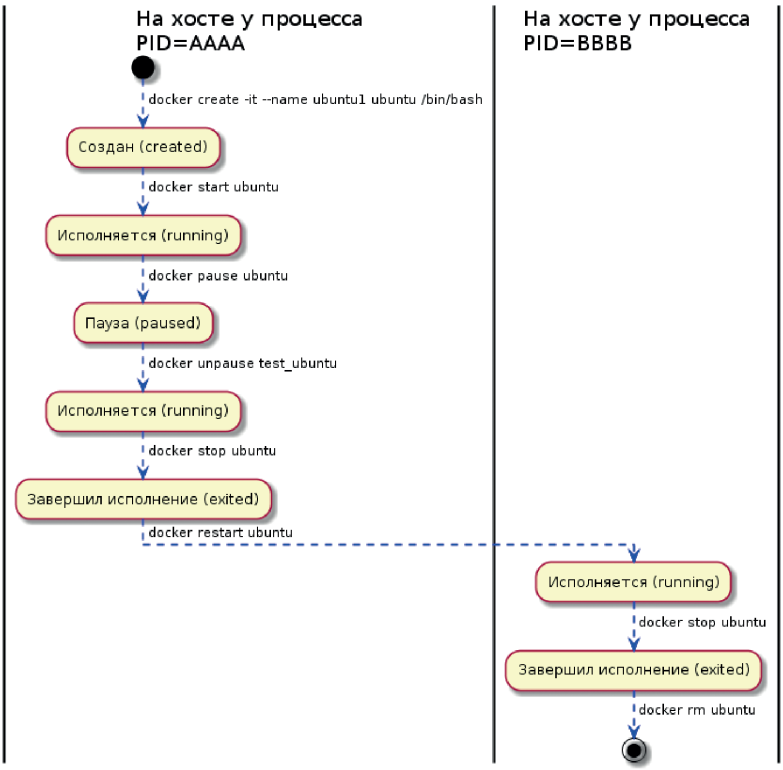


Рис. 2.2 ❖ Пример изменения состояния контейнера

На рис. 2.2 представлен пример перехода контейнера из состояния в состояние. Обратите внимание, что на рисунке показано то, что после рестарта контейнера создается новый процесс. Изначальный процесс контейнера завершил работу после команды docker stop.

Последняя команда на рисунке – docker rm – впервые встречается в книге. Это команда удаления контейнера. После удаления самого контейнера можно удалить и образ, из которого он был запущен. Запущенные контейнеры удалять нельзя.

Если необходимо произвести действия сразу со всеми контейнерами или образами, можно в командах docker images и docker ps использовать опцию -q. При этом будут выведены только идентификаторы контейнеров или образов:

```
$ docker ps -aq
9b0117ecb82d
00e5695fa62f
f568491c665c
$ docker images -q
0766572b4bac
104bec311bcd
594dc21de8de
```

Далее можно подставить вывод этих команд в команды, манипулирующие соответственно контейнерами и образами. В следующем примере удаляются все контейнеры, а затем все образы на узле:

```
$ docker rm $(docker ps -aq)
9b0117ecb82d
00e5695fa62f
f568491c665c
$ docker rmi $(docker images -q)
Untagged: docker.io/alpine:latest
Untagged: docker.io/alpine@
sha256:a4104316f43c73146f1c0af4747d88047a808e58238bcad6506a7fbbf3b30b90
Deleted:
sha256:0766572b4bacfaee9a8eb6bae79e6f6dbcdfac0805c7c6ec8b6c2c0ef097317a
Deleted:
sha256:7cbcbac42c44c6c38559e5df3a494f44987333c8023a40fec48df2fce1fc146b
...
```

Обмен данными с контейнером по сети

Мы научились запускать контейнер с демоном, работающим внутри контейнера. Но толку от такого контейнера немного. Нужно также уметь подключаться к службе из внешней сети. Один из возможных способов – это использование проброса портов из контейнеров на хост. Давайте поэкспериментируем с более наглядным примером – веб-сервером Apache. Страничка официального образа на Docker Hub расположена по адресу [2]. Из описания можно узнать, что корневая директория для веб-сервера /usr/local/apache2/htdocs/.

Загрузим образ и запустим с новой для нас опцией -p:

```
[root@centos7 ~]# docker run -d -p 8888:80 --name my-httpd httpd
Unable to find image 'httpd:latest' locally
Trying to pull repository docker.io/library/httpd ...
..
Status: Downloaded newer image for docker.io/httpd:latest
9f7a39182f9f626a5fd0651d41f26d5d0a540a40d19c8d35970b4c1d3a687642
```

Данная опция позволяет перенаправить tcp-порт 80 контейнера на 8888-й порт хоста. Данный проброс портов будет также отображаться в выводе команды docker ps:

```
[root@centos7 ~]# docker ps
CONTAINER ID        IMAGE               COMMAND              CREATED
STATUS              PORTS                  NAMES
9f7a39182f9f        httpd               "httpd-foreground"   About a minute
ago   Up About a minute   0.0.0.0:8888->80/tcp   my-httpd
```

Организуется это при помощи правил брандмауэра:

```
[root@centos7 ~]# iptables -L DOCKER -t nat
Chain DOCKER (2 references)
target     prot opt source               destination
RETURN     all  --  anywhere             anywhere
DNAT       tcp  --  anywhere             anywhere             tcp dpt:8888
to:172.17.0.2:80
```

В нашем примере у контейнера IP-адрес 172.17.0.2. Протестируем работоспособность веб-сервера. Обратимся на порт 8888 хоста либо по его IP-адресу (в примере 10.0.2.15), либо на localhost:

```
[user@centos7 ~]$ curl http://10.0.2.15:8888
<html><body><h1>It works!</h1></body></html>
```

Отлично! Контейнер доступен из внешнего мира. Попробуем заменить стандартное сообщение своим, изменив index.html:

```
[root@centos7 ~]# docker exec -it my-httpd bash
root@9f7a39182f9f:/usr/local/apache2# echo "My Apache server" > /usr/local/
apache2/htdocs/index.html
root@9f7a39182f9f:/usr/local/apache2# exit
exit
[root@centos7 ~]# curl http://10.0.2.15:8888
My Apache server
```

В качестве альтернативы можно возложить ответственность на выбор порта на сам Docker, отдав команду docker run с ключом -P:

```
[root@centos7 ~]# docker run -d -P --name my-httpd2 httpd
94193722e4f37a1c81562d02fe76e51444f179b96df3a114f194c89643f4a474
```

В этом случае узнать порт, который был выделен контейнеру, можно командой docker port:

```
[root@centos7 ~]# docker port my-httpd2
80/tcp -> 0.0.0.0:32768
```

```
[root@centos7 ~]# curl http://10.0.2.15:32768
<html><body><h1>It works!</h1></body></html>
```

В данном примере порт 32768 был выбран самим движком Docker. По умолчанию Docker берет порт из диапазона «эфемерных портов», задаваемого через параметр ядра GNU/Linux:

```
[root@centos7 ~]# cat /proc/sys/net/ipv4/ip_local_port_range
32768   60999
```

Порт 80 был определен автором образа. Как это задается, мы изучим, когда будем разбирать описание образов при помощи файла Dockerfile.

Просмотр информации о контейнере

Познакомимся с тем, как получить информацию о контейнере. Посмотрим на вывод команды docker inspect. Вывод команды в формате JSON отображен ниже:

```
[root@centos7 ~]# docker inspect my-httpd2 | nl
     1  [
     2      {
     3          "Id":
"94193722e4f37a1c81562d02fe76e51444f179b96df3a114f194c89643f4a474",
     4          "Created": "2019-01-01T10:24:10.264127822Z",
     5          "Path": "httpd-foreground",
     6          "Args": [],
     7          "State": {
     8              "Status": "running",
     9              "Running": true,
    10              "Paused": false,
    11              "Restarting": false,
    12              "OOMKilled": false,
    13              "Dead": false,
    14              "Pid": 4312,
    15              "ExitCode": 0,
    16              "Error": "",
    17              "StartedAt": "2019-01-01T10:24:10.49016784Z",
    18              "FinishedAt": "0001-01-01T00:00:00Z"
    19          },
    20          "Image":
"sha256:ef1dc54703e2daec032eb84e78f88bec496090d098efdbac08f7406473af2bd1",
    21          "ResolvConfPath": "/var/lib/docker/
containers/94193722e4f37a1c81562d02fe76e51444f179b96df3a114f194c89643f4a474/
resolv.conf",
    22          "HostnamePath": "/var/lib/docker/
```

```
containers/94193722e4f37a1c81562d02fe76e51444f179b96df3a114f194c89643f4a474/
hostname",
    23          "HostsPath": "/var/lib/docker/
containers/94193722e4f37a1c81562d02fe76e51444f179b96df3a114f194c89643f4a474/
hosts",
    24          "LogPath": "",
    25          "Name": "/my-httpd2",
    26          "RestartCount": 0,
    27          "Driver": "overlay2",
    28          "MountLabel": "system_u:object_r:svirt_sandbox_
file_t:s0:c12,c428",
    29          "ProcessLabel": "system_u:system_r:svirt_lxc_
net_t:s0:c12,c428",
    30          "AppArmorProfile": "",
    31          "ExecIDs": null,
    32          "HostConfig": {
    33              "Binds": null,
    34              "ContainerIDFile": "",
    35              "LogConfig": {
    36                  "Type": "journald",
    37                  "Config": {}
    38              },
    39              "NetworkMode": "default",
    40              "PortBindings": {},
    41              "RestartPolicy": {
    42                  "Name": "no",
    43                  "MaximumRetryCount": 0
    44              },
    45              "AutoRemove": false,
    46              "VolumeDriver": "",
    47              "VolumesFrom": null,
    48              "CapAdd": null,
    49              "CapDrop": null,
    50              "Dns": [],
    51              "DnsOptions": [],
    52              "DnsSearch": [],
    53              "ExtraHosts": null,
    54              "GroupAdd": null,
    55              "IpcMode": "",
    56              "Cgroup": "",
    57              "Links": null,
    58              "OomScoreAdj": 0,
    59              "PidMode": "",
    60              "Privileged": false,
    61              "PublishAllPorts": true,
    62              "ReadonlyRootfs": false,
    63              "SecurityOpt": null,
    64              "UTSMode": "",
```

```
    65              "UsernsMode": "",
    66              "ShmSize": 67108864,
    67              "Runtime": "docker-runc",
    68              "ConsoleSize": [
    69                  0,
    70                  0
    71              ],
    72              "Isolation": "",
    73              "CpuShares": 0,
    74              "Memory": 0,
    75              "NanoCpus": 0,
    76              "CgroupParent": "",
    77              "BlkioWeight": 0,
    78              "BlkioWeightDevice": null,
    79              "BlkioDeviceReadBps": null,
    80              "BlkioDeviceWriteBps": null,
    81              "BlkioDeviceReadIOps": null,
    82              "BlkioDeviceWriteIOps": null,
    83              "CpuPeriod": 0,
    84              "CpuQuota": 0,
    85              "CpuRealtimePeriod": 0,
    86              "CpuRealtimeRuntime": 0,
    87              "CpusetCpus": "",
    88              "CpusetMems": "",
    89              "Devices": [],
    90              "DiskQuota": 0,
    91              "KernelMemory": 0,
    92              "MemoryReservation": 0,
    93              "MemorySwap": 0,
    94              "MemorySwappiness": -1,
    95              "OomKillDisable": false,
    96              "PidsLimit": 0,
    97              "Ulimits": null,
    98              "CpuCount": 0,
    99              "CpuPercent": 0,
   100              "IOMaximumIOps": 0,
   101              "IOMaximumBandwidth": 0
   102          },
   103          "GraphDriver": {
   104              "Name": "overlay2",
   105              "Data": {
   106                  "LowerDir": "/var/lib/docker/
overlay2/e96c392484ac51f3bbafd8c06ed6ca9d1e6a90655beadedc
bd2049b1288ba39b-init/diff:/var/lib/docker/overlay2/d75fe7
1f16c45e1a4c1906056a304c1541e2cadf8af1f271a5ec5177f9c60a77/
diff:/var/lib/docker/overlay2/
b2d65ab1b2528edffd9b361ab2c2e5a057257e03d54c646f8b5a52e76c1f7bab/
diff:/var/lib/                      docker/
```

```
overlay2/20e1bab4b8d2b14e6ee78e38ae96bf62390993271598fb05f749db752bc892dd/
diff:/var/lib/docker/overlay2/
db910d2c3124ce5421aa468218685868c15c201179b0cca9edfa72eb4
17d2906/diff:/var/lib/docker/
overlay2/3e21a9b9e887b93f7a7c1c2cc17947d65b12a72ef15c0bcd67d435588048cc12/
diff",
   107                 "MergedDir": "/var/lib/docker/overlay2/
e96c392484ac51f3bbafd8c06ed6ca9d1e6a90655beadedcbd2049b1288ba39b/merged",
   108                 "UpperDir": "/var/lib/docker/overlay2/
e96c392484ac51f3bbafd8c06ed6ca9d1e6a90655beadedcbd2049b1288ba39b/diff",
   109                 "WorkDir": "/var/lib/docker/overlay2/
e96c392484ac51f3bbafd8c06ed6ca9d1e6a90655beadedcbd2049b1288ba39b/work"
   110             }
   111         },
   112         "Mounts": [],
   113         "Config": {
   114             "Hostname": "94193722e4f3",
   115             "Domainname": "",
   116             "User": "",
   117             "AttachStdin": false,
   118             "AttachStdout": false,
   119             "AttachStderr": false,
   120             "ExposedPorts": {
   121                 "80/tcp": {}
   122             },
   123             "Tty": false,
   124             "OpenStdin": false,
   125             "StdinOnce": false,
   126             "Env": [
   127                 "PATH=/usr/local/apache2/bin:/usr/local/sbin:/usr/
local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
   128                 "HTTPD_PREFIX=/usr/local/apache2",
   129                 "HTTPD_VERSION=2.4.37",
   130                 "HTTPD_
SHA256=3498dc5c6772fac2eb7307dc7963122ffe243b5e806e0be4fb51974ff759d726",
   131                 "HTTPD_PATCHES=",
   132                 "APACHE_DIST_URLS=https://www.apache.org/dyn/closer.
cgi?action=download&filename= \thttps://www-us.apache.org/dist/ \thttps://
www.apache.org/dis                    t/ \thttps://archive.apache.org/dist/"
   133             ],
   134             "Cmd": [
   135                 "httpd-foreground"
   136             ],
   137             "ArgsEscaped": true,
   138             "Image": "httpd",
   139             "Volumes": null,
   140             "WorkingDir": "/usr/local/apache2",
   141             "Entrypoint": null,
```

```
   142             "OnBuild": null,
   143             "Labels": {}
   144         },
   145         "NetworkSettings": {
   146             "Bridge": "",
   147             "SandboxID":
"f39ab69f3d0b37119acf17839b419b28dfcf5ca91b523767cc64a5c33ec0ce98",
   148             "HairpinMode": false,
   149             "LinkLocalIPv6Address": "",
   150             "LinkLocalIPv6PrefixLen": 0,
   151             "Ports": {
   152                 "80/tcp": [
   153                     {
   154                         "HostIp": "0.0.0.0",
   155                         "HostPort": "32768"
   156                     }
   157                 ]
   158             },
   159             "SandboxKey": "/var/run/docker/netns/f39ab69f3d0b",
   160             "SecondaryIPAddresses": null,
   161             "SecondaryIPv6Addresses": null,
   162             "EndpointID":
"bbbbd3c0b03a9623286c6e181932ce8d0d562dc5ca65311cceab6500e48304c2",
   163             "Gateway": "172.17.0.1",
   164             "GlobalIPv6Address": "",
   165             "GlobalIPv6PrefixLen": 0,
   166             "IPAddress": "172.17.0.3",
   167             "IPPrefixLen": 16,
   168             "IPv6Gateway": "",
   169             "MacAddress": "02:42:ac:11:00:03",
   170             "Networks": {
   171                 "bridge": {
   172                     "IPAMConfig": null,
   173                     "Links": null,
   174                     "Aliases": null,
   175                     "NetworkID":
"4a0d40a16e1bb040c722b4dec7667bb40eed68447e4ebcc64321e35fcac5207e",
   176                     "EndpointID":
"bbbbd3c0b03a9623286c6e181932ce8d0d562dc5ca65311cceab6500e48304c2",
   177                     "Gateway": "172.17.0.1",
   178                     "IPAddress": "172.17.0.3",
   179                     "IPPrefixLen": 16,
   180                     "IPv6Gateway": "",
   181                     "GlobalIPv6Address": "",
   182                     "GlobalIPv6PrefixLen": 0,
   183                     "MacAddress": "02:42:ac:11:00:03"
   184                 }
   185             }
```

```
186          }
187      }
188  ]
```

Опишем ряд полученных параметров:

- строка 3 – идентификатор контейнера;
- строка 4 – время и дата создания контейнера;
- строки с 7 по 19 – информация о статусе контейнера. Текущий статус "running" и PID на хосте – 4312;
- строка 20 – хеш по алгоритму SHA256 образа, из которого запущен контейнер. Используйте команду docker images --digests для его вывода;
- строки 21–23 – расположение на файловой системе хоста файлов resolv.conf, hostname и hosts контейнера;
- строки 28–29 – метка SELinux файловой системы и контекст процесса;
- строки 120–122 – объявленные доступными снаружи порты. В данном случае только один – http;
- строки 126–133 – переменные окружения;
- строки 134–136 – определение запускаемой команды в контейнере;
- строка 178 – IP-адрес контейнера.

Для того чтобы вывести только часть информации, можно воспользоваться шаблоном языка Go, используя опцию -f. Например, для того чтобы получить IP-адрес контейнера, можно воспользоваться командой:

```
[root@centos7 ~]# docker inspect -f '{{ .NetworkSettings.IPAddress}}' my-
httpd2
172.17.0.3
```

Подключение к контейнеру постоянного хранилища

Мы научились запускать контейнеры, однако если вы удалите контейнер, ваши изменения в index.html будут потеряны. На странице с описанием httpd на Dockrer Hub приведен пример использования контейнера:

```
$ docker run -dit --name my-apache-app -v "$PWD":/usr/local/apache2/htdocs/
httpd:2.4
```

Из нового тут опция -v. Эта опция позволяет смонтировать директорию хоста внутри контейнера. Целевая директория в примере – /usr/local/apache2/htdocs/, а монтируемая – текущая рабочая, путь которой содержится в переменной окружения $PWD. Если в команде опустить целевую директорию, то Docker создаст ее в /var/lib/docker/volumes/. Используем предложенный синтаксис, создав index.html в специально выделенной директории:

```
[root@centos7 ~]# mkdir mywww
[root@centos7 ~]# echo "My Apache server – 3" > mywww/index.html
[root@centos7 ~]# docker run -d -p 8889:80 -v /home/andrey/mywww:/usr/local/
apache2/htdocs/ --name my-httpd3 httpd
643010e49c8366751083fe4ca5998e3f31980634c8bba43240e82a25e12cd076
[root@centos7 ~]# curl http://10.0.2.15:8889
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>403 Forbidden</title>
</head><body>
<h1>Forbidden</h1>
<p>You don't have permission to access /
on this server.<br />
</p>
</body></html>
```

Похоже, что у нас проблемы с разрешениями. Дело в том, что в CentOS, а также в ряде других дистрибутивов по умолчанию используется система мандатного контроля доступа SELinux. Для того чтобы контейнер или виртуальная машина получила доступ к файловой ситеме хоста, нужно задать правильный тип для соответствующих файлов и директорий:

```
[root@centos7 ~]# chcon -R -t container_file_t mywww/
[root@centos7 ~]# ls -lZ
drwxr-xr-x. root root unconfined_u:object_r:container_file_t:s0 mywww
```

Повторяем эксперимент:

```
[root@centos7 ~]# docker stop my-httpd3
my-httpd3
[root@centos7 ~]# docker rm my-httpd3
my-httpd3
[root@centos7 ~]# docker run -d -p 8889:80 -v /root/mywww:/usr/local/apache2/
htdocs/ --name my-httpd3 httpd
a3055552ab1b0003d2414450ec20dc7f7ef910185356b861e5a1c0948060cd28
[root@centos7 ~]# curl http://10.0.2.15:8889
My Apache server – 3
```

На этот раз веб-сервер успешно отобразил наш файл index.html. Когда директория контейнера, в которую монтируется директория хоста, существует, ее содержимое заменяется, но не удаляется.

Полезным приемом может оказаться использование выделенных контейнеров только под хранение данных. Идея заключается в том, что у нас будет остановленный контейнер, тома которого будут смонтированы в другой контейнер. При этом работающий контейнер с приложением можно удалять и создавать заново столько раз, сколько необходимо.

Проиллюстрируем на примере. Создадим контейнер для хранения данных под именем my-data, смонтировав в директорию контейнера /usr/local/apache2/htdocs/ директорию хоста с файлом index.html:

```
[root@centos7 ~]# docker run -v /root/mywww:/usr/local/apache2/htdocs/ --name
my-data httpd echo "Data container"
Data container
[root@centos7 ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS                      PORTS                   NAMES
c7268bae0b3a        httpd               "echo 'Data contai..."   7 seconds
ago       Exited (0) 6 seconds ago                            my-data
```

Контейнер my-data остановлен. При помощи опции --volumes-from во время запуска другого контейнера можно смонтировать тома из первого. Проверим это при помощи тестового контейнера test:

```
[root@centos7 ~]# docker run --volumes-from my-data --name test httpd cat /
usr/local/apache2/htdocs/index.html
My Apache server - 3
```

Теперь тестовый контейнер нам не нужен. Удалим его и запустим контейнер с веб-сервером, который будет использовать том с контейнера my-data:

```
[root@centos7 ~]# docker rm test
test
[root@centos7 ~]# docker run -d -p 8899:80 --volumes-from my-data --name my-
httpd-new httpd
431e5018f550ad4d921ccb0bdc26635ece43767c622fc36368d826d68ffb1607
[root@centos7 ~]# curl http://10.0.2.15:8899
My Apache server - 3
```

Ну и убедимся при помощи docker inspect, из какого контейнера используются тома:

```
[root@centos7 ~]# docker inspect -f '{{ .HostConfig.VolumesFrom}}' my-httpd-
new
[my-data]
```

Если теперь удалить контейнер my-data, используемый контейнером my-httpd-new том с index.html удален не будет. Удалить тома вместе с контейнером можно, используя опцию -v команды docker rm и только если тома не используются другими контейнерами.

При запуске контейнера может быть полезной опция --rm. Контейнер, запущенный с такой опцией, удаляется сразу после остановки. Покажем на примере контейнера, предназначенного исключительно для архивирования данных.

Создадим директорию для резервного хранения данных:

```
[root@centos7 ~]# mkdir wwwbackup
[root@centos7 ~]# chcon -R -t container_file_t  /root/wwwbackup
```

Теперь запустим контейнер, единственная цель которого – сохранить данные контейнера в директории хоста. В нашем случае данные – это файл index.html:

```
[root@centos7 ~]# docker run --rm --volumes-from my-httpd-new -v /root/
wwwbackup:/backup httpd cp  /usr/local/apache2/htdocs/index.html /backup
[root@centos7 ~]# ls wwwbackup/
index.html
```

Как мы видим, контейнер сделал свое дело.

Публикация образов в реестре на примере Docker Hub

В этой главе мы подробнее поговорим про работу с образами и реестрами. Ранее мы уже познакомились с тем, как загружать публично доступные образы с Docker Hub. Теперь поговорим о том, как помещать образы в Docker Hub.

На момент написания книги Docker Hub, помимо возможности загружать в него публично доступные образы, бесплатно предоставлял возможность создания одного закрытого репозитория. Для этого необходимо зарегистрироваться на сайте. На рис. 2.3 приведен внешний вид интерфейса панели управления. Подробное рассмотрение Docker Hub выходит за рамки рассматриваемого в книге материала. Мы только познакомимся с тем, как загружать образы в свой репозиторий на Docker Hub, а в главах, посвященных Kubernetes, создадим свой собственный репозиторий.

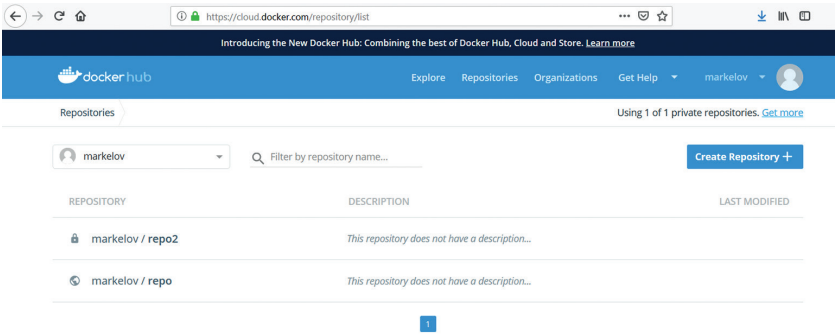


Рис. 2.3 ❖ Внешний вид интерфейса панели управления Docker Hub

Давайте, прежде чем загружать образ в репозиторий, создадим этот образ. В дальнейшем мы научимся создавать образы «правильно» при помощи Dockerfile, а сейчас воспользуемся быстрым вариантом, заодно познакомившись с командами docker diff и docker commit.

Свой образ мы создадим из официального образа Fedora 29, добавив в него исключительно полезную утилиту figlet, которая позволяет рисовать в консоли баннеры при помощи псевдографики. Если вы знакомы с утилитой banner, то figlet – это ее продвинутая версия.

Запустим контейнер с Fedora:

```
[root@centos7 ~]# docker run -it --name figlet fedora:29 bash
[root@79bb49a7863c /]#
```

Затем добавим пакет figlet при помощи пакетного менеджера dnf, заменившего yum в последних версиях Fedora:

```
[root@79bb49a7863c /]# dnf -y install figlet
```

Теперь мы можем рисовать важные послания потомкам:

```
[root@79bb49a7863c /]# figlet Docker was here $(date +%d.%m.%y)
```

Пример отработки команды приведен на рис. 2.4.

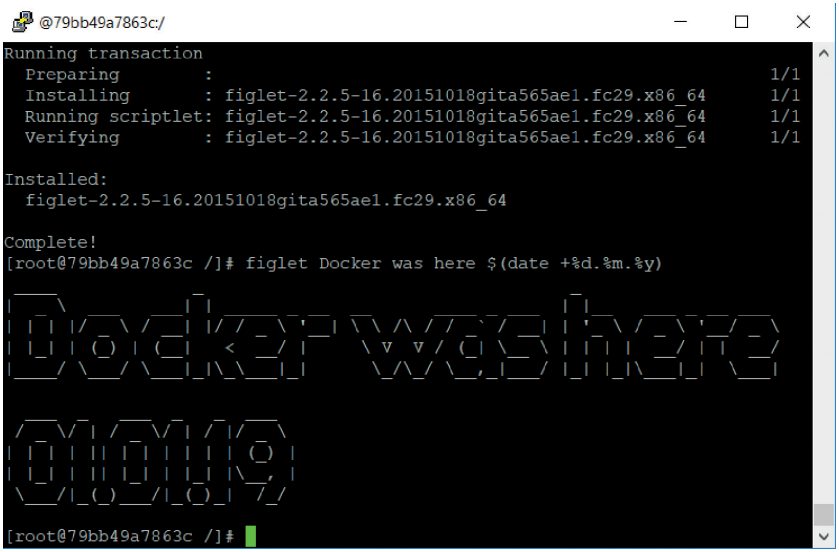


Рис. 2.4 ❖ Результат работы полезной утилиты figlet

Выйдем из контейнера, завершив работу с bash:

```
[root@79bb49a7863c /]# exit
exit
[root@centos7 ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS                      PORTS               NAMES
79bb49a7863c        fedora:29           "bash"              3 minutes ago
Exited (130) 8 seconds ago                      figlet
```

Теперь познакомимся с командой docker diff, которая показывает изменения в файловой системе контейнера по сравнению с оригинальным образом:

```
[root@centos7 ~]# docker diff figlet
C /etc
C /etc/ld.so.cache
C /root
A /root/.bash_history
```

```
C /run
D /run/secrets
C /tmp
C /usr
C /usr/bin
A /usr/bin/chkfont
A /usr/bin/figlet
...
```

В выводе команды A означает, что файл или директория была добавлена, C – изменена, а D означает удаленный файл или директорию. Теперь создадим новый образ с нашими изменениями при помощи команды docker commit:

```
[root@centos7 ~]# docker commit -a 'Andrey' -m 'figlet added' figlet
sha256:00a0ccaf0a61fe87f766acc2df266141f93583d30a38b260f9deb1fd7b44ecb1
```

Проверим список образов в локальном кеше и убедимся, что создан новый. В примере его идентификатор 00a0ccaf0a61:

```
[root@centos7 ~]#  docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
<none>              <none>              00a0ccaf0a61        26 seconds ago
478 MB
docker.io/fedora    29                  25e6809f6fab        2 weeks ago
274 MB
```

Перед тем как опубликовать образ в реестр, ему необходимо присвоить имя и тег. Синтаксис команды следующий:

```
# docker tag образ[:тег] [сервер][имя пользователя]имя[:тег]
```

Мы не будем указывать имя сервера, используя по умолчанию Docker Hub. Также если бы мы не указали тег, то был бы использован тег latest.

```
[root@centos7 ~]# docker tag 00a0ccaf0a61 markelov/figlet:1.0
```

Проверим снова список образов в кеше:

```
[root@centos7 ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
markelov/figlet     1.0                 00a0ccaf0a61        About a minute
ago   478 MB
docker.io/fedora    29                  25e6809f6fab        2 weeks ago
274 MB
```

Осталось два действия: при помощи команды docker login ввести имя и пароль с Docker Hub:

```
[root@centos7 ~]# docker login
Login with your Docker ID to push and pull images from Docker Hub. If you
don't have a Docker ID, head over to https://hub.docker.com to create one.
Username: markelov
Password: ********
Login Succeeded
```

и при помощи команды docker push опубликовать тестовый образ:

```
[root@centos7 ~]# docker push markelov/figlet:1.0
The push refers to a repository [docker.io/markelov/figlet]
ec1264a072b1: Pushed
29395e075665: Mounted from library/fedora
1.0: digest:
sha256:9d8edd351e782d2d348cced4b9c230f4e883a7b36b1053db1fc49b51aa743c95 size:
742
```

Теперь можно для чистоты эксперимента удалить контейнер и все образы в локальном кеше:

```
[root@centos7 ~]# docker rm figlet
figlet
[root@centos7 ~]# docker rmi $(docker images -q)
```

И наконец, запустим контейнер, загрузив наш образ с Docker Hub:

```
[root@centos7 ~]# docker run -it --name figlet markelov/figlet:1.0 figlet
TEST!
Unable to find image 'markelov/figlet:1.0' locally
Trying to pull repository docker.io/markelov/figlet ...
1.0: Pulling from docker.io/markelov/figlet
cd6c8343b590: Pull complete
bb00d7787049: Pull complete
Digest:
sha256:9d8edd351e782d2d348cced4b9c230f4e883a7b36b1053db1fc49b51aa743c95
Status: Downloaded newer image for docker.io/markelov/figlet:1.0

 _____ _____ ____ _____ _
|_   _| ____/ ___|_   _| |
  | | |  _| \___ \ | | | |
  | | | |___ ___) || | |_|
  |_| |_____|____/ |_| (_)
```

Импорт и экспорт образов контейнеров

Хотя публикация образов контейнеров в реестр является рекомендуемым методом распространения образов, рассмотрим еще две альтернативы:

- экспорт контейнера в tar-архив командой docker export;
- сохранение образа контейнера в tar-архив командой docker save.

Начнем с экспорта и потом рассмотрим, чем отличается сохранение. Экспериментировать будем с официальным образом MySQL с Docker Hub. Для начала загрузим в локальный кеш образ, который потянет за собой родительские:

```
[root@centos7 ~]# docker pull mysql
Using default tag: latest
Trying to pull repository docker.io/library/mysql ...
latest: Pulling from docker.io/library/mysql
177e7ef0df69: Pull complete
...
Digest:
sha256:196c04e1944c5e4ea3ab86ae5f78f697cf18ee43865f25e334a6ffb1dbea81e6
Status: Downloaded newer image for docker.io/mysql:latest
```

При помощи новой для нас команды docker history посмотрим созданные слои файловой системы образа, а также команды создания образа, которые не вызывали создания нового уровня. Это, кстати, хороший способ познакомиться с тем, что внутри образа, не запуская его при этом. Подробнее о содержимом, выводимом docker history, мы поговорим в главе, посвященной созданию образов при помощи Dockerfile. Сейчас мы просто отметим, что нам доступна «многослойная» файловая система образа и метаданные:

```
[root@centos7 ~]# docker history mysql
IMAGE               CREATED             CREATED BY
SIZE                COMMENT
102816b1ee7d        4 days ago          /bin/sh -c #(nop)  CMD ["mysqld"]
0 B
<missing>           4 days ago          /bin/sh -c #(nop)  EXPOSE 3306 33060
0 B
<missing>           4 days ago          /bin/sh -c #(nop)  ENTRYPOINT
["docker-ent...   0 B
<missing>           4 days ago          /bin/sh -c ln -s usr/local/bin/
docker-entr...   34 B
...
```

Запустим контейнер из образа, поскольку команда docker export работает с контейнером. При этом не важно, запущен или остановлен контейнер на момент экспорта:

```
[root@centos7 ~]# docker run -d --name mysql-db -e MYSQL_ROOT_PASSWORD=docker
mysql
42cb85f43f089549ee5f58f63d79c04050117920c586c10700a8327af427902e
```

Экспортируем контейнер в tar-архив:

```
[root@centos7 ~]# docker export -o mysql-db.tar mysql-db
[root@centos7 ~]# ls -l mysql-db.tar
-rw-------. 1 root root 484373504 Jan  2 07:23 mysql-db.tar
```

И заново импортируем полученный образ под именем mysql-db-from-exp:1.0:

```
[root@centos7 ~]# docker import mysql-db.tar mysql-db-from-exp:1.0
sha256:b4beff9d8ecdf8c2fed69d6646763e710bd3b4d2f122a83cc62eb474206f077b
```

Проверим, что образ появился в локальном кеше:

```
[root@centos7 ~]# docker images
REPOSITORY                  TAG                 IMAGE ID            CREATED
SIZE
mysql-db-from-exp           1.0                 b4beff9d8ecd        8 seconds
ago       479 MB
docker.io/markelov/figlet   1.0                 8328e380fd5c        About an
hour ago   478 MB
docker.io/mysql             latest              102816b1ee7d        4 days ago
486 MB
```

И что метаданные утеряны:

```
[root@centos7 ~]# docker history mysql-db-from-exp:1.0
IMAGE             CREATED             CREATED BY        SIZE
COMMENT
b4beff9d8ecd      55 seconds ago                        479 MB
Imported from -
```

Фактически tar-архив представляет собой полный слепок файловой системы контейнера, в чем можно убедиться:

```
[root@centos7 ~]# tar xvf mysql-db.tar
```

Второй вариант, сохранение образа, отличается результатом. Сохраним образ, удалим работающий контейнер и образ, который сохранили:

```
[root@centos7 ~]# docker save -o mysql-db2.tar docker.io/mysql
[root@centos7 ~]# docker kill  mysql-db
```

```
mysql-db
[root@centos7 ~]# docker rm mysql-db
mysql-db
[root@centos7 ~]# docker rmi docker.io/mysql
Untagged: docker.io/mysql:latest
Untagged: docker.io/mysql@
sha256:196c04e1944c5e4ea3ab86ae5f78f697cf18ee43865f25e334a6ffb1dbea81e6
Deleted:
sha256:102816b1ee7d6f5943c251647275f0d112f4617bb4ab3f7583206404b7834732
...
```

Загрузим сохраненный образ из архива mysql-db2.tar. Заметьте, нам не нужно указывать имя образа или теги, вся информация присутствует в самом архиве:

```
[root@centos7 ~]# docker load -i mysql-db2.tar
7b4e562e58dc: Loading layer [==================================================
==>] 58.44 MB/58.44 MB
f01f1a2037eb: Loading layer [==================================================
==>] 338.4 kB/338.4 kB
b718c6b527ab: Loading layer [==================================================
==>] 10.44 MB/10.44 MB
...
```

Если теперь провертеть наличие слоев и метаданных командой docker history, мы увидим, что все сохранено:

```
[root@centos7 ~]# docker history docker.io/mysql
IMAGE             CREATED           CREATED BY
SIZE                COMMENT
102816b1ee7d      4 days ago        /bin/sh -c #(nop)  CMD ["mysqld"]
0 B
<missing>         4 days ago        /bin/sh -c #(nop)  EXPOSE 3306 33060
0 B
<missing>         4 days ago        /bin/sh -c #(nop)  ENTRYPOINT ["docker-
ent...   0 B
<missing>         4 days ago        /bin/sh -c ln -s usr/local/bin/docker-
entr...   34 B
<missing>         4 days ago        /bin/sh -c #(nop) COPY
file:1667e4be6bef31...   6.53 kB
...
```

Раскрыв архив, можно убедиться, что данные и слои сохранены отдельно:

```
[root@centos7 ~]# tar xvf mysql-db2.tar
0237d6c035f0c19e46c85b13363cb4655989289ed3a9b9803f0d836a99a0778d/
0237d6c035f0c19e46c85b13363cb4655989289ed3a9b9803f0d836a99a0778d/VERSION
0237d6c035f0c19e46c85b13363cb4655989289ed3a9b9803f0d836a99a0778d/json
```

```
0237d6c035f0c19e46c85b13363cb4655989289ed3a9b9803f0d836a99a0778d/layer.tar
102816b1ee7d6f5943c251647275f0d112f4617bb4ab3f7583206404b7834732.json
...
```

Запуск контейнеров при помощи docker и systemd

В конце данной главы мы рассмотрим задачу старта контейнера без использования сервиса Docker напрямую. Поскольку в операционной системе CentOS и так присутствует демон, управляющий запуском процессов в нужной последовательности – systemd, данную работу можно возложить на него.

Для начала запустим контейнер с необходимыми параметрами. Для теста используем MySQL, а имя контейнера mysql-db-sysd в дальнейшем применяем в конфигурационном файле systemd:

```
[root@centos7 ~]# docker run -d --name mysql-db-sysd -p 3306:3306 -e MYSQL_
ROOT_PASSWORD=docker mysql
38ebb48567edcc711037c0f52731aa1f3e7ba965febadb4486190c7bb373ab1c
```

Для того чтобы демон systemd мог изменять конфигурацию cgroups для контейнеров, необходимо изменить переключатель политики SELinux:

```
[root@centos7 ~]# setsebool -P container_manage_cgroup on
```

Далее создадим простейший конфигурационный файл systemd для запуска контейнера:

```
[root@centos7 ~]# cat /etc/systemd/system/mysql-db-sysd.service

[Unit]
Description=MySQL container
After=docker.service

[Service]
Restart=always
ExecStart=/usr/bin/docker start -a mysql-db-sysd
ExecStop=/usr/bin/docker stop -t 2 mysql-db-sysd

[Install]
WantedBy=local.target
```

Конфигурационный файл легко читается и содержит инструкции, как стартовать и останавливать контейнер, а также зависимость от сервиса docker.service. Стартуем и включаем новый сервис:

```
[root@centos7 ~]# systemctl start mysql-db-sysd.service
[root@centos7 ~]# systemctl enable mysql-db-sysd.service
Created symlink from /etc/systemd/system/local.target.wants/mysql-db-sysd.
service to /etc/systemd/system/mysql-db-sysd.service.
```

Ну, и напоследок проверим, что сервис работает стандартным для systemd способом:

```
[root@centos7 ~]# systemctl status mysql-db-sysd.service
• mysql-db-sysd.service - MySQL container
   Loaded: loaded (/etc/systemd/system/mysql-db-sysd.service; enabled; vendor
preset: disabled)
   Active: active (running) since Wed 2019-01-02 10:31:08 EST; 2s ago
 Main PID: 5653 (docker-current)
    Tasks: 4
   Memory: 0B
   CGroup: /system.slice/mysql-db-sysd.service
           └─5653 /usr/bin/docker-current start -a mysql-db-sysd

Jan 02 10:31:09 centos7.test.local docker[5653]: 2019-01-02T15:31:09.067032Z
0 [Warning] [MY-010315] [Server] 'user' entry 'mysql.session@localhost'
ignored in --skip-name-resolve mode.
Jan 02 10:31:09 centos7.test.local docker[5653]: 2019-01-02T15:31:09.067049Z
0 [Warning] [MY-010315] [Server] 'user' entry 'mysql.sys@localhost' ignored
in --skip-name-resolve mode.
Jan 02 10:31:09 centos7.test.local docker[5653]: 2019-01-02T15:31:09.067065Z
0 [Warning] [MY-010315] [Server] 'user' entry 'root@localhost' ignored in
--skip-name-resolve mode.
```

На этом мы закончим знакомство с базовыми командами docker, а в следующей главе научимся создавать контейнеры при помощи инструкций Dockerfile и команды docker build.

Вопросы для самопроверки

1. **Как найти образ на Docker Hub?**

A. docker search <имя>
B. docker find <имя>
C. docker retrieve <имя>
D. docker <имя>

2. **Как удалить образ контейнера из локального кеша?**

A. docker image delete <имя>
B. docker rmi <имя>
C. docker kill <имя>
D. docker delete <имя>

3. **Какие утверждения верны для тега образа (укажите все правильные ответы)?**

A. Нельзя задать более одного тега для образа
B. Тег может состоять из нескольких частей
C. Как минимум должен быть один тег у любого образа

4. **При помощи команды docker exec -it <имя> bash можно открыть интерактивный сеанс работы с bash для любого запущенного образа.**

A. Верно
B. Не верно

Список ссылок

1. https://alpinelinux.org/
2. https://hub.docker.com/_/httpd/

Глава 3.
СОЗДАНИЕ КОНТЕЙНЕРОВ ПРИ ПОМОЩИ DOCKERFILE

В большинстве случаев, если вы запускаете стандартное программное обеспечение наподобие MySQL, вам не нужно создавать образ. Однако для упаковки своих «самописных» программ вам, скорее всего, это понадобится. В предыдущей главе мы научились создавать образы из запущенных контейнеров, используя команду docker commit. В этой главе мы познакомимся с файлом инструкций для сборки контейнеров Dockerfile.

Базовый синтаксис Dockerfile

Dockerfile – это текстовый файл, в котором последовательно записаны команды по созданию образа контейнера. Как правило, одна строка – это одна команда с соответствующими аргументами. Допускаются пустые строки и строки с комментарием. Комментарии начинаются с символа «решетки» – #. Порядок имеет значение, и команды будут исполняться поочередно от первой к последней. При этом каждая инструкция в Dockerfile выполняется независимо.

Первой значимой инструкцией в Dockerfile должна быть инструкция FROM, которая указывает, из какого базового образа создается ваш образ. Мы не будем рассматривать создание базовых образов «с нуля». Для подавляющего большинства задач предпочтительнее использовать готовые базовые образы. Однако мы рассмотрим специальные инструкции, которые нужны, когда наш образ, созданный из базового, используют в качестве базового для «дочернего» образа.

Создадим тестовый Dockerfile, при помощи которого соберем образ с figlet на основе Fedora 29, аналогичный созданному в предыдущей главе книги при помощи docker commit. Для начала нам необходима рабочая директория:

```
[root@centos7 ~]# mkdir figlet && cd figlet
```

Создадим простейший Dockerfile, который будет состоять из трех строк:

```
[root@centos7 figlet]# cat <<EOF >  Dockerfile
> FROM fedora:29
> RUN dnf -y install figlet
> RUN figlet TEST!
> EOF
```

Инструкция FROM задает базовый образ, а инструкции RUN запускают команду на исполнение во время создания образа. Далее можно воспользоваться командой docker build. Общий синтаксис следующий:

```
# docker build -t имя:тег директория
```

Запускаем команду в рабочей директории, где создан Dockerfile:

```
[root@centos7 figlet]# docker build -t markelov/figlet:2.0 .
Sending build context to Docker daemon 2.048 kB
Step 1/3 : FROM fedora:29
Trying to pull repository docker.io/library/fedora ...
29: Pulling from docker.io/library/fedora
cd6c8343b590: Already exists
Digest:
sha256:50dfa96dd7002374d0ad590cef2de46f68730a8d65d5ea6fb72ff8716630c89f
Status: Downloaded newer image for docker.io/fedora:29
 ---> 25e6809f6fab
Step 2/3 : RUN dnf -y install figlet
 ---> Running in 5da1ed3163ae
...
Complete!
 ---> fc10ce94bfea
Removing intermediate container 5da1ed3163ae
Step 3/3 : RUN figlet TEST!
 ---> Running in 4bb07887b3e9
 _____ _____ ____ _____ _
|_   _| ____/ ___|_   _| |
  | | |  _| \___ \ | | | |
  | | | |___ ___) || | |_|
  |_| |_____|____/ |_| (_)

 ---> a9ba301b28c9
Removing intermediate container 4bb07887b3e9
Successfully built a9ba301b28c9
```

Теперь мы можем запускать контейнер и выводить сообщения следующим образом:

```
[root@centos7 ~]# docker run  markelov/figlet:2.0 figlet test
```

Два замечания по поводу нашего первого файла Dockerfile. Во-первых, каждая строка добавляет новый слой образу, и поэтому команды RUN лучше объединить в одну:

```
RUN dnf -y install figlet && figlet Docker was here $(date +%d.%m.%y)
```

Во-вторых, от второй команды нет никакой пользы, поскольку сообщение выводится только во время создания контейнера.

Но что, если мы хотим выводить при помощи figlet строку по умолчанию, когда не указана команда для исполнения? Для этого нам понадобится инструкция ENTRYPOINT, у которой есть два варианта записи:

```
ENTRYPOINT ["команда", "параметр1", "параметр2"]
```

и

```
ENTRYPOINT команда параметр1 параметр2
```

Первый вариант – предпочтительный. Модифицируем Dockerfile и протестируем:

```
[root@centos7 figlet]# cat Dockerfile
FROM fedora:29
RUN dnf -y install figlet
ENTRYPOINT ["figlet", "TEST!"]
[root@centos7 figlet]# docker build -t markelov/figlet:2.1 .
Sending build context to Docker daemon 2.048 kB
Step 1/3 : FROM fedora:29
 ---> 25e6809f6fab
Step 2/3 : RUN dnf -y install figlet
 ---> Using cache
 ---> fc10ce94bfea
Step 3/3 : ENTRYPOINT figlet TEST!
 ---> Running in 2dc9676eb6b9
 ---> 784adfe0f6a8
Removing intermediate container 2dc9676eb6b9
Successfully built 784adfe0f6a8
[root@centos7 figlet]# docker run  markelov/figlet:2.1
 _____ _____ ____ _____ _
|_   _| ____/ ___|_   _| |
  | | |  _| \___ \ | | | |
  | | | |___ ___) || | |_|
  |_| |_____|____/ |_| (_)
```

Файл может содержать только одну инструкцию ENTRYPOINT. Существует также инструкция CMD, которая передает аргументы в инструкцию ENTRYPOINT. Если в файле более одной инструкции CMD, то применяется только последняя. Также можно обойтись и без ENTRYPOINT, только используя CMD, поскольку значение ENTRYPOINT по умолчанию – это /bin/sh -c. Синтаксис CMD в двух вариантах:

```
CMD ["команда", "параметр1", "параметр2"]
```

и

```
CMD команда параметр1 параметр2
```

Для варианта с ENTRYPOINT по умолчанию:

```
CMD ["параметр1", "параметр2"]
```

Пример запуска веб-сервера как демона в контейнере:

```
CMD ["httpd", "-D", "FOREGROUND"]
```

Изучаем инструкции Dockerfile на примерах

Разберем чуть более сложный пример Dockerfile для образа centos/httpd с Docker Hub и на его примере познакомимся с новыми инструкциями. Скопируем репозиторий CentOS-Dockerfiles, содержащий файлы Dockerfile для CentOS, на локальный диск:

```
[root@centos7 ~]# git clone https://github.com/CentOS/CentOS-Dockerfiles.git
Cloning into 'CentOS-Dockerfiles'...
```

Посмотрим на файл CentOS-Dockerfiles/httpd/centos7/Dockerfile и разберем его построчно:

```
 1  FROM centos:7
 2  MAINTAINER The CentOS Project <cloud-ops@centos.org>
 3  LABEL Vendor="CentOS" \
 4        License=GPLv2 \
 5        Version=2.4.6-40
 6  RUN yum -y --setopt=tsflags=nodocs update && \
 7      yum -y --setopt=tsflags=nodocs install httpd && \
 8      yum clean all
 9  EXPOSE 80
10  # Simple startup script to avoid some issues observed with container
restart
11  ADD run-httpd.sh /run-httpd.sh
12  RUN chmod -v +x /run-httpd.sh
13  CMD ["/run-httpd.sh"]
```

Первая строка – это уже известная нам инструкция FROM. В данном случае в качестве базового образа взят CentOS версии 7. Со второй по пятую строку приведены метаданные. Вторая строка содержит инструкцию MAINTAINER, в которой указывается имя и электронный адрес создателя образа. Третья, четвертая и пятая строки представляют собой одну инструкцию LABEL, в которой указаны вендор, лицензия и версия. Обратите внимание, что создатели Dockerfile следуют рекомендованной практике. Вместо трех инструкций LABEL использована одна для уменьшения числа слоев образа. Метки при этом разнесены по строкам при помощи символа «\».

По аналогии вместо трех инструкций RUN в строках с шестой по восьмую приведены команды обновления всех установленных пакетов, установки пакета httpd и очистки кеша yum. Последняя команда необходима для уменьшения размера образа. Для тех же целей используется флаг tsflags=nodocs. При указании данного флага yum не устанавливает пакеты с документацией. Данная рекомендация, как и многие другие по созданию образов для CentOS, приведена в документе [1].

В девятой строке приведена инструкция EXPOSE, которая указывает, что наш сервис использует порт 80. Обратите внимание, что эта инструкция на самом деле не делает данный порт доступным хосту. Инструкция задает метаданные образа, указывающие, какой порт слушает контейнер.

Строка десять – комментарий. В одиннадцатой строке приведена инструкция ADD, которая позволяет добавить файл или директорию в образ. То, что мы добавляем, должно находиться в той же директории, что и сам Dockerfile. Вы можете открыть файл CentOS-Dockerfiles/httpd/centos7/run-httpd.sh и посмотреть содержащиеся в нем команды:

```
#!/bin/bash
# Make sure we're not confused by old, incompletely-shutdown httpd
# context after restarting the container.  httpd won't start correctly
# if it thinks it is already running.
rm -rf /run/httpd/* /tmp/httpd*
exec /usr/sbin/apachectl -DFOREGROUND
```

Далее данный скрипт будет использоваться для запуска веб-сервера. Нужно заметить, что в качестве источника инструкции ADD можно указать URL. Например:

```
ADD http://myserver/file.txt
```

Существует также устаревшая команда COPY, аналогичная ADD, но не принимающая URL в качестве источника.

Строка двенадцать устанавливает исполнимый бит на добавленный файл, и, наконец, в тринадцатой строке указан аргумент для ENTRYPOINT. Поскольку ENTRYPOINT у нас отсутствует, то при старте контейнера из образа будет выполнена команда:

```
/bin/sh -c /run-httpd.sh
```

Создадим образ:

```
[root@centos7 ~]# cd CentOS-Dockerfiles/httpd/centos7/
[root@centos7 centos7]# docker build -t centos/httpd .
Sending build context to Docker daemon 24.58 kB
Step 1/8 : FROM centos:7
...
Step 8/8 : CMD /run-httpd.sh
 ---> Running in 30a0b85e9964
 ---> 9b1602982b9c
Removing intermediate container 30a0b85e9964
Successfully built 9b1602982b9c
```

После чего запустим контейнер и проверим его работоспособность:

```
[root@centos7 centos7]# docker run --name test-httpd -d -p 18888:80
centos/httpd
598c0e821106024d0751c5da5f5c477fccefcd6f8a931424206aca53bd2ace4e

[root@centos7 centos7]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS                   NAMES
598c0e821106        centos/httpd        "/run-httpd.sh"     About a minute
ago   Up About a minute   0.0.0.0:18888->80/tcp   test-httpd

[root@centos7 centos7]# curl localhost:18888
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/
xhtml11/DTD/xhtml11.dtd"><html><head>
<meta http-equiv="content-type" content="text/html; charset=UTF-8">
               <title>Apache HTTP Server Test Page powered by CentOS</title>
...
```

Рассмотрим еще один пример CentOS-Dockerfiles/mariadb/centos7/Dockerfile:

```
1  FROM centos:centos7
2  MAINTAINER The CentOS Project <cloud-ops@centos.org>
3  LABEL Vendor="CentOS"
4  LABEL License=GPLv2
```

```
     5  LABEL Version=5.5.41
     6  LABEL Build docker build --rm --tag centos/mariadb55 .
     7  RUN yum -y install --setopt=tsflags=nodocs epel-release && \
     8      yum -y install --setopt=tsflags=nodocs mariadb-server bind-utils
pwgen psmisc hostname && \
     9      yum -y erase vim-minimal && \
    10      yum -y update && yum clean all
    11  # Fix permissions to allow for running on openshift
    12  COPY fix-permissions.sh ./
    13  RUN ./fix-permissions.sh /var/lib/mysql/   && \
    14      ./fix-permissions.sh /var/log/mariadb/ && \
    15      ./fix-permissions.sh /var/run/
    16  COPY docker-entrypoint.sh /
    17  ENTRYPOINT ["/docker-entrypoint.sh"]
    18  # Place VOLUME statement below all changes to /var/lib/mysql
    19  VOLUME /var/lib/mysql
    20  # By default will run as random user on openshift and the mysql user
(27)
    21  # everywhere else
    22  USER 27
    23  EXPOSE 3306
    24  CMD ["mysqld_safe"]
```

В данном примере нам знакомы все инструкции, за исключением инструкций VOLUME (19 строка) и USER (22 строка).

Приведенная в девятнадцатой строке инструкция VOLUME объявляет точку монтирования в контейнере. Как использовать точку монтирования, мы уже обсудили в разделе «Подключение к контейнеру постоянного хранилища» предыдущей главы.

Инструкция USER в строке двадцать два задает имя пользователя или его UID, который будет использоваться во время запуска контейнера для последующих инструкций.

Также нужно отметить еще не появлявшуюся в примерах инструкцию WORKDIR, которая задает текущую рабочую директорию.

Модифицируем Dockerfile

Далее мы познакомимся с переменными среды на примере работы с MySQL/MariaDB. Соберем образ MariaDB из Dockerfile репозитория CentOS-Dockerfiles:

```
[root@centos7 ~]# cd /root/CentOS-Dockerfiles/mariadb/centos7
[root@centos7 centos7]# docker build -t centos/mariadb .
Sending build context to Docker daemon 28.67 kB
```

```
Step 1/15 : FROM centos:centos7
...
Successfully built c6a1d93c36fa
```

Запустим контейнер без параметров и убедимся, что он завершает работу по причине того, что мы не передали минимально необходимые параметры через переменные окружения контейнера:

```
[root@centos7 centos7]# docker run --name test-mariadb -d centos/mariadb
599c1437a779892491e371875e32fb26b1c995e1fe36123a0390019b5551551f
[root@centos7 centos7]# docker ps -a
CONTAINER ID        IMAGE                  COMMAND                  CREATED
STATUS                      PORTS                    NAMES
599c1437a779        centos/mariadb         "/docker-entrypoin..."   11 seconds
ago      Exited (1) 10 seconds ago                          test-mariadb
598c0e821106        centos/httpd           "/run-httpd.sh"          21 minutes
ago      Up 21 minutes               0.0.0.0:18888->80/tcp   test-httpd
```

Однозначно какую переменную мы не указали, сообщат логи контейнера:

```
[root@centos7 centos7]# docker logs test-mariadb
error: database is uninitialized and MYSQL_ROOT_PASSWORD not set
  Did you forget to add -e MYSQL_ROOT_PASSWORD=... ?
```

Для того чтобы контейнер запустился, как мы знаем из предыдущей главы, нужно указать переменную через опцию -e:

```
[root@centos7 ~]# docker run --name test-mariadb -e MYSQL_ROOT_
PASSWORD=docker -d  mariadb
```

Что, если мы хотим для тестирования задать переменную по умолчанию во время создания контейнера? Необходимо отредактировать Dockerfile и добавить следующую строку:

```
ENV MYSQL_ROOT_PASSWORD=changeme
```

Инструкция ENV позволяет определить переменные окружения. Учтите, что данный пример с паролем, «зашитым» в контейнер, приведен только в учебных целях, и его не следует использовать в «боевой» среде. Также переменные, определенные при помощи ENV, можно использовать во многих других инструкциях, например ADD, EXPOSE, FROM, LABEL, USER, VOLUME.

Пересоберем контейнер:

```
[root@centos7 centos7]# docker build -t centos/mariadb .
```

И запустим его:

```
[root@centos7 centos7]# docker run --name test-mariadb2 -d centos/mariadb
[root@centos7 centos7]# docker ps -a
CONTAINER ID        IMAGE                  COMMAND                  CREATED
STATUS                      PORTS                    NAMES
5fc2fd978def        centos/mariadb         "/docker-entrypoin..."   16 seconds
ago      Up 15 seconds              3306/tcp                 test-mariadb2
599c1437a779        c6a1d93c36fa           "/docker-entrypoin..."   6 minutes
ago       Exited (1) 6 minutes ago                            test-mariadb
598c0e821106        centos/httpd           "/run-httpd.sh"          27 minutes
ago      Up 27 minutes              0.0.0.0:18888->80/tcp    test-httpd
```

На этом мы заканчиваем знакомство с основами создания файлов инструкций Dockerfile. По ссылке [2] доступно справочное руководство по Dockerfile на сайте Docker.

Вопросы для самопроверки

1. Что такое базовый образ?

A. Образ дистрибутива с установленными базовыми библиотеками
B. Образ, из которого создается новый образ
C. Образ, в котором отсутствуют конфигурационные файлы программного обеспечения
D. Образ базы данных

2. Какое утверждение верное?

A. В Dockerfile обязательно наличие инструкции ENTRYPOINT
B. Значение ENTRYPOINT по умолчанию /bin/bash -c
C. В Dockerfile может быть несколько инструкций CMD

3. Инструкция EXPOSE автоматически делает доступным указанный порт контейнера по сети.

A. Верно
B. Не верно

Список ссылок

1. http://docs.projectatomic.io/container-best-practices/
2. https://docs.docker.com/engine/reference/builder

Глава 4.

РАБОТА С КОНТЕЙНЕРАМИ DOCKER БЕЗ ДВИЖКА DOCKER

В предыдущих главах мы рассмотрели, как можно использовать «движок» работы с контейнерами Docker. В этой главе мы познакомимся с инструментами, изначально разработанными для альтернативной среды исполнения контейнеров CRI-O [1]: podman, buildah и skopeo. Все три утилиты не требуют наличия запущенного демона и независимы от Docker.

Введение в podman, buildah и skopeo

Проект CRI-O был инициирован для создания альтернативной Docker «облегченной» среды исполнения контейнеров для системы оркестрации Kubernetes. Мы будем обсуждать Kubernetes позже, но сейчас, для того чтобы перебросить «мост» от среды запуска контейнеров к оркестратору и определить роль рассматриваемых утилит, необходимо кратко коснуться такого понятия, как интерфейс среды исполнения контейнеров – Container Runtime Interface (CRI).

Изначально единственной средой в контексте Kubernetes являлся Docker. Затем к нему прибавился rkt от проекта CoreOS. Параллельно компания Microsoft начала работать над своей средой исполнения контейнеров в Windows. Для унификации взаимодействия между оркестратором контейнеров и средой исполнения и необходим стандартный интерфейс (CRI). CRI-O расшифровывается как OCI-compliant Container Runtime Interface (совместимый с CRI). Рассматриваемые утилиты предназначены заменить интерфейс командной строки docker при работе с CRI-O, но также могут использоваться и без среды исполнения контейнеров. Перечислим их.

Утилита podman позволяет запускать контейнеры и управлять образами контейнеров. Она поддерживает большинство опций команды docker. Главное отличие в том, что podman не требует

демона docker или какой-либо иной запущенной среды выполнения контейнеров. Также данная утилита совместима по структуре каталогов с CRI-O и позволяет работать не только с отдельными контейнерами, но и с pod-модулями (связанными наборами контейнеров).

Утилита skopeo предназначена для работы с реестрами образов и самими образами.

Утилита buildah поддерживает синтаксис Dockerfile и представляет собой альтернативу команды docker build. Опять же, она не требует среды исполнения контейнеров.

Запуск контейнеров при помощи podman

Утилита podman (Pod Manager) копирует большинство команд docker. Команды, не реализованные в podman, относятся к системе оркестрации контейнеров Docker Swarm, и эта тема выходит за пределы рассматриваемых в книге.

Podman изначально входил в проект CRI-O, но затем был вынесен в проект libpod [3]. Непосредственно функционал обеспечивает библиотека, а podman – это интерфейс командной строки. Для упрощения перехода с команды docker в рамках проекта ведется документ со сравнением и эквивалентом команд [3].

Как было сказано выше, podman не использует демон, но для запуска контейнеров вызывает runc [4]. runc – это утилита для запуска контейнеров в соответствии со спецификацией OCI.

Установить podman можно командой

```
[root@centos7 ~]# yum -y install podman
```

Выполняйте предыдущую команду и последующие команды данной главы на системе без установленного движка Docker. Как вы увидите, пакет runc будет установлен по зависимостям. Далее можно попробовать запустить уже знакомые по docker команды и убедиться, что работа с ними совершенно аналогична:

```
[root@centos7 ~]# podman pull haproxy
Trying to pull docker.io/haproxy:latest...Getting image source signatures
...
15dd50e93fe0d1332f4d4e8035b09b945a720f61a2311e80b71b83e038cfa97b
[root@centos7 ~]# podman images
REPOSITORY                  TAG      IMAGE ID       CREATED        SIZE
docker.io/library/haproxy   latest   15dd50e93fe0   40 hours ago   75.3 MB
```

```
[root@centos7 ~]# podman run -it fedora
Trying to pull docker.io/fedora:latest...Getting image source signatures
...
[root@95fc8becc952 /]#
```

Фактически, если вы умеете работать с docker, вы умеете работать с podman. Вы можете установить средствами командной оболочки псевдоним для podman и вовсе забыть, что вы работаете не с docker:

```
[root@centos7 ~]# alias docker=podman
[root@centos7 ~]# docker ps -a
CONTAINER ID  IMAGE                            COMMAND    CREATED
STATUS                      PORTS  NAMES
95fc8becc952  docker.io/library/fedora:latest  /bin/bash  23 minutes ago
Exited (127) 13 minutes ago        tender_gates
```

Запуск pod-модулей при помощи podman

Концепция pod-модулей впервые была введена проектом Kubernetes, и pod-модули в podman очень на них похожи. Pod-модуль – это группа, состоящая из одного или более контейнеров, которые используют общее пространство имен, сеть и тома. Pod-модуль можно рассматривать как «логический сервер».

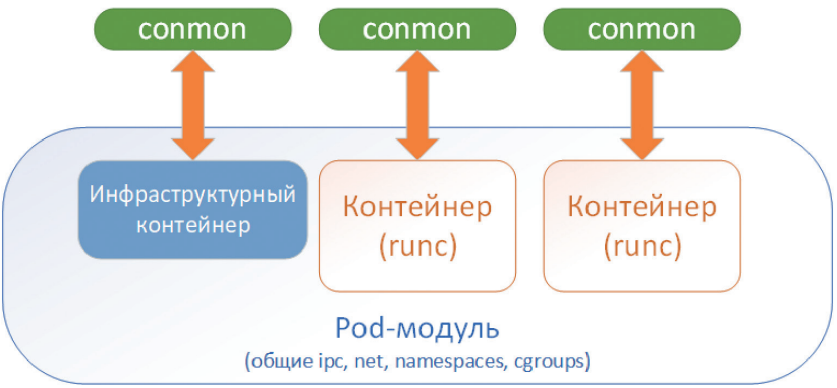


Рис. 4.1 ❖ Pod-модуль в архитектуре podman

Как вы видите в примере на рис. 4.1, у нас имеется три контейнера. Два из них предназначены для полезной нагрузки, третий,

присутствующий в каждом pod-модуле, – это контейнер обеспечения инфраструктуры, или инфра-контейнер. Данный контейнер находится в состоянии паузы, и его задача – «держать» пространство имен pod-модуля. Это позволяет останавливать и перезапускать контейнеры, при этом pod-модуль будет запущенным. Фактически большинство параметров, определяющих pod-модуль, таких как пространство имен ядра, сопоставление портов и контрольные группы, привязано именно к инфра-контейнеру. Отсюда следует также тот факт, что вы не можете поменять эти параметры после запуска pod-модуля без его удаления. Например, вы не сможете добавить новое сопоставление портов для сервиса.

Каждый контейнер на рис. 4.1 привязан к монитору контейнеров (conmon). Это небольшая программа, чья функция заключается в мониторинге главного процесса в контейнере. Также conmon отвечает за привязку терминала tty к контейнеру.

Для работы с pod-модулями необходимо использовать команду pod утилиты podman. Ключевые слова команды pod приведены в табл. 4.1.

Ключевое слово	Назначение
create	Создать новый пустой pod-модуль
inspect	Показать конфигурацию pod-модуля
kill	Отправить указанный сигнал контейнерам. По умолчанию SIGKILL
pause	Поставить один или более pod-модулей на паузу
ps, ls, list	Вывести список pod-модулей
restart	Перезапустить один или более pod-модулей
rm	Удалить один или более pod-модулей
start	Запустить один или более pod-модулей
stats	Вывести статистику по использованию ресурсов контейнерами
stop	Остановить один или более pod-модулей
top	Показать процессы контейнеров в pod-модуле
unpause	Снять с паузы один или более pod-модулей

Таблица 4.1 ❖ Список ключевых слов команды podman pod

Создадим «пустой» pod-молуль, без полезной нагрузки:

```
[root@centos7 ~]# podman pod create --name pod1
8923a66aa6d2e42453edfcf5ae567bfce8c0ed986f57af289ae39fbe0122d0d0
```

Проверим список pod-модулей. Должен присутствовать только один модуль, содержащий один инфраструктурный контейнер:

```
[root@centos7 ~]# podman pod list
POD ID         NAME   STATUS    CREATED          # OF CONTAINERS   INFRA ID
8923a66aa6d2   pod1   Running   11 seconds ago   1
5c166acaab5e
```

Проверим общий список контейнеров в системе. По умолчанию для инфраструктурного контейнера используется образ k8s.gcr.io/pause:

```
[root@centos7 ~]# podman ps -a
CONTAINER ID  IMAGE                 COMMAND  CREATED         STATUS
PORTS  NAMES
5c166acaab5e  k8s.gcr.io/pause:3.1           20 seconds ago  Up 19 seconds
ago         8923a66aa6d2-infra
```

Если в команду добавить ключ «--pod», то в вывод добавится идентификатор pod-модуля, которому принадлежит контейнер:

```
[root@centos7 ~]# podman ps -a --pod
CONTAINER ID  IMAGE                 COMMAND  CREATED         STATUS
PORTS  NAMES               POD
5c166acaab5e  k8s.gcr.io/pause:3.1           46 seconds ago  Up 45 seconds
ago         8923a66aa6d2-infra  8923a66aa6d2
```

Добавим в контейнер pod-модуль при помощи команды run и ключа «--pod»:

```
[root@centos7 ~]# podman run --pod pod1 -d --name my-httpd httpd
41574d9b8fa466ccafc8752a7dd711ccfef2f91b65859fabf6c2bb76ea2e1826
```

Убедимся, что число контейнеров выросло до двух:

```
[root@centos7 ~]# podman pod list
POD ID         NAME   STATUS    CREATED         # OF CONTAINERS   INFRA ID
8923a66aa6d2   pod1   Running   4 minutes ago   2
5c166acaab5e
```

Получим список контейнеров и проверим, что оба принадлежат одному pod-модулю:

```
[root@centos7 ~]# podman ps -a --pod
CONTAINER ID  IMAGE                             COMMAND           CREATED
STATUS             PORTS  NAMES               POD
41574d9b8fa4  docker.io/library/httpd:latest  httpd-foreground  32 seconds
ago  Up 32 seconds ago         my-httpd            8923a66aa6d2
5c166acaab5e  k8s.gcr.io/pause:3.1                              4 minutes ago
Up 4 minutes ago          8923a66aa6d2-infra  8923a66aa6d2
```

Теперь можно остановить подмодуль и удалить его:

```
[root@centos7 ~]# podman pod stop pod1
8923a66aa6d2e42453edfcf5ae567bfce8c0ed986f57af289ae39fbe0122d0d0
[root@centos7 ~]# podman pod rm pod1
failed to delete pod
8923a66aa6d2e42453edfcf5ae567bfce8c0ed986f57af289ae39fbe0122d0d0: pod
8923a66aa6d2e42453edfcf5ae567bfce8c0ed986f57af289ae39fbe0122d0d0 contains
containers and cannot be removed: container already exists
```

Однако, как видите, вторая команда не сработала. После остановки контейнера с рабочей нагрузкой удаление pod-модуля пройдет успешно:

```
[root@centos7 ~]# podman rm 41574d9b8fa4
41574d9b8fa466ccafc8752a7dd711ccfef2f91b65859fabf6c2bb76ea2e1826
[root@centos7 ~]# podman pod rm pod1
8923a66aa6d2e42453edfcf5ae567bfce8c0ed986f57af289ae39fbe0122d0d0
```

Запуск контейнеров при помощи podman и systemd

В конце второй главы мы рассмотрели задачу старта контейнера с помощью systemd и Docker. При помощи podman данная задача решается практически аналогично. Плюсом данного варианта является отсутствие демона docker.

Как и в прошлый раз, запустим контейнер с необходимыми параметрами. Для теста используем MySQL в контейнере с именем mysql-db-sysd:

```
[root@centos7 ~]# podman run -d --name mysql-db-sysd -p 3306:3306 -e MYSQL_
ROOT_PASSWORD=docker mysql
```

Далее создадим простейший конфигурационный файл systemd для запуска контейнера, аналогичный приведенному во второй главе:

```
[root@centos7 ~]# cat /etc/systemd/system/mysql-db-sysd.service

[Unit]
Description=MySQL container
After=network.target

[Service]
Restart=always
ExecStart=/usr/bin/podman start -a mysql-db-sysd
ExecStop=/usr/bin/podman stop -t 2 mysql-db-sysd
```

```
[Install]
WantedBy=local.target
```

Основное отличие данного конфигурационного файла – это использование команды podman, а также зависимость от сервиса network.service. Стартуем и включим новый сервис:

```
[root@centos7 ~]# systemctl start mysql-db-sysd.service
[root@centos7 ~]# systemctl enable mysql-db-sysd.service
Created symlink from /etc/systemd/system/local.target.wants/mysql-db-sysd.
service to /etc/systemd/system/mysql-db-sysd.service.
```

Проверим, что сервис работает:

```
[root@centos7 ~]# systemctl status mysql-db-sysd.service
• mysql-db-sysd.service - MySQL container
   Loaded: loaded (/etc/systemd/system/mysql-db-sysd.service; enabled; vendor
preset: disabled)
   Active: active (running) since Sat 2019-01-2619:10:58 CET; 27s ago
 Main PID: 3024 (podman)
   CGroup: /system.slice/mysql-db-sysd.service
           └─3024 /usr/bin/podman start -a mysql-db-sysd

Jan 2619:10:58 centos7.test.local systemd[1]: Started MySQL container.
Jan 2619:10:58 centos7.test.local systemd[1]: Starting MySQL container...
```

Использование утилиты buildah для создания образов контейнеров

Кратко рассмотрим еще одну утилиту, в данном случае предназначенную для создания OCI-совместимых образов контейнеров без использования Docker. Утилита buildah [5] может заменить команду docker build, а также использоваться интерактивно, без применения Dockerfile при помощи интерфейса командной строки. Кроме того, buildah не включает в образ утилиты, использованные для построения образа, что снижает его размер.

Установить утилиту можно командой

```
[root@centos7 centos7]# yum -y install buildah
```

Поскольку buildah использует ту же структуру каталогов (/var/lib/containers, определяется в конфигурационном файле /etc/containers/storage.conf библиотеки containers/storage), что и podman и CRI-O, команда buildah images покажет образы, с которыми вы уже работали ранее:

```
[root@centos7 centos7]# buildah images
IMAGE ID             IMAGE NAME                                CREATED AT
SIZE
da86e6ba6ca1         k8s.gcr.io/pause:3.1                      Dec 20, 2017
22:30     747 KB
e40405f80704         docker.io/library/httpd:latest            Jan 23, 2019
00:15     137 MB
71b5c7e10f9b         docker.io/library/mysql:latest            Jan 23, 2019
05:23     482 MB
```

Ключевые слова buildah приведены в табл. 4.2. Попробуем самую очевидную команду, позволяющую собрать образ при помощи Dockerfile. Используем уже знакомый пример из репозитория CentOS-Dockerfiles:

```
[root@centos7 centos7]# git clone https://github.com/CentOS/CentOS-
Dockerfiles.git
[root@centos7 ~]# cd /root/CentOS-Dockerfiles/mariadb/centos7
[root@centos7 ~]# buildah bud -t centos/mariadb .
```

Последняя команда при помощи ключевого слова bud (build-using-dockerfile) создает образ. Проверим, что он присутствует на локальном диске:

```
[root@centos7 centos7]# buildah images
IMAGE ID             IMAGE NAME                                CREATED AT
SIZE
...
1e1148e4cc2c         docker.io/library/centos:centos7          Dec 6, 2018
01:21      210 MB
6ee6ff5d242a         localhost/centos/mariadb:latest           Jan 26, 2019
19:47     433 MB
```

Ключевое слово	**Назначение**
add	Добавить файлы в образ
build-using-dockerfile, bud	Создать образ при помощи Dockerfile
commit	Создать образ из запущенного контейнера
config	Обновить настройки образа
containers	Вывести список запущенных контейнеров и их базовых образов
copy	Скопировать в контейнер
from	Создать контейнер на основе базового образа
images	Список образов на локальном диске
inspect	Показать конфигурацию контейнера или образа
mount	Подключить корневую систему запущенного контейнера

pull	Загрузить образ
push	Отправить образ
rename	Переименовать контейнер
rm, delete	Удалить один или более контейнеров
rmi	Удалить один или более образов с локального диска
run	Запустить команду в контейнере
tag	Задать дополнительное имя для локального образа
umount, unmount	Отключить корневую файловую систему контейнера
unshare	Запуск команды в модифицированном пространстве пользователя
version	Показать версию Buildah

Таблица 4.2 ❖ Список ключевых слов команды buildah

Попробуем создать образ интерактивно. Запустим контейнер, используя в качестве базового образа CentOS 7:

```
[root@centos7 ~]# container=$(buildah from centos)
Getting image source signatures
Skipping fetch of repeat blob
sha256:a02a4930cb5d36f3290eb84f4bfa30668ef2e9fe3a1fb73ec015fc58b9958b17
Copying config
sha256:1e1148e4cc2c148c6890a18e3b2d2dde41a6745ceb4e5fe94a923d811bf82ddb
 2.13 KiB / 2.13 KiB [=====================================================
] 0s
Writing manifest to image destination
Storing signatures
```

Мы использовали переменную container, для того чтобы в дальнейшем обращаться к контейнеру по имени. По умолчанию buildah создает имя, добавляя к имени базового образа «-working-container»:

```
[root@centos7 ~]# echo $container
centos-working-container
```

Проверим, что базовый образ присутствует на локальном диске:

```
[root@centos7 ~]# buildah images
IMAGE ID             IMAGE NAME                                CREATED AT
SIZE
...
1e1148e4cc2c         docker.io/library/centos:latest           Dec 6, 2018
01:21      210 MB
```

И что наш рабочий контейнер запущен:

```
[root@centos7 ~]# buildah containers
CONTAINER ID  BUILDER  IMAGE ID     IMAGE NAME
CONTAINER NAME
d4340ef9f232     *     1e1148e4cc2c docker.io/library/centos:latest  centos-
working-container
```

При помощи ключевого слова inspect можно получить информацию о контейнере или образе:

```
[root@centos7 ~]# buildah inspect $container
{
    "Type": "buildah 0.0.1",
    "FromImage": "docker.io/library/centos:latest",
    "FromImageID":
"1e1148e4cc2c148c6890a18e3b2d2dde41a6745ceb4e5fe94a923d811bf82ddb",
    "Config": "{\"architecture\":\"amd64\",\"config\":
  "Manifest": "{\n   \"schemaVersion\": 2,\n   \"mediaType\":
\"application...
  "Container": "centos-working-container",...
    "ContainerID":
"d4340ef9f2324160005b1f8a1671bd0f66cc45afb4f830e82a8cdf4021b4e698",
    "MountPoint": "",
    "ProcessLabel": "system_u:system_r:svirt_lxc_net_t:s0:c68,c866",
    "MountLabel": "system_u:object_r:svirt_sandbox_file_t:s0:c68,c866",
    "ImageAnnotations": null,
    "ImageCreatedBy": "",
    "OCIv1": {
        "created": "2018-12-06T00:21:07.135655444Z",
        "architecture": "amd64",
```

Создадим исполнимый файл, который должен запускаться при старте контейнера. Единственная работа, которую он будет выполнять, – это вывод сообщения «TEST»:

```
[root@centos7 ~]# echo 'echo TEST!!!' > test.sh
[root@centos7 ~]# chmod 755 test.sh
```

Скопируем файл в рабочий контейнер:

```
[root@centos7 ~]# buildah copy $container test.sh /usr/local/bin
d2045e6052749afc9c734fd762be3297739b8d063da7e533011ba84fbc7ae76c
```

Зададим выполнение команды test.sh как команды по умолчанию при старте контейнера (аналог инструкции ENTRYPOINT в Dockerfile):

```
[root@centos7 ~]# buildah config --entrypoint "/bin/sh -c /usr/local/bin/
test.sh" $container
```

И наконец, создадим образ из рабочего контейнера:

```
[root@centos7 ~]# buildah commit $container containers-storage:testcontainer
Getting image source signatures
Skipping fetch of repeat blob
sha256:071d8bd765171080d01682844524be57ac9883e53079b6ac66707e192ea25956
Copying blob
sha256:df5e53c489a096664b303b60c23ba3011a6570a135a520e145b74124e0517ba9
 198 B / 198 B [============================================================
] 0s
Copying config
sha256:9fdbbc71d57a9779358c92bef00f84a9f131cd639327fe4507cfca2be8e8070a
 1.23 KiB / 1.23 KiB [======================================================
] 0s
Writing manifest to image destination
Storing signatures
9fdbbc71d57a9779358c92bef00f84a9f131cd639327fe4507cfca2be8e8070a
```

Проверим, что образ присутствует на локальном диске:

```
[root@centos7 ~]# buildah images
IMAGE ID          IMAGE NAME                                      CREATED AT
SIZE
...
9fdbbc71d57a      docker.io/library/testcontainer:latest      Feb 9, 2019
15:50      210 MB
```

И проверим его работоспособность, запустив новый контейнер при помощи команды podman run:

```
[root@centos7 ~]# podman run testcontainer
TEST!!!
```

Работа с образами при помощи skopeo

Рассмотрим последнюю в этой главе утилиту skopeo [6]. Данная утилита в первую очередь предназначена для работы с образами и реестрами образов (registry). Отличительные особенности skopeo:

- позволяет работать простым пользователем и не требует привилегий root;
- дает возможность копировать образы между двумя реестрами или реестром и локальным диском, минуя локальный;
- поддерживает удаление образов из реестров;
- поддерживает формат Open Container Initiative (OCI);
- поддерживает подписанные образы.

Установить утилиту можно командой

```
[root@centos7 ~]# yum -y install skopeo
```

Приведем несколько примеров использования skopeo. При помощи команды можно посмотреть информацию об образе без его загрузки. Собственно, это был первый функционал, который появился в утилите:

```
[root@centos7 ~]# skopeo inspect docker://docker.io/library/mysql
{
    "Name": "docker.io/library/mysql",
    "Digest":
"sha256:a571337738c9205427c80748e165eca88edc5a1157f8b8d545fa127fc3e29269",
    "RepoTags": [
        "5.5.40",
...
        "8",
        "latest"
    ],
    "Created": "2019-02-06T07:06:04.898919867Z",
    "DockerVersion": "18.06.1-ce",
    "Labels": null,
    "Architecture": "amd64",
    "Os": "linux",
    "Layers": [
"sha256:6ae821421a7debccb4151f7a50dc8ec0317674429bec0f275402d697047a8e96",
...
    ]
}
```

Второй пример показывает, как можно скопировать все уровни заданного контейнера вместе с манифестом в локальную директорию:

```
[root@centos7 ~]# skopeo copy  docker://docker.io/library/mysql dir:/tmp/test
Getting image source signatures
...
Copying config
sha256:81f094a7e4ccc963fde3762e86625af76b6339924bf13f1b7bd3c51dbcfda988
 6.86 KB / 6.86 KB [=========================================================
] 0s
Writing manifest to image destination
Storing signatures
```

Проверим содержимое директории:

```
[root@centos7 ~]# ls /tmp/test
...
```

```
a9e976e3aa6d5d9153f006b807cf8b626371b1e3bd541d806a7e27cb1c40666d  manifest.
json
b60db6d282cd49a5625cc1b572c236758758bbf9b262567eb6c58b05fc99626b  version
b8e2d50f1513ed58fc5b32041fd2640e9d5c76db416d6164fff9dbcc703dd186
```

На этом мы заканчиваем знакомство с утилитами, предназначенными для работы с контейнерами на одном узле, и переходим к системе оркестрации контейнеров.

Вопросы для самопроверки

1. **Какие из утилит не требуют привилегий root (укажите все правильные ответы)?**

A. podman
B. docker
C. buildah
D. skopeo

2. **Какие из утилит можно использовать для создания образа из Dockerfile (укажите все правильные ответы)?**

A. podman
B. docker
C. buildah
D. skopeo

Список ссылок

1. https://cri-o.io/
2. https://github.com/containers/libpod
3. https://github.com/containers/libpod/blob/master/transfer.md
4. https://github.com/opencontainers/runc
5. https://github.com/containers/buildah
6. https://github.com/containers/skopeo

Глава 5.
ВВЕДЕНИЕ В KUBERNETES И УСТАНОВКА КЛАСТЕРА

В данной главе мы познакомимся с архитектурой системы оркестрации контейнеров Kubernetes и создадим кластер, состоящий из трех узлов, для последующих экспериментов в процессе обучения.

Знакомство с Kubernetes

Начиная с этой главы мы переходим к рассмотрению системы оркестрации контейнеров Kubernetes. Kubernetes (от греч. κυβερνήτης, что в переводе означает кормчий) по определению с сайта проекта является «программным обеспечением для автоматического внедрения, масштабирования и управления контейнеризированными приложениями». Также вы можете встретиться с сокращенным написанием названия – K8S. Отличительной особенностью Kubernetes является то, что в основе разработки положен более чем пятнадцатилетний опыт компании Google. В настоящее время разработка управляется специально созданным фондом Cloud Native Computing Foundation (CNCF) [1].

Kubernetes является основой крупнейших облачных инфраструктур. Дадим определение облачным технологиям и месту Kubernetes в них. Устоявшимся в индустрии определением является определение, данное National Institute of Standards and Technology (NIST):

Облачные вычисления – это модель предоставления широкодоступного, удобного доступа по сети к общему пулу настраиваемых вычислительных ресурсов по требованию (к таким как сети, серверы, системы хранения данных, приложения и сервисы). Эти ресурсы оперативно выделяются и освобождаются при минимальных усилиях, затрачиваемых заказчиком на организацию управления и на взаимодействие с поставщиком услуг.

Этой модели присущи пять основных характеристик, три сервисные модели и четыре модели внедрения. В число характеристик входят: самообслуживание, универсальный доступ по сети, общий пул ресурсов, эластичность и учет потребления.

Сервисные модели различаются по границе контроля потребителем услуг предоставляемой инфраструктуры и включают в себя:

- инфраструктуру как сервис (IaaS). В данном случае пользователь получает контроль за всеми уровнями стека программного обеспечения, лежащими выше облачной платформы, а именно: виртуальными машинами, сетями, выделенным пользователю объемом пространства на системе хранения данных (СХД). Тогда пользователь выступает администратором операционной системы и всего, что работает поверх, вплоть до приложений. В качестве примеров платформ, обеспечивающих подобную модель, можно назвать OpenStack, Apache CloudStack, Eucalyptus и OpenNebula;
- программное обеспечение как сервис (SaaS) – в этом случае граница контроля пользователя – само приложение. Пользователь может даже не знать, что такое виртуальная машина или операционная система, он просто работает с приложением. Примеры таких облачных продуктов: Google Docs, Office 365 или, например, Яндекс-почта;
- платформу как сервис (PaaS) – облако, построенное по такой модели, вполне может располагаться «внутри» облака модели IaaS. В этом случае граница контроля пользователя лежит на уровне платформы построения приложений, например сервера приложения, библиотек, среды разработки или базы данных. Пользователь не контролирует и не администрирует виртуальные машины и операционные системы, установленные на них, СХД и сети. Kubernetes является основой крупнейших облачных инфраструктур типа PaaS.

Четыре модели внедрения облачной платформы включают в себя:

- частное облако – вся инфраструктура развернута в центре обработки данных (ЦОД) и служит подразделением одной компании или группы компаний;

- публичное облако – заказчиком облачных услуг может выступать любая компания или даже частное лицо. Это модель внедрения, на которой зарабатывают провайдеры облачных услуг;
- облако сообщества, или общественное облако. Модель, при которой потребителем является сообщество потребителей из организаций, имеющих общие задачи (например, миссии, требований безопасности, политики и соответствия различным требованиям);
- гибридное облако – это комбинация из двух или трех вышеописанных облаков, где разная нагрузка может располагаться, как в частном, публичном или общественном облаке. Как правило, гибридное облако – это больше, чем просто сумма облаков, поскольку ему требуются механизмы и инструменты централизованного управления, распределения и миграции нагрузки между облачными инфраструктурами.

В основу архитектуры Kubernetes были положены разработки проекта Borg – проприетарной системы Google для управления различными облачными приложениями, такими как Google Docs и Gmail. Впервые о Borg было рассказано в 2015 году. И в июле 2015 года была выпущена первая версия Kubernetes. В настоящий момент разработчики придерживаются трехмесячного цикла разработки. Код Kubernetes написан на относительно молодом языке программирования Go, который представляет собой некий гибрид из самых распространенных языков: Java, Python и C++. Крупнейшими компаниями-разработчиками в проекте Kubernetes в настоящее время являются Google и Red Hat. Kubernetes не единственное решение для управления контейнеризированными приложениями (можно также назвать Docker Swarm, Nomad, Rancher), но, видимо, самое успешное и распространенное.

Kubernetes меняет традиционный подход в написании и развертывании приложений с монолитной на микросервисную архитектуру. При таком подходе приложение состоит из отдельных относительно небольших и самостоятельно разрабатываемых частей, возможно, даже написанных на разных языках программирования. У каждого компонента может быть свой жизненный цикл разработки и обновлений. Компоненты при этом взаимодействуют друг с другом посредством концепции сервисов и вызовов API. Для ознакомления с принципами построения высокомасштабируемых

облачных приложений, которые идеально ложатся на инфраструктуру Kubernetes, вы можете изучить принципы «двенадцати факторов» [2]. Сам проект Kubernetes также следует этим принципам.

Рассмотрим архитектуру и компоненты Kubernetes, представленные на рис. 5.1. Это упрощенная диаграмма без учета реализации высокой доступности компонентов. Кластер состоит из как минимум одного управляющего узла и от одного и более вычислительных (или рабочих) узлов, на которых непосредственно запускаются контейнеры. Управляющий узел может быть совмещен с вычислительным, и в самом простом случае все компоненты могут работать в одной виртуальной машине или на одном-единственном сервере. В качестве примера можно привести инструмент Minikube, который позволяет создать такую виртуальную машину в VirtualBox. Возможны варианты также с несколькими управляющими узлами для обеспечения высокой доступности и с единой базой etcd, а также с несколькими управляющими узлами и распределенным кластером etcd.

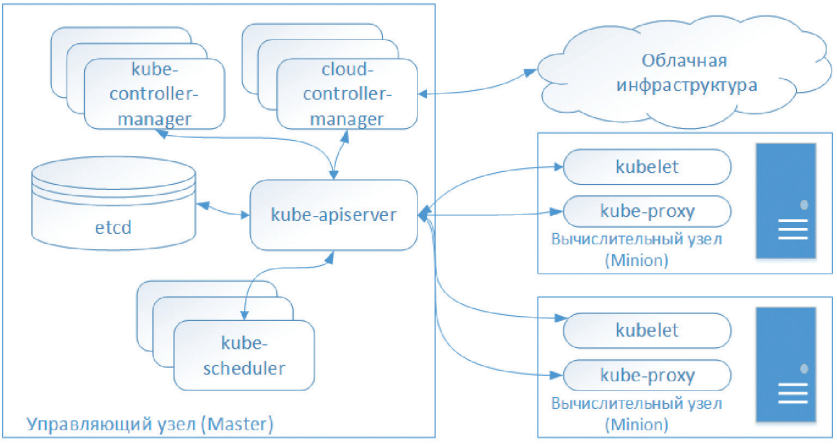


Рис. 5.1 ❖ Упрощенная архитектура Kubernetes

Также обратите внимание, что мы далее будем рассматривать установку управляющих узлов при помощи kubeadm, где сервисы Kubernetes сами работают в контейнерах, однако вполне возможен вариант, когда они работают как обычные демоны GNU/Linux под управлением, например, systemd.

Рассмотрим сервисы управляющего узла.

kube-apiserver – это центральный компонент кластера Kubernetes. Он является связывающим компонентом для всех остальных сервисов. Все взаимодействие как самих компонентов, так и обращение извне к кластеру проходит через kube-apiserver и валидируется им. Это единственный компонент кластера, который общается с базой данных etcd, где хранится состояние кластера.

etcd – распределенное хранилище типа «ключ-значение». Не является «базой данных» в классическом понимании СУБД. Изначально развивалось как составная часть проекта CoreOS. Кластер Kubernetes хранит в etcd всю информацию о состоянии кластера, сервисах, сети и т. д. Доступ к данным осуществляется через REST API. При изменениях записей вместо поиска и изменения старой копии она помечается как устаревшая, а новые значения дописываются в конец. Позже старые значения удаляются специальным процессом.

kube-scheduler – компонент, отвечающий за выбор вычислительного узла, на котором будут запускаться контейнеры (на самом деле pod-модули, но мы рассмотрим, что это такое, позже). Существуют механизмы, которые позволяют вмешаться в этот алгоритм и, например, привязать контейнеры к конкретному узлу. Также на запуск и размещение контейнеров влияют существующие квоты на ресурсы.

kube-controller-manager – компонент, отвечающий за запуск так называемых контроллеров, которые определяют текущее состояние системы. Далее мы рассмотрим примеры контроллеров, таких как, например, Deployment. Фактически kube-controller-manager следит, чтобы кластер и его ресурсы соответствовали заданному состоянию.

cloud-controller-manager – начиная с версии Kubernetes 1.6 этот компонент отвечает за взаимодействие с облачными провайдерами и абстрагирует специфический для конкретного облака код. Ранее за этот функционал отвечал kube-controller-manager.

Перейдем к сервисам вычислительных (управляемых) узлов.

kubelet – данный сервис управляет pod-модулями, основываясь на их спецификации. Сервис взаимодействует с kube-apiserver. Это главный сервис для вычислительных узлов.

kubeproxy – отвечает за сетевое взаимодействие pod-модулей, управляя правилами iptables (таблица nat).

Среда исполнения контейнеров – в нашем демонстрационном стенде мы будем использовать Docker.

Установка локального кластера

На момент написания этой книги последней версией Kubernetes была 1.13. Поскольку эта книга является введением в Kubernetes, различия между релизами с точки зрения изучения системы оркестрации должны быть не очень большими. В книге рассмотрена установка и настройка базового кластера Kubernetes с использованием дистрибутива GNU/Linux CentOS 7 и утилиты kubeadm. Наиболее свежие инструкции по использованию kubeadm должны быть доступны на сайте Kubernetes [3].

Нам понадобится несколько виртуальных машин с оперативной памятью от 2 Гб. Автор в примерах использует кластер, состоящий из трех узлов. Соответственно, для выполнения упражнений вам будет достаточно персонального компьютера с 8–16 Гб оперативной памяти. При желании, безусловно, можно использовать публичное облако, развернув в нем виртуальные машины, и точно таким же способом установить там сервисы Kubernetes вручную. Существует достаточно известный проект «Kubernetes The Hard Way»[4], который содержит инструкции, как это сделать. Кроме того, можно использовать ряд других инструментов: Minikube, kubespray, hyperkube, kops и др. В качестве альтернативы можно рассмотреть кластер, обслуживаемый самим облачным провайдером. Далее мы рассмотрим пример установки Azure Kubernetes Service (AKS) в публичном облаке Microsoft.

Диски всех тестовых виртуальных машин рекомендуется сделать «тонкими». В этом случае размер реально занятого места на файловой системе для каждой виртуальной машины будет от 3 до 6 Гб.

Подготовка операционной системы

Подготовка операционной системы и настройка репозиториев для всех виртуальных машин с одинаковым дистрибутивом GNU/Linux будут выполняться одинаково.

Убедитесь, что отключен swap-раздел операционной системы, иначе во время запуска kubeadm вы получите сообщение об ошибке – о том, что работа со swap не поддерживается:

```
[ERROR Swap]: running with swap on is not supported. Please disable swap
```

Установите для всех виртуальных машин статические адреса. В конфигурационных файлах сетевых адаптеров ifcfg-* в директории /etc/sysconfig/network-scripts/ необходимо поправить параметры BOOTPTOTO, IPADDR0, PREFIX0 и GATEWAY0. Пример файла после редактирования:

```
TYPE="Ethernet"
BOOTPROTO="none"
DEFROUTE="yes"
IPV6INIT="no"
NAME="eth0"
ONBOOT="yes"
IPADDR0="10.0.3.4"
PREFIX0="24"
GATEWAY0="10.0.3.1"
DNS1="8.8.8.8"
DOMAIN="test.local"
```

IP-адресацию вы можете изменить по своему усмотрению, не ориентируясь на выбор автора. Приведенные адреса выбраны лишь в качестве примера. Также убедитесь, что все взаимодействующие виртуальные машины могут разрешать DNS-имена друг друга. Можно поднять локальный DNS-сервер или, что намного проще, прописать на всех узлах имена взаимодействующих с ними лабораторных машин в файл /etc/hosts. Пример для узлов, используемых в следующих главах:

```
127.0.0.1   localhost localhost.localdomain localhost4 localhost4.
localdomain4
::1         localhost localhost.localdomain localhost6 localhost6.
localdomain6
10.0.3.4 master master.test.local
10.0.3.5 node1 node1.test.local
10.0.3.6 node2 node2.test.local
```

Процесс подготовки и установки кластера Kubernetes на CentOS 7 несильно отличается от соответствующих действий для других производных Red Hat Enterprise Linux.

Первое, что необходимо, – установить саму операционную систему. Предполагается, что это не должно вызвать затруднений у читателя. В качестве варианта установки можно выбрать Minimal или Server with GUI.

После установки операционной системы обновите установленные пакеты и перезагрузите операционную систему на всех трех узлах командой

```
# yum -y update && reboot
```

Далее для упрощения отладки мы отключим сервис брандмауэра firewalld. Если вы не планируете отключать брандмауэр, то список используемых входящих TCP-портов приведен в таблице ниже. Начиная с CentOS 7 по умолчанию вместо iptables используется firewalld:

```
# systemctl stop firewalld.service
# systemctl disable firewalld.service
```

Важно отметить, что такая конфигурация не подходит для промышленного применения. На «боевых» узлах брандмауэр должен быть обязательно включен.

Тип узла	TCP-порты	Назначение
Мастер	6443	Сервер API Kubernetes по умолчанию
Мастер	2379-2380	API хранилища etcd
Мастер и вычислительный узел	10250	Kubelet API
Мастер	10251	kube-scheduler
Мастер и вычислительный узел	10252	kube-controller-manager
Мастер и вычислительный узел	10255	Сервер API Kubelet (только чтение)
Рабочий узел	30000-32767	Порты по умолчанию NodePort

Таблица 5.1 ❖ Список входящих TCP-портов кластера Kubernetes

Во время написания книги сервис kubelet еще не поддерживал работу на системах с включенным механизмом мандатного контроля доступа SELinux. Для выключения SELinux необходимо отредактировать файл /etc/sysconfig/selinux, заменив

```
SELINUX=enforcing
```

на

```
SELINUX=disabled
```

Далее необходимо просто перезагрузить систему.

Еще одно из предварительных условий: необходимо убедиться, что пакеты, проходящие через сетевые мосты linux bridge,

обрабатываются iptables. Для этого необходимо установить переменную ядра net.bridge.bridge-nf-call-iptables в 1:

```
# cat <<EOF >  /etc/sysctl.d/k8s.conf
> net.bridge.bridge-nf-call-iptables = 1
> EOF
# sysctl --system
...
* Applying /etc/sysctl.d/k8s.conf ...
* Applying /etc/sysctl.conf ...
```

Устанавливаем на всех трех узлах Docker и запускаем сервис:

```
# yum -y install docker
# systemctl start docker.service
# systemctl enable docker.service
```

Следующее – это добавление репозиториев с пакетами kubeadm, kubelet и kubectl. Kubeadm – это утилита, при помощи которой мы будем развертывать кластер. Kubelet – это сервис, запускаемый на всех узлах и отвечающий за старт pod-модулей и контейнеров. Все службы Kubernetes при установке с помощью kubeadm будут запускаться в pod-модулях, а не непосредственно на управляющих узлах. Наконец, kubectl – это утилита управления Kubernetes.

Создаем на всех узлах файл репозитория /etc/yum.repos.d/kubernetes.repo следующего содержания:

```
[root@master ~]# cat /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg https://
packages.cloud.google.com/yum/doc/rpm-package-key.gpg
```

Установка управляющего узла

Следующие действия выполняем только на управляющем узле. Устанавливаем пакеты:

```
[root@master ~]# yum -y install kubelet kubeadm kubectl
```

Включаем и запускаем службу kubelet:

```
[root@master ~]# systemctl enable kubelet.service
[root@master ~]# systemctl start kubelet.service
```

Теперь kubelet будет рестартовать каждые несколько секунд, ожидая инструкций от kubeadm:

```
[root@master ~]# systemctl status kubelet.service
• kubelet.service - kubelet: The Kubernetes Node Agent
   Loaded: loaded (/etc/systemd/system/kubelet.service; enabled; vendor
preset: disabled)
  Drop-In: /etc/systemd/system/kubelet.service.d
           └─10-kubeadm.conf
   Active: activating (auto-restart) (Result: exit-code) since Sun 2019-01-
06 16:18:51 CET; 7s ago
     Docs: https://kubernetes.io/docs/
  Process: 12203 ExecStart=/usr/bin/kubelet $KUBELET_KUBECONFIG_ARGS
$KUBELET_CONFIG_ARGS $KUBELET_KUBEADM_ARGS $KUBELET_EXTRA_ARGS (code=exited,
status=255)
 Main PID: 12203 (code=exited, status=255)

Jan 06 16:18:51 master.test.local systemd[1]: kubelet.service: main process
exited, code=exited, status=255/n/a

Jan 06 16:18:51 master.test.local systemd[1]: Unit kubelet.service entered
failed state.
Jan 06 16:18:51 master.test.local systemd[1]: kubelet.service failed.
```

Тем временем нам необходимо решить, каким сетевым подключаемым модулем через интерфейс Container Networking Interface (CNI) мы будем пользоваться для организации сети в кластере. В настоящий момент может быть только одна pod-сеть на кластер. В рамках проекта CNI-Genie [5] ведется работа по одновременному использованию нескольких подключаемых модулей. Для нашей тестовой среды мы выберем Flannel, развиваемый проектом CoreOS (в настоящий момент компания, стоящая за проектом, приобретена компанией Red Hat).

Скачаем yml-файл, отвечающий за настройку Flannel (при необходимости установите утилиту wget):

```
[root@master ~]# wget https://raw.githubusercontent.com/coreos/flannel/
master/Documentation/kube-flannel.yml
```

Из этого файла нам необходимо получить сеть, которую Flannel создает по умолчанию для взаимодействия pod-модулей. Адрес сети в формате CIDR нам потребуется во время инициализации кластера:

```
[root@master ~]# grep -i network kube-flannel.yml
      "Network": "10.244.0.0/16",
      hostNetwork: true
```

Запускаем инициализацию управляющего узла с указанием найденного адреса pod-сети кластера по умолчанию:

```
[root@master ~]# kubeadm init --pod-network-cidr 10.244.0.0/16
[init] Using Kubernetes version: v1.13.1
[preflight] Running pre-flight checks
...

Your Kubernetes master has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

You can now join any number of machines by running the following on each node
as root:

  kubeadm join 10.0.3.4:6443 --token 96gi3c.
ax1pp27h13o1ebig --discovery-token-ca-cert-hash
sha256:0d30b4386fd3609ae7a98e00cc249f05d20a22196623dcc5b0707e829ef35a3b
```

Фактически при успешном выполнении команды мы увидим сообщение о последующих шагах в настройке кластера. Переключаемся под учетную запись непривилегированного пользователя (при необходимости создайте его) и копируем конфигурационный файл /etc/kubernetes/admin.conf в домашнюю директорию под именем .kube/config:

```
[root@master ~]# su - user
[user@master ~]$ mkdir .kube
[user@master ~]$ sudo cp -i /etc/kubernetes/admin.conf .kube/config
[user@master ~]$ sudo chown user.user .kube/config
```

Теперь при помощи конфигурационного файла мы можем подключаться и выполнять действия как администратор кластера. Далее нам надо применить конфигурацию Flannel:

```
[user@master ~]$ sudo cp /root/kube-flannel.yml .
[user@master ~]$ kubectl apply -f kube-flannel.yml
clusterrole.rbac.authorization.k8s.io/flannel created
clusterrolebinding.rbac.authorization.k8s.io/flannel created
```

```
serviceaccount/flannel created
configmap/kube-flannel-cfg created
daemonset.extensions/kube-flannel-ds-amd64 created
...
```

Проверяем, что создался новый интерфейс flannel.1:

```
[user@master ~]$ ip a s dev flannel.1
4: flannel.1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state
UNKNOWN group default
    link/ether 42:b9:92:71:56:0e brd ff:ff:ff:ff:ff:ff
    inet 10.244.0.0/32 scope global flannel.1
       valid_lft forever preferred_lft forever
    inet6 fe80::40b9:92ff:fe71:560e/64 scope link
       valid_lft forever preferred_lft forever
```

Выведем список узлов в кластере:

```
[user@master ~]$ kubectl get node
NAME                STATUS   ROLES    AGE   VERSION
master.test.local   Ready    master   12m   v1.13.1
```

Один узел в состоянии Ready. Можно посмотреть подробную информацию об узле:

```
[user@master ~]$ kubectl describe node master.test.local
Name:               master.test.local
Roles:              master
Labels:             beta.kubernetes.io/arch=amd64
                    beta.kubernetes.io/os=linux
                    kubernetes.io/hostname=master.test.local
                    node-role.kubernetes.io/master=
Annotations:        flannel.alpha.coreos.com/backend-data:
{"VtepMAC":"42:b9:92:71:56:0e"}
                    flannel.alpha.coreos.com/backend-type: vxlan
                    flannel.alpha.coreos.com/kube-subnet-manager: true
                    flannel.alpha.coreos.com/public-ip: 10.0.3.4
                    kubeadm.alpha.kubernetes.io/cri-socket: /var/run/
dockershim.sock
                    node.alpha.kubernetes.io/ttl: 0
                    volumes.kubernetes.io/controller-managed-attach-detach:
true
CreationTimestamp:  Sun, 06 Jan 201916:31:50 +0100
Taints:             node-role.kubernetes.io/master:NoSchedule
```

Обратите внимание на строчку Taints. В ней указано, что на нашем узле пользовательские pod-модули не будут запускаться. Это поведение по умолчанию для управляющего узла. В тестовом окружении мы можем снять флаг taint:

```
[user@master ~]$ kubectl taint nodes --all node-role.kubernetes.io/master-
node/master.test.local untainted
```

Таким образом, мы получили функционирующий кластер, состоящий из одного узла. Прежде чем продолжить, рекомендуется установить пакет bash-completion и включить возможность автоматического дополнения командной строки при использовании kubectl. Kubectl самостоятельно может создать скрипт автодополнения. Нам остается только исполнить его и добавить в .bashrc:

```
[user@master ~]$ sudo yum -y install bash-completion
[user@master ~]$ source <(kubectl completion bash)
[user@master ~]$ echo "source <(kubectl completion bash)" >> ~/.bashrc
```

Наконец, проверьте, что все служебные pod-модули запущены:

```
[user@master ~]$ kubectl get pods --all-namespaces
NAMESPACE     NAME                                        READY   STATUS
RESTARTS   AGE
kube-system   coredns-86c58d9df4-qscdh                    1/1     Running
0          19m
kube-system   coredns-86c58d9df4-w8fx5                    1/1     Running
0          19m
kube-system   etcd-master.test.local                      1/1     Running
0          18m
kube-system   kube-apiserver-master.test.local            1/1     Running
0          18m
kube-system   kube-controller-manager-master.test.local   1/1     Running
0          18m
kube-system   kube-flannel-ds-amd64-7vng2                 1/1     Running   0
9m56s
kube-system   kube-proxy-g557g                            1/1     Running
0          19m
kube-system   kube-scheduler-master.test.local            1/1     Running
0          18m
```

Установка рабочих узлов

Следующие действия мы производим на обеих виртуальных машинах: node1 и node2. Точно так же, как и на управляющем узле, установите пакеты сервисов Kubernetes и включите kubelet:

```
[root@node1 ~]# yum -y install kubelet kubeadm kubectl
[root@node1 ~]# systemctl enable kubelet.service
```

На управляющем узле необходимо найти значение токена, который понадобится для добавления нового узла в кластер:

```
[user@master ~]$ kubeadm token list
TOKEN                     TTL       EXPIRES                     USAGES
DESCRIPTION                                                EXTRA GROUPS
96gi3c.ax1pp27h13o1ebig   23h       2019-01-07T16:31:53+01:00
authentication,signing   The default bootstrap token generated by 'kubeadm
init'.   system:bootstrappers:kubeadm:default-node-token
```

Токен по умолчанию действует двадцать четыре часа. Если его срок действия истек, новый токен можно создать командой kubeadm token create. Начиная с версии 1.9 в целях повышения безопасности при подключении новых узлов вам также понадобится хеш сертификата (Discovery Token CA Cert Hash). Получить его можно опять же на управляющем узле, на котором хранится сертификат в файле /etc/kubernetes/pki/ca.crt:

```
[user@master ~]$ openssl x509 -pubkey -in /etc/kubernetes/pki/ca.crt |
openssl rsa -pubin -outform der 2>/dev/null | openssl dgst -sha256 -hex | sed
's/^.* //'
0d30b4386fd3609ae7a98e00cc249f05d20a22196623dcc5b0707e829ef35a3b
```

Далее запускаем команду kubeadm join на обоих вычислительных узлах, указав IP-адрес управляющего узла, токен и хеш:

```
[root@node1 ~]# kubeadm join --token 96gi3c.
ax1pp27h13o1ebig 10.0.3.4:6443 --discovery-token-ca-cert-hash
sha256:0d30b4386fd3609ae7a98e00cc249f05d20a22196623dcc5b0707e829ef35a3b
[preflight] Running pre-flight checks
[discovery] Trying to connect to API Server "10.0.3.4:6443"
[discovery] Created cluster-info discovery client, requesting info from
"https://10.0.3.4:6443"
[discovery] Requesting info from "https://10.0.3.4:6443" again to validate
TLS against the pinned public key
...
This node has joined the cluster:
* Certificate signing request was sent to apiserver and a response was
received.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the master to see this node join the cluster.
```

Проверяем, чтобы оба узла были добавлены в кластер:

```
[user@master ~]$ kubectl get node
NAME               STATUS   ROLES    AGE    VERSION
master.test.local   Ready    master   42m    v1.13.1
node1.test.local    Ready    <none>   2m1s   v1.13.1
node2.test.local    Ready    <none>   50s    v1.13.1
```

Если вам необходимо подключаться к кластеру с узла, отличного от управляющего, скопируйте конфигурационный файл:

```
[user@node1 ~]$ scp root@master:/etc/kubernetes/admin.conf .
[user@node1 ~]$ kubectl --kubeconfig ./admin.conf get nodes
```

Установка веб-консоли Kubernetes

Для удобства управления кластером можно установить веб-консоль Kubernetes. Ее внешний вид приведен на рис. 5.2. Установка достаточно простая. Применяем yaml-файл:

```
[user@master ~]$ kubectl create -f https://raw.githubusercontent.com/
kubernetes/dashboard/master/aio/deploy/recommended/kubernetes-dashboard.yaml
secret/kubernetes-dashboard-certs created
secret/kubernetes-dashboard-csrf created
serviceaccount/kubernetes-dashboard created
role.rbac.authorization.k8s.io/kubernetes-dashboard-minimal created
rolebinding.rbac.authorization.k8s.io/kubernetes-dashboard-minimal created
deployment.apps/kubernetes-dashboard created
service/kubernetes-dashboard created
```

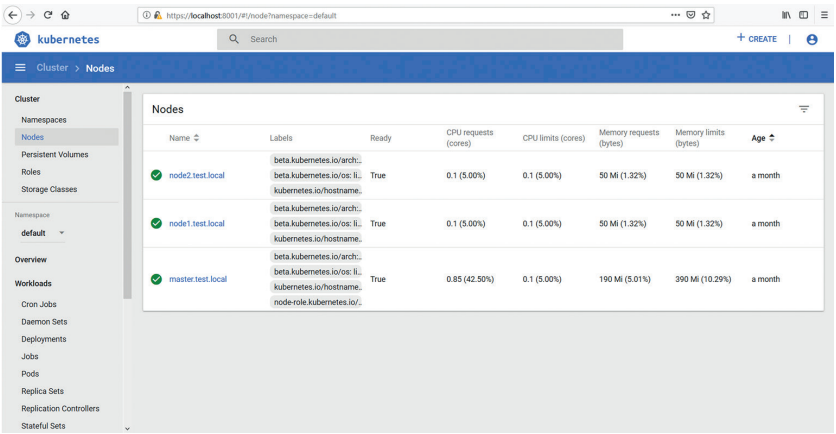


Рис. 5.2 ❖ Внешний вид веб-консоли Kubernetes

Проверим, что запущен соответствующий pod-модуль:

```
[user@master ~]$ kubectl get pods --all-namespaces
NAMESPACE     NAME                                    READY   STATUS
RESTARTS   AGE
kube-system   coredns-86c58d9df4-qscdh                1/1     Running
8          55d
```

```
kube-system   coredns-86c58d9df4-w8fx5                    1/1     Running
8          55d
...
kube-system   kubernetes-dashboard-57df4db6b-x669z        1/1     Running
0          60s
```

Веб-консоль поддерживает авторизацию при помощи токенов (Bearer Token) и файла Kubeconfig. Для демонстрационных целей мы создадим сервисную учетную запись (Service Account) с именем admin-user:

```
[user@master ~]$ cat << EOF > ServiceAccount.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: admin-user
  namespace: kube-system
EOF
[user@master ~]$ kubectl create -f ServiceAccount.yaml
```

Далее необходимо присвоить созданной учетной записи роль cluster-admin:

```
[user@master ~]$ cat << EOF > ClusterRoleBinding.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: admin-user
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: admin-user
  namespace: kube-system
EOF
[user@master ~]$ kubectl create -f  ClusterRoleBinding.yaml
```

Чтобы получить токен, выполним команду:

```
[user@master ~]$ kubectl -n kube-system describe secret $(kubectl -n kube-
system get secret | grep admin-user | awk '{print $1}')
Name:         admin-user-token-f9j68
Namespace:    kube-system
Labels:       <none>
Annotations:  kubernetes.io/service-account.name: admin-user
              kubernetes.io/service-account.uid: 53eda26c-3d9b-11e9-aa0b-
0800279bf286
```

```
Type:  kubernetes.io/service-account-token

Data
====
ca.crt:     1025 bytes
namespace:  11 bytes
token:      eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9...
```

Копируем строчку после «token:». Теперь запустим прокси между локальным узлом и сервером Kubernetes API:

```
[user@master ~]$ kubectl proxy
Starting to serve on 127.0.0.1:8001
```

После этого можно зайти браузером на адрес http://localhost:8001/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy/. Веб-панель будет доступна только на узле, на котором запущена команда kubectl proxy.

Установка кластера в публичном облаке Microsoft Azure

Рассмотрим еще один вариант установки и использования Kubernetes: на этот раз в публичном облаке. В качестве примера используем сервис Azure Kubernetes Service (AKS) [6], работающий в облаке Microsoft Azure. Отличительной особенностью такого использования Kubernetes является то, что вы получаете кластер как сервис. За процедуру установки и обслуживания кластеров отвечает Microsoft. Плюсом для вас является то, что вы не платите за сам сервис AKS, а только за потребляемые вычислительные ресурсы. Мы не будем касаться основ работы с Azure, поскольку это выходит за рамки данной книги. Предполагается, что вы обладаете учетной записью в облаке Microsoft Azure, знаете, как работать с порталом, и т. д.

Подключившись к порталу Azure, нажимаем в левом верхнем углу «Create a resource» → «Containers» → «Kubernetes Service» (рис. 5.3).

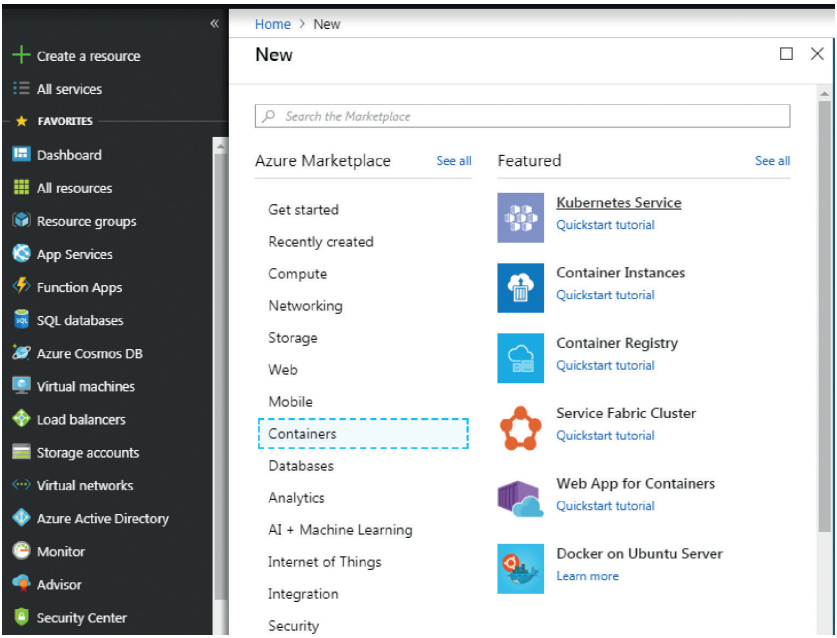


Рис. 5.3 ❖ Начало создания кластера AKS на портале Azure

В появившемся окне выбираем подписку, в которой будет создан кластер (AKSTest), существующую ресурсную группу или имя новой (в примере также AKSTest), имя кластера (akstest01), регион Azure (West Europe), версию Kubernetes (1.11.2) и DNS-префикс, который будет добавлен к полностью квалифицированному имени хоста, по которому будет доступен API кластера. Наконец, выбираем размер (Standard DS2 v2) и число вычислительных узлов кластера (3). Пример экрана приведен на рис. 5.4.

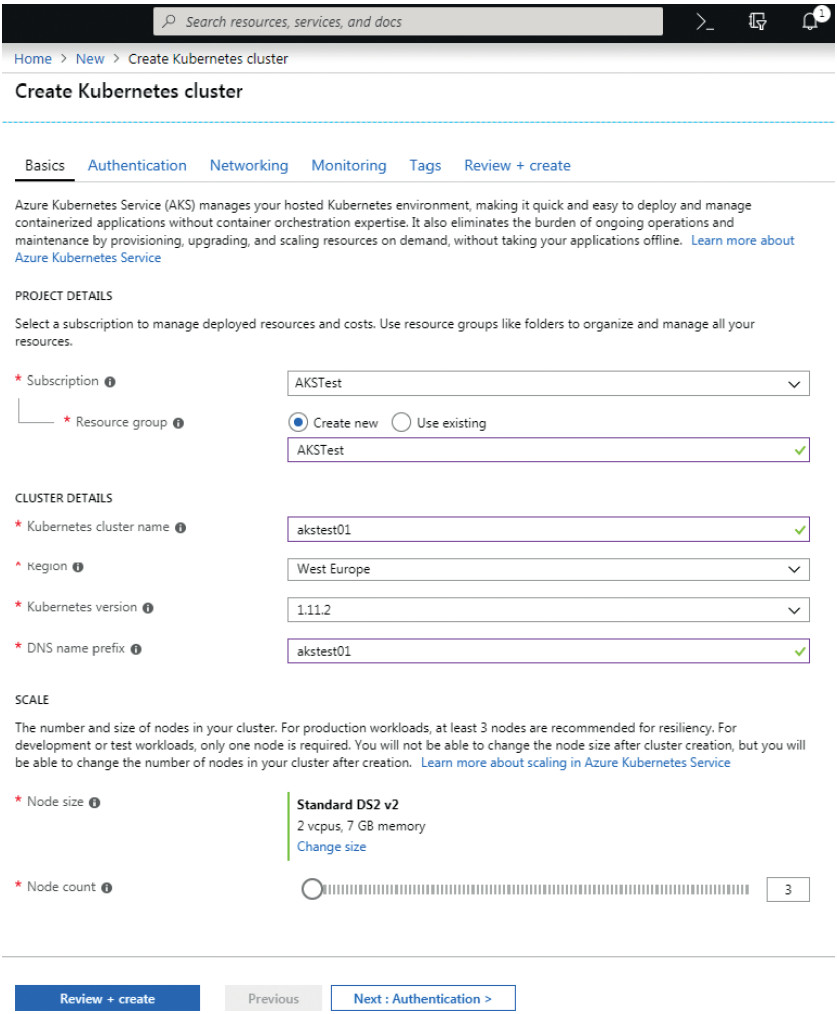


Рис. 5.4 ❖ Задание основных параметров кластера AKS

Далее нажимаем кнопку **Next: Authentication**, где можно создать новую субъект-службу или ввести параметры существующей. Также можно выбрать, включен или нет механизм RBAC (ролевого доступа) в кластере Kubernetes. На этом все обязательные параметры введены, и можно пропустить остальные шаги, просто

нажав **Review+create**. Более подробно про дополнительные параметры при создании кластера можно прочесть на сайте Microsoft [7]. Спустя какое-то время вы получите сообщение о том, что кластер создан. После этого можно подключиться к нему и работать, как с локальным. Для того чтобы это сделать, необходимо несколько дополнительных шагов.

В системе, с которой вы планируете работать с кластером, необходимо получить реквизиты подключения к кластеру, которые будут сохранены в локальный конфигурационный файл. Подробнее о конфигурационном файле мы поговорим в следующей главе, а сейчас вам необходимо подключиться к своей учетной записи при помощи Azure CLI [8].

Подключаемся к подписке Azure при помощи команды az login:

```
[aksuser@centos7 ~]$ az login
To sign in, use a web browser to open the
page https://microsoft.com/devicelogin
and enter the code A7VT0JMH3 to authenticate.
[
  {
    "cloudName": "AzureCloud",
    "id": "0b481e...",
    "isDefault": true,
    "name": "AKSTest",
    "state": "Enabled",
    "tenantId": "3dd3f8...",
    "user": {
      "name": "amarkelov@yandex.ru",
      "type": "user"
    }
]
```

Далее скачиваем конфигурационный файл с реквизитами кластера при помощи команды az aks get-credentials, в которой указываем ресурсную группу и имя кластера:

```
[aksuser@centos7 ~]$ az aks get-credentials --resource-group AKSTest --name
akstest01
Merged "akstest01" as current context in /home/aksuser/.kube/config
```

Проверяем доступность кластера:

```
[aksuser@centos7 ~]$ kubectl get nodes
NAME                       STATUS    ROLES     AGE       VERSION
aks-agentpool-20512754-0   Ready     agent     11m       v1.11.2
aks-agentpool-20512754-1   Ready     agent     10m       v1.11.2
aks-agentpool-20512754-2   Ready     agent     11m       v1.11.2
```

Теперь у нас все готово, для того чтобы начать работать с кластером в следующей главе.

Вопросы для самопроверки

1. **Какой из перечисленных компонентов Kubernetes опциональный?**

A. kube-scheduler
B. etcd
C. cloud-controller-manager
D. kube-scheduler

2. **Где хранятся настройки доступа к кластеру по умолчанию?**

A. k8s/conf
B. kube/configuration
C. kube/config

Список ссылок

1. https://www.cncf.io/
2. https://12factor.net/ru/
3. https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/
4. https://github.com/kelseyhightower/kubernetes-the-hard-way
5. https://github.com/Huawei-PaaS/CNI-Genie
6. https://azure.microsoft.com/en-us/services/kubernetes-service/
7. https://docs.microsoft.com/en-us/azure/aks/kubernetes-walkthrough-portal
8. https://docs.microsoft.com/en-us/cli/azure/?view=azure-cli-latest

Глава 6.
ОСНОВЫ РАБОТЫ С KUBERNETES

Основные объекты Kubernetes

Прежде чем переходить к практическому введению в Kubernetes, познакомимся с основными объектами системы. Все объекты можно создать как из командной строки, так и из описания сохраненного в файле формата YAML или JSON.

Пространство имен (namespace, ns) – область видимости для ресурсов и объектов в кластере. Не следует путать пространство имен Kubernetes с пространством имен GNU/Linux. Некоторые ресурсы являются ресурсами уровня кластера, и к ним пространства имен не применимы. Примеры таких ресурсов: Node, PersistentVolume или ClusterRole. Можно рассматривать пространство имен как аналог тенанта или проекта в OpenStack.

Pod-модуль (pod, po) – это группа, состоящая из одного или более контейнеров, которые используют общее пространство имен GNU/Linux UTS, Network и IPC и тома. В последних версиях они также могут использовать общее пространство PID. Pod-модуль можно рассматривать как «логический сервер». Это базовый тип нагрузки в Kubernetes. Все контейнеры модуля запускаются на одном узле. Все pod-модули объединяются одной плоской сетью без NAT.

Метка (label) – пара ключ/значение, которая может быть присвоена любому ресурсу. Селектор (selector) использует метки для фильтрации ресурсов и других операций.

Аннотации (annotations) – позволяют ассоциировать с ресурсами произвольные метаданные, менять свойства или поведение объектов Kubernetes, а также настраивать их взаимодействие с другими сервисами Kubernetes. Аннотации, например, используются в OpenShift для расширения функционала Kubernetes.

Контроллер (controller) – это процесс в Kubernetes, который следит за состоянием ресурсов и, в случае если состояние отличается от заданного, приводит ресурсы к заданному.

Контроллер репликации (replication controller, rc) – базовый тип контроллера. Определяет, как pod-модули реплицируются по узлам кластера.

Набор реплик (replica set, rs) – более современная реализация контроллера, имеющего ту же задачу, что и контроллер репликации. Основное отличие – более гибкие правила селекторов. Далее мы познакомимся и с другими типами контроллеров.

Сервис (service, svc) – комбинация IP-адреса и порта, предоставляющая доступ к набору pod-модулей. По умолчанию сервис подключает клиентов к pod-модулям по очереди (по-английски данный термин звучит round-robin).

Словарь конфигурации ConfigMap (ConfigMap, cm) – набор ключей и значений, который может использоваться другими ресурсами. Как правило, ConfigMap используется для хранения и передачи параметров конфигурации.

Секрет (secret) – аналогичен ConfigMap, но содержит закодированные значения при помощи алгоритма Base64. Секреты по умолчанию не шифруются в etcd, но есть возможность настроить шифрование при установке кластера.

Постоянный том (persistent volume, pv) – постоянный том (хранилище данных), который можно подключить к pod-модулю посредством **запроса на постоянный том (persistent volume claim, pvc)**.

Далее мы рассмотрим и некоторые другие ресурсы. Получить список доступных ресурсов и их сокращения можно при помощи команды:

```
[user@master ~]$ kubectl api-resources
NAME                     SHORTNAMES  APIGROUP  NAMESPACED  KIND
bindings                                       true        Binding
componentstatuses        cs                    false       ComponentStatus
configmaps               cm                    true        ConfigMap
endpoints                ep                    true        Endpoints
events                   ev                    true        Event
limitranges              limits                true        LimitRange
namespaces               ns                    false       Namespace
nodes                    no                    false       Node
persistentvolumeclaims   pvc                   true
PersistentVolumeClaim
persistentvolumes        pv                    false       PersistentVolume
pods                     po                    true        Pod
...
```

В Kubernetes версии 1.13 команда выведет список из 53 ресурсов.

Создаем первый pod-модуль

Напомним, что pod-модуль – это группа, состоящая из одного или более контейнеров, которые используют общее пространство имен и тома. pod-модуль – базовый тип нагрузки в Kubernetes. Все контейнеры модуля запускаются на одном узле и имеют один IP-адрес. Рассмотрим рис. 6.1, на котором изображен pod-модуль, состоящий из двух контейнеров.

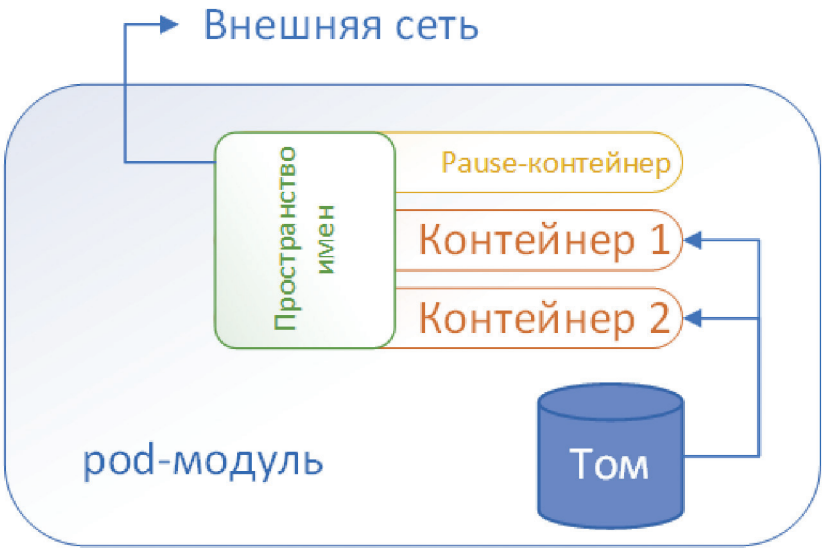


Рис. 6.1 ❖ pod-модуль в Kubernetes

На рисунке изображен модуль, состоящий из двух контейнеров плюс один инфраструктурный pause-контейнер. Инфраструктурный контейнер предназначен для тех же целей, что и в podman. Он инициализирует пространства имен GNU/Linux. Также к контейнерам смонтирован внешний том. Тома мы будем рассматривать далее. Для взаимодействия друг с другом контейнеры модуля могут использовать интерфейс короткой петли (loopback interface), писать файлы на общую файловую систему (в случае

использования общих томов) и использовать механизм IPC (inter-process communication).

Прежде чем создать наш первый pod-модуль, создадим выделенное для него пространство имен Kubernetes (не путать с пространством имен GNU/Linux!):

```
[user@master ~]$ kubectl create ns test1
namespace/test1 created
```

Просмотрим список всех пространств имен:

```
[user@master ~]$ kubectl get ns
NAME          STATUS   AGE
default       Active   78d
kube-public   Active   78d
kube-system   Active   78d
test1         Active   22s
```

Как видно, к трем созданным по умолчанию пространствам имен прибавилось наше новое test1. Перечислим пространства имен, которые уже были созданы во время установки кластера:

- **default** – если явно не задано иное, все объекты по умолчанию создаются в этом пространстве имен;
- **kube-public** – пространство имен, доступное всем без аутентификации. Общая информация о кластере обычно располагается тут;
- **kube-system** – пространство имен, предназначенное для инфраструктурных pod-модулей.

Как и для всех ресурсов Kubernetes, определение пространства имен можно сохранить в JSON- или YAML-файл:

```
[user@master ~]$ kubectl get ns test1 -o yaml
apiVersion: v1
kind: Namespace
metadata:
  creationTimestamp: "2019-03-25T19:32:54Z"
  name: test1
  resourceVersion: "45208"
  selfLink: /api/v1/namespaces/test1
  uid: c92d7436-4f34-11e9-9a7a-0800279bf286
spec:
  finalizers:
  - kubernetes
status:
  phase: Active
```

Создадим второе пространство имен через определение из файла yaml. В нем нам не нужна информация времени выполнения. Определение ресурса будет выглядеть так:

```
[user@master ~]$ cat test2.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: test2
```

Создаем пространство имен и проверяем его наличие:

```
[user@master ~]$ kubectl create -f test2.yaml
namespace/test2 created
[user@master ~]$ kubectl get ns
NAME          STATUS   AGE
default       Active   78d
kube-public   Active   78d
kube-system   Active   78d
test1         Active   12m
test2         Active   6s
```

Как мы уже сказали, по умолчанию объект создается в пространстве имен default. Для указания имени пространства имен в команде kubectl можно использовать опцию -n с требуемым именем или --all-namespaces для выбора ресурсов из всех пространств имен.

Создадим pod-модуль при помощи команды kubectl run, указав в качестве имени модуля и имени образа nginx:

```
[user@master ~]$ kubectl run nginx --image=nginx --port=80 -n test1
--generator=run-pod/v1
pod/ nginx created
```

Также мы использовали опцию --port, указав порт, который должен быть доступен для модуля, пространство имен test1 и опцию --generator=run-pod/v1, которая необходима для того, чтобы был создан только pod-модуль без каких-либо контроллеров, которые нам пока не нужны. Проверим, что модуль стартовал:

```
[user@master ~]$ kubectl get pod -n test1
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0          5s
```

Перенаправим порт 80 pod-модуля на локальный порт 10000, используя утилиту kubectl. Как правило, данная команда требуется во время отладки и разработки:

```
[user@master ~]$ kubectl port-forward nginx 10000:80 -n test1
Forwarding from 127.0.0.1:10000 -> 80
Forwarding from [::1]:10000 -> 80
```

Проверим доступность nginx на порту 10000:

```
[user@master ~]$ curl http://127.0.0.1:10000
Handling connection for 10000
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

Определение pod-модуля в yaml выглядит следующим образом:

```
[user@master ~]$ cat nginx.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: test2
spec:
  containers:
  - image: nginx
    name: nginx
    ports:
    - containerPort: 80
      protocol: TCP
```

Определение порта, как и в команде kubectl run, является информационным и не влияет на возможность подключиться к порту. Данная информация используется для создания сервиса, обеспечивающего подключение к модулю.

Создадим из файла ресурс и проверим, что модуль запущен. Обратите внимание, что теперь мы используем второе пространство имен:

```
[user@master ~]$ kubectl create -f nginx.yaml
pod/nginx created
[user@master ~]$ kubectl get pod -n test2
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0          16s
```

Мы говорили, что любому ресурсу можно присвоить метку. Продемонстрируем это на примере pod-модуля:

```
[user@master ~]$ kubectl label pod nginx status=dev -n test2
pod/nginx labeled
```

Просмотреть метки можно при помощи ключа --show-labels:

```
[user@master ~]$ kubectl get pod --show-labels -n test2
NAME    READY   STATUS    RESTARTS   AGE   LABELS
nginx   1/1     Running   0          5m    status=dev
```

Просмотреть журнал контейнера можно при помощи команды

```
[user@master ~]$ kubectl describe pod -n test2
```

Под конец удалим pod-модуль и пространство имен:

```
[user@master ~]$ kubectl delete pod nginx -n test2
pod "nginx" deleted
[user@master ~]$ kubectl delete ns test2
namespace "test2" deleted
```

На самом деле первая команда была не нужна, поскольку при удалении пространства имен удаляются и все объекты этого пространства.

Важно отметить, что мы рассмотрели не все возможности pod-модулей. Для промышленной эксплуатации необходимо ознакомиться с такими понятиями и внедрить в конфигурацию пробы «самочувствия» модулей Liveness и Readiness probes, а также понятия лимитов и запросов модулей на ресурсы и качество сервиса (resources requests, limits, Pod Quality of Service).

В следующем разделе мы научимся управлять pod-модулями при помощи внедрений.

Внедрения (Deployments)

Мы научились запускать отдельные pod-модули, но на практике обычно pod-модули запускаются и масштабируются при помощи так называемых контроллеров (controllers). Контроллер может создавать и управлять множеством pod-модулей, осуществляя их развертывание и репликацию. Контроллеры следят за тем, чтобы

поддерживалось заданное число pod-модулей. Например, если узел, на котором работал модуль, выходит из строя, контроллер перезапустит копии пострадавших pod-модулей на одном из работающих узлов. Все контроллеры в конечном итоге представляются в виде pod-модулей. Взаимосвязь между основными типами контроллеров и pod-модулями приведена на рис. 6.2.

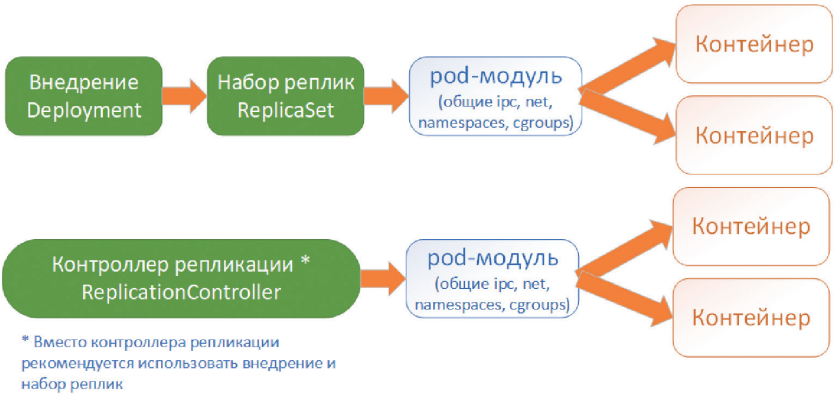


Рис. 6.2 ❖ Взаимосвязь между контроллерами и pod-модулями

На рисунке приведены три типа контроллеров. По умолчанию рекомендуется использовать внедрение (Deployment). Задачей внедрения является убедиться, что все необходимые для pod-модулей ресурсы доступны. Примерами ресурсов могут быть IP-адрес или постоянное хранилище. После этого внедрение развертывает набор реплик (ReplicaSet). Набор реплик контролирует развертывание и перезапуск pod-модулей. На рисунке также приведен контроллер репликации (ReplicationController). Исторически это был первый тип контроллера для запуска рабочей нагрузки в Kubernetes. В настоящее время вместо него рекомендуется использование внедрения и набора реплик.

Познакомимся с внедрением на практике. Создадим новое пространство имен:

```
[user@master ~]$ kubectl create ns new-deploy
```

В новом пространстве имен new-deploy создадим внедрение командой kubectl create deployment. В качестве образа укажем nginx:

```
[user@master ~]$ kubectl create deployment nginx-deploy --image=nginx -n new-
deploy
deployment.apps/nginx-deploy created
```

Проверим, что были созданы одно новое внедрение, набор реплик и один pod-модуль:

```
[user@master ~]$ kubectl get deployments,rs,pods -n new-deploy
NAME                                 DESIRED   CURRENT   UP-TO-DATE
AVAILABLE   AGE
deployment.extensions/nginx-deploy   1         1         1            1
11m

NAME                                            DESIRED   CURRENT   READY
AGE
replicaset.extensions/nginx-deploy-6d78f8cf68   1         1         1
11m

NAME                                READY   STATUS    RESTARTS   AGE
pod/nginx-deploy-6d78f8cf68-k5t4w   1/1     Running   0          11m
```

Разбор шаблона внедрения

Выведем на экран терминала описание созданных ресурсов в формате YAML и пройдемся по основным его частям:

```
[user@master ~]$ kubectl get deployments,rs -o yaml -n new-deploy | nl
...
```

Полный листинг приведен в приложении 1, а здесь мы рассмотрим его по частям. Начнем с внедрения.

```
1  apiVersion: v1
2  items:
3  - apiVersion: extensions/v1beta1
4    kind: Deployment
```

В первой строке приведена версия v1. Это не версия объекта Deployment. Это версия списка. Далее в строке четыре идет тип первого объекта в списке – Deployment, а в третьей строке версия объекта Deployment. Хотя, как вы видите, данный объект рассматривается как бета-версия, тем не менее он рекомендован к применению.

```
5    metadata:
6      annotations:
7        deployment.kubernetes.io/revision: "1"
8      creationTimestamp: "2019-04-07T10:06:29Z"
9      generation: 1
```

```
    10      labels:
    11        app: nginx-deploy
    12      name: nginx-deploy
    13      namespace: new-deploy
    14      resourceVersion: "4376"
    15      selfLink: /apis/extensions/v1beta1/namespaces/new-deploy/
deployments/nginx-deploy
    16      uid: cffc5bbe-591c-11e9-8edc-628057a8c709
```

С пятой по шестнадцатую строки представлены метаданные внедрения. В метаданных задаются аннотации, метки и часть информации, которая подставляется автоматически, например время создания внедрения (строка восемь). Часть настроек, таких как правила соразмещения pod-модулей, не отображается, так как не задана.

В строках шесть и семь заданы аннотации. Они напрямую не используются Kubernetes, однако могут использоваться в надстройках над ним. Например, OpenShift, который мы рассмотрим в главе 8, активно использует аннотации. В девятой строке указано «поколение» объекта внедрения. Номер увеличивается каждый раз, когда объект редактируется, например меняется число реплик. В строках десять и одиннадцать заданы метки, используемые для выбора или исключения объектов в операциях. Можно попробовать выбрать все pod-модули с указанной меткой:

```
[user@master ~]$ kubectl get pods -l app=nginx-deploy --all-namespaces
NAMESPACE    NAME                              READY   STATUS    RESTARTS   AGE
new-deploy   nginx-deploy-6d78f8cf68-hpq59   1/1     Running   0          18m
```

Можно вывести все pod-модули и их метки:

```
[user@master ~]$ kubectl get pods --show-labels --all-namespaces
NAMESPACE     NAME                                    READY   STATUS
RESTARTS   AGE   LABELS
...
new-deploy    nginx-deploy-6d78f8cf68-hpq59           1/1     Running   0
20m   app=nginx-deploy,pod-template-hash=2834947924
...
```

В двенадцатой строке задано имя внедрения. Это обязательный параметр. Имя должно быть уникально в пределах одного пространства имен. А в следующей строке указано пространство имен, в котором созданы объекты внедрения. Параметр resourceVersion в строке четырнадцать связан с целостностью базы данных

etcd. Изменения в базе данных повлекут изменение этого числа. SelfLink в строке пятнадцать указывает на то, как объект представляется в вызовах API. Параметр uid задает уникальный идентификатор объекта.

```
17    spec:
18      progressDeadlineSeconds: 600
19      replicas: 1
20      revisionHistoryLimit: 10
21      selector:
22        matchLabels:
23          app: nginx-deploy
24      strategy:
25        rollingUpdate:
26          maxSurge: 25%
27          maxUnavailable: 25%
28        type: RollingUpdate
```

Первый блок spec в описании внедрения относится к тому, как будет создаваться «дочерний» объект ReplicaSet. Параметр progressDeadlineSeconds в строке восемнадцать задает тайм-аут в секундах до того, как будет выдана ошибка во время изменения внедрения. Причиной ошибки могут быть, например, квоты или недоступный образ. В девятнадцатой строке указано количество реплик pod-модуля. Если изменить это число при помощи команды kubectl edit, то соответственно увеличится количество реплик. В строке под номером двадцать задано, сколько будет храниться старых версий спецификаций ReplicaSet. Они необходимы для отката на предыдущие версии внедрения. В строках с двадцать первой по двадцать третью задан селектор на основе совпадения метки app значению nginx-deploy. Контроллер будет следить, чтобы в кластере присутствовало заданное число pod-модулей, и их идентификация будет выполняться по данному селектору. Строки с двадцать четвертой по двадцать восьмую определяют стратегию и тип обновления. По умолчанию стратегия – RollingUpdate. При ней задано, что одновременно может создаваться 25 % новых pod-модулей на замену старым, и максимальное число модулей в состоянии, отличном от Ready, – 25 % от общего числа модулей.

```
29      template:
30        metadata:
31          creationTimestamp: null
32          labels:
33            app: nginx-deploy
```

```
34        spec:
35          containers:
36          - image: nginx
37            imagePullPolicy: Always
38            name: nginx
39            resources: {}
40            terminationMessagePath: /dev/termination-log
41            terminationMessagePolicy: File
42          dnsPolicy: ClusterFirst
43          restartPolicy: Always
44          schedulerName: default-scheduler
45          securityContext: {}
46          terminationGracePeriodSeconds: 30
```

Следующий блок представляет собой шаблон описания pod-модулей внедрения. Выделим несколько строк, которые необходимо прокомментировать. Параметр imagePullPolicy в строке тридцать семь определяет, когда необходимо скачивать образ из репозитория, а когда можно использовать локальный кеш. В строках сорок и сорок один задано, куда и в какой форме выдается информация о статусе контейнера. Параметр dnsPolicy в строке сорок два говорит, что в первую очередь необходимо делать запрос к dns-серверу кластера, а только потом по настройкам dns-сервера с узла. RestartPolicy говорит о том, что в случае если pod-модуль «умер», его необходимо рестартовать. Параметр schedulerName в строке номер сорок четыре позволяет задать нестандартный планировщик модулей. Через параметр securityContext можно передавать параметры безопасности модуля, например настройки SELinux или AppArmor. TerminationGracePeriodSeconds в последней строке блока задает, что необходимо ждать тридцать секунд между отправлением сигнала SIGKILL и SIGTERM в случае остановки pod-модуля.

```
47    status:
48      availableReplicas: 1
49      conditions:
50      - lastTransitionTime: "2019-04-07T10:06:40Z"
51        lastUpdateTime: "2019-04-07T10:06:40Z"
52        message: Deployment has minimum availability.
53        reason: MinimumReplicasAvailable
54        status: "True"
55        type: Available
56      - lastTransitionTime: "2019-04-07T10:06:29Z"
```

```
    57        lastUpdateTime: "2019-04-07T10:06:40Z"
    58        message: ReplicaSet "nginx-deploy-6d78f8cf68" has successfully
progressed.
    59        reason: NewReplicaSetAvailable
    60        status: "True"
    61        type: Progressing
    62      observedGeneration: 1
    63      readyReplicas: 1
    64      replicas: 1
    65      updatedReplicas: 1
```

Блок со строки сорок семь по шестьдесят пять показывает информацию времени выполнения о статусе внедрения.

Далее показан вывод команды kubectl describe, описывающей внедрение:

```
[user@master ~]$ kubectl describe deployments -n new-deploy
Name:                   nginx-deploy
Namespace:              new-deploy
CreationTimestamp:      Sun, 07 Apr 2019 12:06:29 +0200
Labels:                 app=nginx-deploy
Annotations:            deployment.kubernetes.io/revision: 1
Selector:               app=nginx-deploy
Replicas:               1 desired | 1 updated | 1 total | 1 available | 0
unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=nginx-deploy
  Containers:
   nginx:
    Image:        nginx
    Port:         <none>
    Host Port:    <none>
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
  Progressing    True    NewReplicaSetAvailable
OldReplicaSets:  <none>
NewReplicaSet:   nginx-deploy-6d78f8cf68 (1/1 replicas created)
Events:          <none>
```

Масштабирование и откат внедрений

В настоящий момент у нас одна реплика. Проверим это еще раз:

```
[user@master ~]$ kubectl get deployments -n new-deploy
NAME           DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deploy   1         1         1            1           4h
```

При помощи команды kubectl scale мы можем увеличить или уменьшить число реплик:

```
[user@master ~]$ kubectl scale deploy/nginx-deploy --replicas=3 -n new-deploy
deployment.extensions/nginx-deploy scaled
```

Через несколько секунд kubectl get deployments покажет нам изменения:

```
[user@master ~]$ kubectl get deployments -n new-deploy
NAME           DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deploy   3         3         3            3           4h
```

Следующее, что мы попробуем сделать, – это изменим образ, при помощи которого было выполнено наше внедрение. Убедимся, что все три pod-модуля в настоящий момент используют образ nginx:latest:

```
[user@master ~]$ kubectl get po -o yaml -n new-deploy | grep image
      image: nginx
      imagePullPolicy: Always
      image: nginx:latest
      imageID: docker-pullable://nginx@
sha256:c8a861b8a1eeef6d48955a6c6d5dff8e2580f13ff4d0f549e082e7c82a8617a2
...
```

Теперь отредактируем внедрение при помощи команды kubectl edit. Необходимо изменить выделенную строку, добавив в конце имени образа тег, указывающий на одну из старых версий образа nginx:

```
[user@master ~]$ kubectl edit deployments nginx-deploy -n new-deploy
deployment.extensions/nginx-deploy edited
...
    spec:
      containers:
      - image: nginx:1.10
        imagePullPolicy: Always
        name: nginx
        resources: {}
        terminationMessagePath: /dev/termination-log
```

```
        terminationMessagePolicy: File
...
```

Проверим, что начали создаваться новые pod-модули и удаляться старые:

```
[user@master ~]$ kubectl get po -n new-deploy
NAME                              READY   STATUS              RESTARTS   AGE
nginx-deploy-5f78fd994b-dplm9     1/1     Running             0          25s
nginx-deploy-5f78fd994b-whg9s     0/1     ContainerCreating   0          11s
nginx-deploy-6d78f8cf68-k5t4w     1/1     Running             0          4h
nginx-deploy-6d78f8cf68-plnqr     1/1     Running             0          10m
```

Наконец, проверим, что имя образа и его хеш изменились для всех pod-модулей. Их в конечном итоге опять должно стать три:

```
[user@master ~]$ kubectl get po -o yaml -n new-deploy | grep image
      image: nginx:1.10
      imagePullPolicy: Always
      image: nginx:1.10
      imageID: docker-pullable://nginx@
sha256:6202beb06ea61f44179e02ca965e8e13b961d12640101fca213efbfd145d7575
...
```

Теперь попробуем еще один функционал внедрения. Допустим, по какой-то причине нам требуется откатиться к предыдущей версии приложения. Это можно сделать при помощи команды kubectl rollout. Для начала посмотрим историю:

```
[user@master ~]$ kubectl rollout history deployment/nginx-deploy -n new-
deploy
deployment.extensions/nginx-deploy
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
```

На настоящий момент имеются две ревизии внедрения. Откатимся к первой:

```
[user@master ~]$ kubectl rollout undo deployment/nginx-deploy --to-revision=1
-n new-deploy
deployment.extensions/nginx-deploy rolled back
```

Подождем, пока все три pod-модуля пересоздадутся, и убедимся, что имя и хеш образа вернулись в оригинальное состояние:

```
[user@master ~]$ kubectl get po -o yaml -n new-deploy | grep image
      image: nginx
      imagePullPolicy: Always
      image: nginx:latest
```

```
    imageID: docker-pullable://nginx@
sha256:c8a861b8a1eeef6d48955a6c6d5dff8e2580f13ff4d0f549e082e7c82a8617a2
```

Доступ к pod-модулю извне кластера

Для предоставления доступа к pod-модулям по сети – как друг к другу, так и снаружи кластера – используется концепция сервиса. Сервисы созданы с учетом того, что любой pod-модуль в любой момент может быть уничтожен и создан заново. Соответственно, IP-адрес тоже может поменяться. Сервис же предоставляет общий IP-адрес и порт для всех pod-модулей. Обычно определение того, какой pod-модуль необходимо включать в тот или иной сервис, задается при помощи селектора по общей метке. Конечный ресурс предоставляется посредством объекта Endpoint (конечная точка). Существует несколько типов сервисов, и они могут не только открывать внешний доступ или доступ внутри кластера, но и ссылаться на внешние ресурсы. Например, на базу данных, развернутую вне кластера Kubernetes.

Компонент **kube-proxy**, который запускается на каждом узле, отслеживает создание новых сервисов и конечных точек при помощи **kube-apiserver**. Он открывает случайный порт на узле и перенаправляет трафик на заданный порт pod-модуля.

Kube-proxy реализовал набор pod-модулей, запускаемых по одному на каждом узле при помощи контроллера DaemonSet. Этот тип контроллера мы рассмотрим далее:

```
[user@master ~]$ kubectl get pods  -n kube-system -o wide
NAME                                    READY   STATUS    RESTARTS   AGE
IP            NODE                NOMINATED NODE   READINESS GATES
...
kube-proxy-g557g                        1/1     Running   18         91d
10.0.3.4      master.test.local   <none>           <none>
kube-proxy-nzkmn                        1/1     Running   13         91d
10.0.3.6      node2.test.local    <none>           <none>
kube-proxy-pfk6t                        1/1     Running   13         91d
10.0.3.5      node1.test.local    <none>           <none>
```

Создадим сервис типа ClusterIP (при таком типе сервиса pod-модули доступны только из сети кластера) при помощи команды kubectl expose:

```
[user@master ~]$ kubectl expose deployment/nginx-deploy --port=80
--type=ClusterIP -n new-deploy
service/nginx-deploy exposed
```

Проверим, какой IP-адрес был присвоен сервису:

```
[user@master ~]$ kubectl get svc -n new-deploy -o wide
NAME           TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
SELECTOR
nginx-deploy   ClusterIP   10.107.173.45   <none>        80/TCP    11m
app=nginx-deploy
```

Теперь можно убедиться, что сервис перенаправляет запросы к pod-модулям внедрения:

```
[user@master ~]$ curl 10.107.173.45:80
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
...
```

Посмотрим описание сервиса в формате YAML:

```
[user@master ~]$ kubectl get svc nginx-deploy -n new-deploy -o yaml
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: "2019-04-08T08:06:28Z"
  labels:
    app: nginx-deploy
  name: nginx-deploy
  namespace: new-deploy
  resourceVersion: "68158"
  selfLink: /api/v1/namespaces/new-deploy/services/nginx-deploy
  uid: 3632e788-59d5-11e9-9532-0800279bf286
spec:
  clusterIP: 10.107.173.45
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: nginx-deploy
  sessionAffinity: None
  type: ClusterIP
status:
  loadBalancer: {}
```

В данном выводе выделен селектор, который по метке определяет, на какие pod-модули сервис будет отправлять запросы. Проверим, что метки совпадают:

```
[user@master ~]$ kubectl get pods -n new-deploy --show-labels -o wide
NAME                            READY   STATUS    RESTARTS   AGE   IP
NODE                NOMINATED NODE   READINESS GATES   LABELS
nginx-deploy-66f67d8697-lfqgr   1/1     Running   0          76m
10.244.0.67    master.test.local   <none>           <none>
app=nginx-deploy,pod-template-hash=66f67d8697
nginx-deploy-66f67d8697-wnrq4   1/1     Running   0          78m
10.244.2.108   node2.test.local    <none>           <none>
app=nginx-deploy,pod-template-hash=66f67d8697
nginx-deploy-66f67d8697-x9xgs   1/1     Running   0          76m
10.244.1.83    node1.test.local    <none>           <none>
app=nginx-deploy,pod-template-hash=66f67d8697
```

При помощи команды kubectl describe svc можно также увидеть IP-адреса и порты pod-модулей, перечисленные выше:

```
[user@master ~]$ kubectl describe svc nginx-deploy -n new-deploy
Name:              nginx-deploy
Namespace:         new-deploy
Labels:            app=nginx-deploy
Annotations:       <none>
Selector:          app=nginx-deploy
Type:              ClusterIP
IP:                10.107.173.45
Port:              <unset>  80/TCP
TargetPort:        80/TCP
Endpoints:         10.244.0.67:80,10.244.1.83:80,10.244.2.108:80
Session Affinity:  None
Events:            <none>
```

Рисунок 6.3 упрощенно демонстрирует связи описанных компонентов.

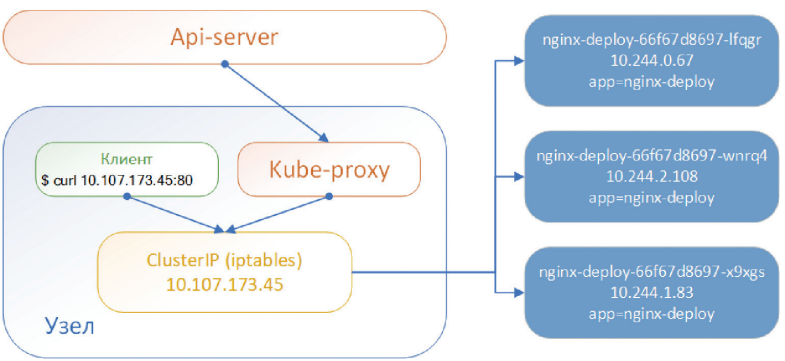


Рис. 6.3 ❖ Упрощенная взаимосвязь между pod-модулями и сервисом

Нужно сказать, что не все подключаемые модули CNI по умолчанию предоставляют доступ с узлов в сеть кластера. Можно также упомянуть тип сервиса NodePort. Этот тип аналогичен стандартному сервису с типом ClusterIP, но сервис при этом доступен не только через внутренний IP сети кластера, но и через IP узла.

Покажем, как можно дать доступ к нашим pod-модулям извне сети кластера. Для этого мы используем кластер, развернутый в публичном облаке, который интегрирован с балансировщиком нагрузки облачного провайдера. Настройка подобной конфигурации на локальных виртуальных машинах выходит за рамки рассмотрения данной книги.

Создадим точно такой же сервис, но в качестве типа укажем LoadBalancer:

```
[user@aks ~]$ kubectl expose deployment/nginx-deploy --port=80
--type=LoadBalancer -n new-deploy
service/nginx-deploy exposed
```

Посмотрим на состояние сервиса. Некоторое время в столбце, где указывается внешний IP-адрес, будет указано «pending»:

```
[user@aks ~]$ kubectl get svc -n new-deploy
NAME           TYPE           CLUSTER-IP   EXTERNAL-IP   PORT(S)        AGE
nginx-deploy   LoadBalancer   10.0.3.156   <pending>     80:30899/TCP   53s
```

Однако спустя некоторое время мы увидем маршрутизируемый внешний IP-адрес, доступный из интернета:

```
[user@aks~]$ kubectl get svc -n new-deploy
NAME           TYPE           CLUSTER-IP   EXTERNAL-IP     PORT(S)        AGE
nginx-deploy   LoadBalancer   10.0.3.156   13.80.249.181   80:30899/TCP   4m
```

Существует еще один способ обеспечения доступа к службам извне кластера, обычно применяемый для HTTP/HTTPS, – Ingress. Основное преимущество Ingress, по сравнению с уже рассмотренными способами, – что для него достаточно одного IP-адреса и балансировщика нагрузки. Для обеспечения работы объектов Ingress требуется установленный контроллер Ingress. Если вы используете GCE/Google Kubernetes Engine, то он у вас уже имеется. В остальных случаях вам, возможно, придется его установить отдельно. На рис. 6.4 представлена логическая схема работы Ingress Controller.

Рис. 6.4 ❖ Логическая схема работы Ingress Controller

Существует множество реализаций Ingress Controller. Со списком можно ознакомиться по ссылке [1]. В одном кластере можно использовать несколько типов Ingress Controller.

На этом мы закончим рассмотрение способов доступа к pod-модулям извне кластера.

Постоянные тома и запросы постоянных томов

Спецификация pod-модуля может включать в себя один или более томов (volume) для хранения файлов и доступа к ним из нескольких контейнеров одного pod-модуля или из разных модулей. В последнем случае сам Kubernetes не отслеживает целостности данных, и блокировки должны быть реализованы внешними методами. Срок жизни тома привязан к сроку жизни pod-модуля, не контейнера. Таким образом, контейнеры могут перезапускаться, но данные на томе при этом сохраняются. В настоящий момент в Kubernetes существует порядка тридцати различных типов томов, например azureDisk, cephfs, cinder, Fibre Channel (fc), glusterfs, iscsi, nfs, scaleIO. В качестве томов к pod-модулю можно подключать такие объекты, как Secret и ConfigMap. Они будут рассмотрены далее.

Как видите, большинство типов томов являются специфическими для конкретных технологий хранения данных. Однако имеются и универсальные типы:

- hostPath – монтирование директорий узла, на котором выполняется контейнер. Данные в таком томе сохранятся и после удаления pod-модуля. Также данные можно переиспользовать из другого pod-модуля, но только если он будет назначен на тот же узел;

- emptyDir – пустая директория для хранения временных данных. Данный тип применяется, например, когда необходимо обменяться данными между контейнерами внутри одного модуля. В случае удаления pod-модуля данные на таком томе теряются;
- PersistentVolumeClaim для запроса постоянного тома persistentVolume.

Поговорим подробнее про постоянный том (persistent volume, pv). Он абстрагирует конкретную реализацию хранилища и позволяет создавать тома, не привязанные к сроку жизни pod-модуля. Отсюда и «постоянный» в названии. Постоянные тома создаются заранее администратором или динамически во время запроса на постоянный том (PersistentVolumeClaim, pvc). Запрос определяется в описании pod-модуля. Модуль задает требуемый размер, тип доступа и, при необходимости, тип хранилища (через параметр StorageClass). Типичное использование постоянного тома – хранение файлов базы данных.

Таким образом, постоянное хранилище проходит через следующие фазы своей жизни:

- создание постоянного тома. Этот процесс может быть произведен заранее администратором кластера или динамически, например если кластер развернут в облачном провайдере;
- привязка постоянного тома к pod-модулю при помощи pvc. Кластер стремится удовлетворить минимальные требования по объему, но может выделить и том большего размера. Например, если имеется pvs на 3 Гб, а в наличии имеются тома размерами в 1 Гб, 2 Гб и 5 Гб, будет выделен последний из перечисленных томов;
- использование тома до тех пор, пока он необходим;
- том освобождается, когда удаляется pvc.

Что далее происходит с освобожденным томом, зависит от политики persistentVolumeReclaimPolicy физического тома. Имеется три опции:

- Retain – данные остаются, и администратор кластера сам дальше решает, что делать с томом и данными;
- Delete – автоматически удаляется и API-объект pv, и сам том;
- Recycle – выполняется команда rm -rf /<точка_монтирования>, после чего том доступен для новых запросов.

Тома поддерживают три типа доступа:

- ReadWriteOnce (RWO) – том может быть смонтирован для чтения и записи с одного узла;
- ReadOnlyMany (ROX) – том может быть смонтирован с нескольких узлов только для чтения;
- ReadWriteMany (RWX) – том может быть смонтирован для чтения и записи с нескольких узлов.

Попробуем поработать с постоянными томами в нашем тестовом кластере на практике. Проще и быстрее всего будет создать том с использованием NFS-сервера. Для этого нам понадобится установить пакет nfs-utils на все три узла кластера:

```
[root@master ~]# yum -y install nfs-utils
[root@node1 ~]# yum -y install nfs-utils
[root@node2 ~]# yum -y install nfs-utils
```

Далее, на управляющем узле создадим конфигурационный файл сервера (обратите внимание, что между знаком «звездочки» и открывающей скобкой не должно быть пробела):

```
[root@master ~]# cat /etc/exports
/exportdata *(rw,sync,no_root_squash,subtree_check)
```

А затем саму директорию /exportdata, которая будет доступна клиентам nfs:

```
[root@master ~]# mkdir /exportdata
[root@master ~]# chmod 1777 /exportdata
```

Запустим и включим сервис:

```
[root@master ~]# systemctl start nfs.service
[root@master ~]# systemctl enable nfs.service
Created symlink from /etc/systemd/system/multi-user.target.wants/nfs-server.
service to /usr/lib/systemd/system/nfs-server.service.
```

После чего с одного из узлов проверим доступность экспортированной директории:

```
[root@node1 ~]# showmount -e master
Export list for master:
/exportdata *
```

Теперь создадим YAML-файл с описанием постоянного тома размером 1 Гб, который будет указывать на путь /exportdata (строка 12) на сервере master (строка 13). Тип доступа укажем Read-

WriteMany. Как мы помним, это означает, что том может быть смонтирован для чтения и записи с нескольких узлов:

```
[user@master ~]$ cat pv.yaml | nl
     1  apiVersion: v1
     2  kind: PersistentVolume
     3  metadata:
     4    name: pvnfs1
     5  spec:
     6    capacity:
     7      storage: 1Gi
     8    accessModes:
     9      - ReadWriteMany
    10    persistentVolumeReclaimPolicy: Retain
    11    nfs:
    12      path: /exportdata
    13      server: master
    14      readOnly: false
```

При помощи команды kubectl create создаем объект из описания в файле:

```
[user@master ~]$ kubectl create -f pv.yaml
persistentvolume/pvnfs1 created
```

Проверяем, что pv создан:

```
[user@master ~]$ kubectl get pv
NAME    CAPACITY  ACCESS MODES  RECLAIM POLICY  STATUS     CLAIM
STORAGECLASS  REASON  AGE
pvnfs1  1Gi       RWX           Retain          Available
24s
```

Теперь наступает очередь pvc:

```
[user@master ~]$ cat pvc.yaml | nl
     1  apiVersion: v1
     2  kind: PersistentVolumeClaim
     3  metadata:
     4    name: pvc1
     5  spec:
     6    accessModes:
     7    - ReadWriteMany
     8    resources:
     9      requests:
    10        storage: 500Mi
```

В запросе мы указали тип доступа, совпадающий с типом доступа тома, а также объем. Для того чтобы том был выделен на запрос,

декларированный размер тома должен быть больше или равен размеру в запросе. Также обратите внимание, что мы не заботились о том, чтобы на нашем NFS-сервере действительно был требуемый свободный объем пространства. Kubernetes не проверяет фактически присутствующий на диске объем свободного пространства. Об этом необходимо заботиться администратору кластера. Создаем pvc и проверяем, что он привязывается к нашему тому pvnfs1:

```
[user@master ~]$ kubectl create -f pvc.yaml -n new-deploy
persistentvolumeclaim/pvc1 created
[user@master ~]$ kubectl get pvc -n new-deploy
NAME   STATUS   VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS   AGE
pvc1   Bound    pvnfs1   1Gi        RWX                           26s
```

Проверим, что том pvnfs1 привязан к pvc и в выводе kubectl get pv:

```
[user@master ~]$ kubectl get pv
NAME     CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM
STORAGECLASS   REASON   AGE
pvnfs1   1Gi        RWX            Retain           Bound    new-deploy/
pvc1                            2m
```

Теперь используем PVC в уже существующем внедрении nginx-deploy из предыдущего раздела. Для этого отредактируем объект deployment:

```
[user@master ~]$ kubectl edit deployment nginx-deploy -n new-deploy
deployment.extensions/nginx-deploy edited
```

Изменения выделены в тексте:

```
  labels:
    app: nginx-deploy
  name: nginx-deploy
  namespace: new-deploy
...
spec:
...
    spec:
      containers:
      - image: nginx
        imagePullPolicy: Always
        name: nginx
        resources: {}
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
        volumeMounts:
```

```
      - mountPath: /opt
        name: vol1
    dnsPolicy: ClusterFirst
    restartPolicy: Always
    schedulerName: default-scheduler
    securityContext: {}
    terminationGracePeriodSeconds: 30
    volumes:
    - name: vol1
      persistentVolumeClaim:
        claimName: pvc1
```

В качестве тома vol1 мы используем том, предоставляемый запросом pvc1, а в спецификации контейнера указываем, что том vol1 должен быть смонтирован в директорию /opt. Связь между ресурсами для наглядности приведена на рис. 6.5.

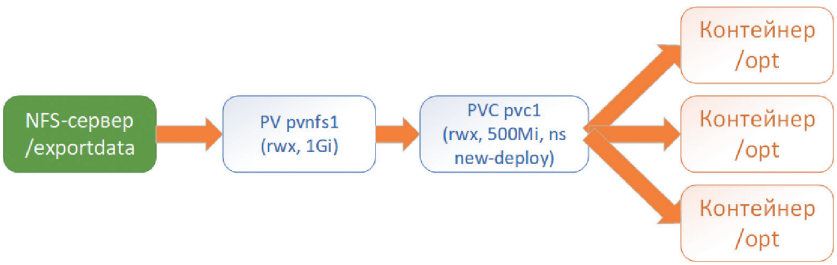


Рис. 6.5 ❖ Связь между ресурсами постоянного тома

Проверим, что pod-модули внедрения были пересозданы:

```
[user@master ~]$ kubectl get pod -n new-deploy
NAME                          READY   STATUS    RESTARTS   AGE
nginx-deploy-8d779b8c8-57zvv  1/1     Running   0          27s
nginx-deploy-8d779b8c8-knvpm  1/1     Running   0          32s
nginx-deploy-8d779b8c8-ltpsw  1/1     Running   0          17s
```

Посмотрим описание любого из модулей и убедимся, что том доступен через запрос:

```
[user@master ~]$ kubectl describe pod nginx-deploy-8d779b8c8-ltpsw -n new-
deploy
Name:               nginx-deploy-8d779b8c8-ltpsw
Namespace:          new-deploy
Priority:           0
PriorityClassName:  <none>
Node:               master.test.local/10.0.3.4
```

```
Start Time:         Wed, 10 Apr 201920:29:57 +0200
Labels:             app=nginx-deploy
                    pod-template-hash=8d779b8c8
Annotations:        <none>
Status:             Running
IP:                 10.244.0.82
Controlled By:      ReplicaSet/nginx-deploy-8d779b8c8
Containers:
  nginx:
    Container ID:
docker://77c7bac84ba14f67dce094bfedac9e361338e9d7fa9d67165830bd48343e3f45
...
    Mounts:
      /opt from vol1 (rw)
...
Volumes:
  vol1:
    Type:       PersistentVolumeClaim (a reference to a PersistentVolumeClaim
in the same namespace)
    ClaimName:  pvc1
    ReadOnly:   false
...
```

Проверим, на каких узлах запущены pod-модули:

```
[user@master ~]$ kubectl get pod -n new-deploy -o wide
NAME                           READY   STATUS    RESTARTS   AGE   IP
NODE                NOMINATED NODE   READINESS GATES
nginx-deploy-8d779b8c8-57zvv   1/1     Running   0          26m
10.244.2.156   node2.test.local    <none>           <none>
nginx-deploy-8d779b8c8-knvpm   1/1     Running   0          26m   10.244.1.94
node1.test.local    <none>           <none>
nginx-deploy-8d779b8c8-ltpsw   1/1     Running   0          26m   10.244.0.82
master.test.local   <none>           <none>
```

Если подключиться к любому из указанных рабочих узлов, то можно убедиться, что директория /exportdata смонтирована с узла master:

```
[root@node1 ~]# mount | grep nfs
master:/exportdata on /var/lib/kubelet/pods/9bc15eef-5bbe-11e9-85ad-
0800279bf286/volumes/kubernetes.io~nfs/pvnfs1 type nfs4 (rw,relatime,
vers=4.1,rsize=524288,wsize=524288,namlen=255,hard,proto=tcp,timeo=600,
retrans=2,sec=sys,clientaddr=10.0.3.5,local_lock=none,addr=10.0.3.4)
```

Обратите внимание на то, что том смонтирован с нескольких узлов, поскольку при создании мы указали тип доступа Read-WriteMany.

Словари конфигурации и секреты

Обычно настройки приложениям передаются при помощи таких методов, как конфигурационные файлы, переменные окружения, или при помощи параметров командной строки во время запуска приложения. В отличие от неконтейнерезированных приложений, не рекомендуется хранить файлы приложений (образ контейнера) и настройки вместе. Для применения настроек к контейнерам в Kubernetes можно воспользоваться двумя типами объектов: секретами (secrets) и картами конфигурации (configmap). Далее через описание pod-модуля можно передать настройки из этих объектов либо как переменные окружения, либо как файлы, смонтированные как тома. Отличие секретов от карт конфигурации заключается в том, что информация в первых кодируется при помощи алгоритма Base64.

Попробуем создать оба типа объектов из командной строки:

```
[user@master ~]$ kubectl create secret generic secret1 --from-literal
username=andrey --from-literal password=p@ssw0rd -n new-deploy
secret/secret1 created
[user@master ~]$ kubectl create configmap cmap1 --from-literal
username=andrey --from-literal password=p@ssw0rd -n new-deploy
configmap/cmap1 created
```

Если попросить описать оба объекта, то мы увидим, что в случае секрета значения ключей не выводятся:

```
[user@master ~]$ kubectl describe secrets secret1 -n new-deploy
Name:         secret1
Namespace:    new-deploy
Labels:       <none>
Annotations:  <none>

Type:  Opaque

Data
====
password:  8 bytes
username:  6 bytes
```

Для карты конфигурации значения приводятся в открытом виде:

```
[user@master ~]$ kubectl describe cm cmap1 -n new-deploy
Name:         cmap1
Namespace:    new-deploy
```

```
Labels:       <none>
Annotations:  <none>

Data
====
password:
----
p@ssw0rd
username:
----
andrey
Events:  <none>
```

Если посмотреть описание секрета в YAML, то мы увидим значения, но они будут закодированы:

```
[user@master ~]$ kubectl get secrets secret1 -n new-deploy -o yaml
apiVersion: v1
data:
  password: cEBzc3cwcmQ=
  username: YW5kcmV5
kind: Secret
metadata:
  creationTimestamp: "2019-04-11T11:34:19Z"
  name: secret1
  namespace: new-deploy
  resourceVersion: "105491"
  selfLink: /api/v1/namespaces/new-deploy/secrets/secret1
  uid: bf007148-5c4d-11e9-af76-0800279bf286
type: Opaque
```

Для раскодирования можно воспользоваться утилитой base64:

```
[user@master ~]$ echo cEBzc3cwcmQ= | base64 -d
p@ssw0rd
```

В случае карты конфигурации, опять же, все в открытом виде:

```
[user@master ~]$ kubectl get configmaps cmap1 -n new-deploy -o yaml
apiVersion: v1
data:
  password: p@ssw0rd
  username: andrey
kind: ConfigMap
metadata:
  creationTimestamp: "2019-04-11T11:34:30Z"
  name: cmap1
  namespace: new-deploy
  resourceVersion: "105507"
```

```
  selfLink: /api/v1/namespaces/new-deploy/configmaps/cmap1
  uid: c5858558-5c4d-11e9-af76-0800279bf286
```

Можно также создать карту или секрет из файла:

```
[user@master ~]$ kubectl create configmap passwdcm --from-file=/etc/passwd -n
new-deploy
configmap/passwdcm created
[user@master ~]$ kubectl describe configmap passwdcm -n new-deploy
Name:         passwdcm
Namespace:    new-deploy
Labels:       <none>
Annotations:  <none>

Data
====
passwd:
----
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
...
nfsnobody:x:65534:65534:Anonymous NFS User:/var/lib/nfs:/sbin/nologin

[user@master ~]$ kubectl create secret generic passwdsc --from-file=/etc/
passwd -n new-deploy
secret/passwdsc created
[user@master ~]$ kubectl describe secret passwdsc  -n new-deploy
Name:         passwdsc
Namespace:    new-deploy
Labels:       <none>
Annotations:  <none>

Type:  Opaque

Data
====
passwd:  1092 bytes
```

Теперь посмотрим, как сделать доступными секреты и карты конфигурации в нашем приложении. Начнем с представления карты как переменных окружения. Отредактируем уже имеющееся внедрение:

```
[user@master ~]$ kubectl edit deployment nginx-deploy -n new-deploy
deployment.extensions/nginx-deploy edited
```

Необходимые изменения приведены ниже:

```
...
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: nginx-deploy
    spec:
      containers:
      - env:
        - name: APPUSERNAME
          valueFrom:
            configMapKeyRef:
              key: username
              name: cmap1
        - name: APPPASSWORD
          valueFrom:
            configMapKeyRef:
              key: password
              name: cmap1
        image: nginx
        imagePullPolicy: Always
        name: nginx
        resources: {}
...
```

Проверим, что после обновления pod-модули были созданы заново:

```
[user@master ~]$ kubectl get pod -n new-deploy
NAME                            READY   STATUS    RESTARTS   AGE
nginx-deploy-7975cb855-5cm48   1/1     Running   0          45s
nginx-deploy-7975cb855-8v9kh   1/1     Running   0          29s
nginx-deploy-7975cb855-zxrlc   1/1     Running   0          34s
```

Убедимся, что переменные появились в описании модулей:

```
[user@master ~]$ kubectl describe pod nginx-deploy-7975cb855-8v9kh -n new-
deploy
Name:               nginx-deploy-7975cb855-8v9kh
Namespace:          new-deploy
...
      Started:      Fri, 12 Apr 201921:59:33 +0200
    Ready:          True
    Restart Count:  0
    Environment:
      APPUSERNAME:  <set to the key 'username' of config map 'cmap1'>
Optional: false
      APPPASSWORD:  <set to the key 'password' of config map 'cmap1'>
Optional: false
...
```

Также можно проверить наличие переменных при помощи команды env, запустив ее в контейнере:

```
[user@master ~]$ kubectl exec -it nginx-deploy-7975cb855-8v9kh -n new-deploy
-- env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=nginx-deploy-7975cb855-8v9kh
TERM=xterm
APPPASSWORD=p@ssw0rd
APPUSERNAME=andrey
...
```

Существует еще один способ передать сразу все содержимое карты в виде переменных, задав единый префикс. Изменения в описании внедрения должны выглядеть следующим образом:

```
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: nginx-deploy
    spec:
      containers:
      - envFrom:
        - configMapRef:
            name: cmap1
          prefix: APPCONF_
        image: nginx
        imagePullPolicy: Always
        name: nginx
```

Проверяем результат:

```
[user@master ~]$ kubectl exec -it nginx-deploy-5c6c869f6b-62c9z -n new-deploy
-- env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=nginx-deploy-5c6c869f6b-62c9z
TERM=xterm
APPCONF_password=p@ssw0rd
APPCONF_username=andrey
```

В конце данного раздела покажем, как передать карту конфигурации в виде файла. Как мы помним, у нас имеется карта, созданная из файла:

```
[user@master ~]$ kubectl get configmap passwdcm -n new-deploy
NAME       DATA   AGE
passwdcm   1      7m53s
[user@master ~]$ kubectl describe configmap passwdcm -n new-deploy
```

```
Name:         passwdcm
Namespace:    new-deploy
Labels:       <none>
Annotations:  <none>

Data
====
passwd:
----
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
...
```

Внесем изменения во внедрение:

```
 spec:
   containers:
   - image: nginx
     imagePullPolicy: Always
     name: nginx
     resources: {}
     terminationMessagePath: /dev/termination-log
     terminationMessagePolicy: File
     volumeMounts:
     - mountPath: /data
       name: passvol
       readOnly: true
   dnsPolicy: ClusterFirst
   restartPolicy: Always
   schedulerName: default-scheduler
   securityContext: {}
   terminationGracePeriodSeconds: 30
   volumes:
   - configMap:
       defaultMode: 420
       name: passwdcm
     name: passvol
```

После чего можно проверить содержимое файла /data/passwd в контейнере:

```
[user@master ~]$ kubectl exec -it nginx-deploy-68757b6568-fnqfb -n new-deploy
-- cat /data/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
...
```

Вопросы для самопроверки

1. **Для каких целей служит пространство имен?**

A. Ограничение видимости объектов
B. Фильтрация сетевого трафика
C. Создание каталога пользователей
D. Разграничение процессов GNU/Linux

2. **Какие объекты может создавать внедрение (укажите все правильные варианты)?**

A. Набор реплик
B. Контроллер репликации
C. pod-модуль
D. Пространство имен

Список ссылок

1. https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/

Глава 7.
РАСШИРЕННЫЕ ВОЗМОЖНОСТИ KUBERNETES

Контроллеры DaemonSet и StatefulSet

Рассмотрим еще два типа контроллеров, управляющих запуском и распределением pod-модулей по узлам. Первый из них – набор демонов (DaemonSet). Его назначение в том, чтобы убедиться в наличии pod-модуля на каждом узле кластера. Внедрение, с которым мы познакомились ранее, умеет поддерживать заданное число pod-модулей, но не гарантирует, что на каждом узде будет один модуль. DaemonSet удобно использовать для инфраструктурных модулей, например сбора метрик или журналирования. Начиная с Kubernetes v1.12 также можно исключить часть узлов из набора.

В качестве примера DaemonSet можно привести набор, управляющий pod-модулями kube-proxy. Как мы видим, на каждом узле запущено по одному модулю:

```
[user@master ~]$ kubectl get pods --all-namespaces -o wide
NAMESPACE     NAME                                      READY   STATUS
RESTARTS   AGE    IP             NODE                NOMINATED NODE
READINESS GATES
...
kube-system   kube-proxy-g557g                          1/1     Running
23         97d    10.0.3.4       master.test.local   <none>           <none>
kube-system   kube-proxy-nzkmn                          1/1     Running
17         97d    10.0.3.6       node2.test.local    <none>           <none>
kube-system   kube-proxy-pfk6t                          1/1     Running
17         97d    10.0.3.5       node1.test.local    <none>           <none>
```

Найти соответствующий DaemonSet можно командой kubectl get daemonsets:

```
[user@master ~]$ kubectl get daemonsets -n kube-system
NAME                     DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE
NODE SELECTOR                     AGE
```

```
...
kube-proxy                3         3         3       3            3
<none>                            97d
```

Уже знакомыми командами можно посмотреть информацию и вывести описание:

```
[user@master ~]$ kubectl describe daemonsets kube-proxy -n kube-system
Name:           kube-proxy
Selector:       k8s-app=kube-proxy
Node-Selector:  <none>
Labels:         k8s-app=kube-proxy
Annotations:    deprecated.daemonset.template.generation: 1
Desired Number of Nodes Scheduled: 3
Current Number of Nodes Scheduled: 3
Number of Nodes Scheduled with Up-to-date Pods: 3
Number of Nodes Scheduled with Available Pods: 3
Number of Nodes Misscheduled: 0
Pods Status:  3 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:           k8s-app=kube-proxy
  Annotations:      scheduler.alpha.kubernetes.io/critical-pod:
  Service Account:  kube-proxy
  Containers:
   kube-proxy:
    Image:      k8s.gcr.io/kube-proxy:v1.13.1
...
[user@master ~]$ kubectl get daemonsets kube-proxy -n kube-system -o yaml
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  creationTimestamp: "2019-01-06T15:31:53Z"
  generation: 1
  labels:
    k8s-app: kube-proxy
  name: kube-proxy
  namespace: kube-system
...
```

Второй тип контроллера, который мы рассмотрим в этой главе, – это набор с сохранением состояния (StatefulSet). Он переназначен для приложений, в которых экземпляры приложения должны иметь собственное постоянное имя и состояние. Свойства наборов с сохранением состояний:

- сохраняемые в процессе работы и уникальные сетевые идентификаторы;

- сохраняемое в процессе работы уникальное для каждого модуля постоянное хранилище;
- внедрение и масштабирование с сохранением заданного порядка (каждый модуль получает свой индекс с отсчетом от нуля);
- автоматическое обновление с сохранением определенного порядка.

Помимо постоянного сетевого имени, у каждого модуля в StatefulSet организация сети набора отличается тем, что доступ к модулям осуществляется при помощи так называемого headless-сервиса. У такого сервиса нет кластерного IP, поскольку нам не нужна балансировка нагрузки между уникальными модулями и необходимо обращаться к каждому модулю напрямую. Каждый pod-модуль получает свою DNS-запись. Хотя IP-адреса могут меняться вследствие перезапуска модуля, имя будет постоянным.

Для того чтобы создать StatefulSet, нам понадобится еще пара постоянных томов. На NFS-сервере создадим еще две директории и установим необходимые права доступа:

```
[root@master ~]# mkdir /exportdata2
[root@master ~]# mkdir /exportdata3
[root@master ~]# chmod 1777 /exportdata2
[root@master ~]# chmod 1777 /exportdata3
```

Добавим в конфигурационный файл /etc/exports еще две строки для новых директорий и попросим сервис перечитать конфигурацию:

```
[root@master ~]# cat /etc/exports
/exportdata *(rw,sync,no_root_squash,subtree_check)
/exportdata2 *(rw,sync,no_root_squash,subtree_check)
/exportdata3 *(rw,sync,no_root_squash,subtree_check)
[root@master ~]# exportfs -r
```

Проверим с одного из узлов, что две новые директории экспортированы с сервера:

```
[root@node1 ~]# showmount -e master
Export list for master:
/exportdata3 *
/exportdata2 *
/exportdata  *
```

Теперь необходимо создать два новых тома. Возьмем за основу файл pv.yaml из раздела «Постоянные тома и запросы постоянных

томов». Создадим два файла для двух новых директорий. Ниже приведен один из них. Изменения выделены:

```
[user@master ~]$ cat pv3.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pvnfs3
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  nfs:
    path: /exportdata3
    server: master
    readOnly: false
```

Создаем тома и проверяем их наличие:

```
[user@master ~]$ kubectl create -f pv2.yaml
persistentvolume/pvnfs2 created
[user@master ~]$ kubectl create -f pv3.yaml
persistentvolume/pvnfs3 created
[user@master ~]$ kubectl get pv
NAME     CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM
STORAGECLASS   REASON   AGE
pvnfs1   1Gi        RWX            Retain           Released    new-deploy/
pvc1                           3d18h
pvnfs2   1Gi        RWO            Retain           Available
6m9s
pvnfs3   1Gi        RWO            Retain           Available
5m49s
```

Скачаем файл примера из документации Kubernetes:

```
[user@master ~]$ wget https://raw.githubusercontent.com/kubernetes/website/
master/content/en/examples/application/web/web.yaml
```

В нем в строках с первой по тринадцатую определен «headless»-сервис («безголовый», или можно его назвать управляющий) и сам набор StatefulSet, начиная со строки четырнадцать:

```
[user@master ~]$ cat web.yaml | nl
     1  apiVersion: v1
     2  kind: Service
     3  metadata:
     4    name: nginx
```

```
    labels:
      app: nginx
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  serviceName: "nginx"
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: k8s.gcr.io/nginx-slim:0.8
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

Применяем файл для создания определенных в нем ресурсов:

```
[user@master ~]$ kubectl apply -f web.yaml
service/nginx created
statefulset.apps/web created
```

Проверяем, что создаются pod-модули с именами <имя набора>-индекс:

```
[user@master ~]$ kubectl get pod
NAME    READY   STATUS              RESTARTS   AGE
web-0   1/1     Running             0          50s
web-1   0/1     ContainerCreating   0          11s
```

Также проверяем, что создан сам набор:

```
[user@master ~]$ kubectl get statefulset web
NAME   READY   AGE
web    2/2     6m33s
```

Вы должны увидеть, что набор создал два запроса на постоянные тома:

```
[user@master ~]$ kubectl get pvc
NAME        STATUS   VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS   AGE
www-web-0   Bound    pvnfs2   1Gi        RWO                           8m44s
www-web-1   Bound    pvnfs3   1Gi        RWO                           3m45s
```

А два наших постоянных тома подключены к модулям:

```
[user@master ~]$ kubectl get pv
NAME     CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS     CLAIM
STORAGECLASS   REASON   AGE
pvnfs1   1Gi        RWX            Retain           Released   new-deploy/pvc
1                                 3d18h
pvnfs2   1Gi        RWO            Retain           Bound      default/www-
web-0                             5m8s
pvnfs3   1Gi        RWO            Retain           Bound      default/www-
web-1                             4m55s
```

Последнее, что мы сделаем, – это проверим работу сервиса при помощи утилиты nslookup, которая присутствует в образе busybox. Запустим контейнер и сделаем три запроса к DNS-серверу для определения имен web-0.nginx, web-1.nginx и nginx. Мы увидим, что нам возвращаются IP-адреса модулей и имена в домене nginx.default.svc.cluster.local, состоящем из домена svc.cluster.local с добавлением имени пространства имен и имени набора:

```
[user@master ~]$ kubectl run -i --tty --image busybox:1.28 dns-test
--restart=Never --rm

/ # nslookup web-0.nginx
Server:    10.96.0.10
Address 1: 10.96.0.10 kube-dns.kube-system.svc.cluster.local
```

```
Name:      web-0.nginx
Address 1: 10.244.2.162 web-0.nginx.default.svc.cluster.local
/ # nslookup web-1.nginx
Server:    10.96.0.10
Address 1: 10.96.0.10 kube-dns.kube-system.svc.cluster.local

Name:      web-1.nginx
Address 1: 10.244.1.105 web-1.nginx.default.svc.cluster.local
/ # nslookup nginx
Server:    10.96.0.10
Address 1: 10.96.0.10 kube-dns.kube-system.svc.cluster.local

Name:      nginx
Address 1: 10.244.1.105 web-1.nginx.default.svc.cluster.local
Address 2: 10.244.2.162 web-0.nginx.default.svc.cluster.local
```

Выполнение заданий при помощи Job и CronJob

В этом разделе мы рассмотрим еще два типа контроллеров: Job – для однократного выполнения задания и CronJob для периодического. В обоих случаях Kubernetes не перезапускает контейнер, когда процесс, запущенный внутри, завершается успешно. Если процесс возвращает код выхода с ошибкой, задание может быть настроено на перезапуск контейнера. Также задание будет повторно запущено на новом узле, если узел, на котором оно выполнялось, завершил работу аварийно.

Для демонстрации работы заданий Job также воспользуемся примером из официальной документации Kubernetes:

```
[user@master ~]$ wget https://raw.githubusercontent.com/kubernetes/website/
master/content/en/examples/controllers/job.yaml
```

Текст файла приведен ниже. Работа запускается в контейнере с Perl, после чего рассчитывается и выводится на стандартный вывод число пи. Как вы можете заметить, политика рестарта restartPolicy установлена в «никогда»:

```
[user@master ~]$ cat job.yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    spec:
      containers:
```

```
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
  backoffLimit: 4
```

После применения YAML-файла посмотрим на описание работы. Из строки «Pods Statuses» видно, что работа стартовала:

```
[user@master ~]$ kubectl create -f job.yaml
[user@master ~]$ kubectl describe jobs/pi
Name:           pi
Namespace:      default
Selector:       controller-uid=b3aea1f9-5eca-11e9-aea0-0800279bf286
Labels:         controller-uid=b3aea1f9-5eca-11e9-aea0-0800279bf286
                job-name=pi
Annotations:    <none>
Parallelism:    1
Completions:    1
Start Time:     Sun, 14 Apr 201917:33:50 +0200
Pods Statuses:  1 Running / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  controller-uid=b3aea1f9-5eca-11e9-aea0-0800279bf286
           job-name=pi
  Containers:
   pi:
    Image:      perl
    Port:       <none>
    Host Port:  <none>
    Command:
      perl
      -Mbignum=bpi
      -wle
      print bpi(2000)
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Events:
  Type    Reason            Age   From            Message
  ----    ------            ----  ----            -------
  Normal  SuccessfulCreate  8s    job-controller  Created pod: pi-4svcj
```

Проверим, что pod-модуль запущен:

```
[user@master ~]$ kubectl get pod
NAME       READY   STATUS    RESTARTS   AGE
pi-4svcj   1/1     Running   0          2m58s
```

По окончании работы модуль не удаляется, а сохраняется в состоянии «Completed» для анализа и просмотра журнала:

```
[user@master ~]$ kubectl get pod
NAME       READY   STATUS      RESTARTS   AGE
pi-4svcj   0/1     Completed   0          3m19s
```

При помощи команды kubectl logs можно просмотреть число пи до двухсотого знака:

```
[user@master ~]$ kubectl logs pi-4svcj
3.14159265358979323846264338327950288419 71...
```

Если нам нужно запланировать выполнение задачи в будущем или периодическое выполнение задач, то необходимо воспользоваться контроллером CronJob. Скачаем файл примера:

```
[user@master ~]$ wget https://raw.githubusercontent.com/kubernetes/website/
master/content/en/examples/application/job/cronjob.yaml
```

Создадим контроллер из файла и посмотрим на его содержимое. Основное отличие от предыдущего примера – это наличие строки с расписанием schedule: "*/1 * * * *" в формате Cron. Данный конкретный пример означает, что задание должно выполняться раз в минуту. Изучить описание формата Cron можно, задав команду man 5 crontab и прочитав соответствующее описание.

```
[user@master ~]$ kubectl create -f cronjob.yaml
[user@master ~]$ cat cronjob.yaml
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox
            args:
            - /bin/sh
            - -c
            - date; echo Hello from the Kubernetes cluster
          restartPolicy: OnFailure
```

Посмотрим на текущее состояние:

```
[user@master ~]$ kubectl get cronjob hello
NAME    SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
hello   */1 * * * *   False     0        <none>          21s
```

При помощи kubectl get jobs –watch можно смотреть за выполнением задания в реальном времени:

```
[user@master ~]$ kubectl get jobs --watch
NAME   COMPLETIONS   DURATION   AGE
hello-1555256520   0/1         0s
hello-1555256520   0/1   0s    0s
hello-1555256520   1/1   6s    6s
hello-1555256580   0/1         0s
hello-1555256580   0/1   0s    0s
hello-1555256580   1/1   5s    5s
hello-1555256640   0/1         0s
hello-1555256640   0/1   0s    0s
hello-1555256640   1/1   4s    4s
```

Одновременно изменится статус задания, показав текущую и последнюю активности:

```
[user@master ~]$ kubectl get cronjob hello
NAME    SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
hello   */1 * * * *   False     0        26s             4m7s
```

Наконец, можно убедиться, что pod-модули по завершении работы остаются:

```
[user@master ~]$ kubectl get pod
NAME                     READY   STATUS      RESTARTS   AGE
hello-1555256640-4hdrk   0/1     Completed   0          2m11s
hello-1555256700-bz92g   0/1     Completed   0          71s
hello-1555256760-h5tqf   0/1     Completed   0          11s
```

При помощи kubectl logs можно посмотреть на результаты выполнения заданий:

```
[user@master ~]$ kubectl logs hello-1555256640-4hdrk
Sun Apr 1415:44:14 UTC 2019
Hello from the Kubernetes cluster
```

В конце удалим задание, иначе у нас каждую минуту будут запускаться новые модули:

```
[user@master ~]$ kubectl delete cronjob hello
cronjob.batch "hello" deleted
```

Менеджер пакетов Helm

Кратко рассмотрим менеджер пакетов Helm [1], предназначенный для установки приложений в Kubernetes. Изначально Helm был разработан компанией Deis, которую купила компания Microsoft в 2017 году.

Helm состоит из четырех основных компонентов:

- **сервер Tiller**. Он отвечает за управление релизами и взаимодействует с API Kubernetes и клиентом Helm;
- **клиент Helm**. Клиент устанавливается на локальный компьютер и отвечает за взаимодействие с сервером, управление репозиториями, отправку схем на сервер, обновление или удаление релизов;
- **схемы (Charts)** – коллекция файлов, описывающих ресурсы Kubernetes;
- **репозитории схем** – онлайн-хранилища схем.

Преимущества Helm:

- управление сложностью приложений;
- простота обновлений приложений;
- простота обмена схемами;
- простота обновлений.

В этом разделе мы рассмотрим, как установить сервер и клиент, как искать схемы и как их устанавливать. Для начала скачаем и установим клиент helm:

```
[root@master ~]# wget https://storage.googleapis.com/kubernetes-helm/helm-
v2.13.1-linux-amd64.tar.gz
[root@master ~]# tar -zxvf helm-v2.13.1-linux-amd64.tar.gz
[root@master ~]# cp linux-amd64/helm /usr/local/bin/
```

Helm использует конфигурацию файла *Kubeconfig, поэтому предполагается, что у вас имеется соответствующий файл. Инициализируем клиента с установкой сервера Tiller:*

```
[user@master ~]$ helm init
Creating /home/user/.helm
Creating /home/user/.helm/repository
Creating /home/user/.helm/repository/cache
Creating /home/user/.helm/repository/local
Creating /home/user/.helm/plugins
Creating /home/user/.helm/starters
Creating /home/user/.helm/cache/archive
```

```
Creating /home/user/.helm/repository/repositories.yaml
Adding stable repo with URL: https://kubernetes-charts.storage.googleapis.com
Adding local repo with URL: http://127.0.0.1:8879/charts
$HELM_HOME has been configured at /home/user/.helm.

Tiller (the Helm server-side component) has been installed into your
Kubernetes Cluster.

Please note: by default, Tiller is deployed with an insecure 'allow
unauthenticated users' policy.
To prevent this, run `helm init` with the --tiller-tls-verify flag.
For more information on securing your installation see: https://docs.helm.sh/
using_helm/#securing-your-helm-installation
Happy Helming!
```

Проверим, что Tiller запущен в пространстве имен kube-system:

```
[user@master ~]$ kubectl get pods --namespace kube-system
NAME                                    READY   STATUS    RESTARTS   AGE
...
tiller-deploy-5f4fc5bcc6-rbzfd              1/1     Running   0          47s
...
```

Далее, для того чтобы дать Tiller возможность работы с правами администратора кластера, создадим сервисную учетную запись и внесем изменение во внедрение:

```
[user@master ~]$ kubectl create serviceaccount --namespace kube-system tiller
[user@master ~]$ kubectl create clusterrolebinding tiller-cluster-rule
--clusterrole=cluster-admin --serviceaccount=kube-system:tiller
[user@master ~]$ kubectl patch deploy --namespace kube-system tiller-deploy
-p '{"spec":{"template":{"spec":{"serviceAccount":"tiller"}}}}'
```

Обновим информацию о репозиториях:

```
[user@master ~]$ helm repo update
Hang tight while we grab the latest from your chart repositories...
...Skip local chart repository
...Successfully got an update from the "stable" chart repository
Update Complete. ⎈ Happy Helming!⎈
```

Попробуем найти схемы, связанные с mysql:

```
[user@master ~]$ helm search mysql
NAME                                    CHART VERSION   APP VERSION
DESCRIPTION
stable/mysql                            0.15.0          5.7.14          Fast,
reliable, scalable, and easy to use open-source rel...
stable/mysqldump                        2.4.0           2.4.0           A
Helm chart to help backup MySQL databases using mysqldump
```

```
stable/prometheus-mysql-exporter        0.3.2           v0.11.0         A
Helm chart for prometheus mysql exporter with cloudsqlp...
stable/percona                          0.3.4           5.7.17          free,
fully compatible, enhanced, open source drop-in rep...
stable/percona-xtradb-cluster           0.6.6           5.7.19          free,
fully compatible, enhanced, open source drop-in rep...
stable/phpmyadmin                       2.2.0           4.8.5
phpMyAdmin is an mysql administration frontend
stable/gcloud-sqlproxy                  0.6.1           1.11
DEPRECATED Google Cloud SQL Proxy
stable/mariadb                          5.11.1          10.1.38         Fast,
reliable, scalable, and easy to use open-source rel...
```

Проверим информацию о найденной схеме stable/mysql:

```
[user@master ~]$ helm inspect stable/mysql
appVersion: 5.7.14
description: Fast, reliable, scalable, and easy to use open-source relational
database
  system.
engine: gotpl
home: https://www.mysql.com/
icon: https://www.mysql.com/common/logos/logo-mysql-170x115.png
keywords:
- mysql
- database
- sql
...
---
## mysql image version
## ref: https://hub.docker.com/r/library/mysql/tags/
##
image: "mysql"
imageTag: "5.7.14"

busybox:
  image: "busybox"
  tag: "1.29.3"

testFramework:
  image: "dduportal/bats"
  tag: "0.4.0"

## Specify password for root user
##
## Default: random 10 character string
# mysqlRootPassword: testing
```

```
## Create a database user
##
# mysqlUser:
## Default: random 10 character string
# mysqlPassword:

...

## Configure the service
## ref: http://kubernetes.io/docs/user-guide/services/
service:
  annotations: {}
  ## Specify a service type
  ## ref: https://kubernetes.io/docs/concepts/services-networking/
service/#publishing-services---service-types
  type: ClusterIP
  port: 3306
  # nodePort: 32000
...

## Configuration

The following table lists the configurable parameters of the MySQL chart and
their default values.

| Parameter                                  | Description
| Default                                          |
| ------------------------------------------ | ---------------------------
---------------------------------------------------------------- | ----------
---------------------------------------- |
| `image`                                    | `mysql` image repository.
| `mysql`                                          |
| `imageTag`                                 | `mysql` image tag.
| `5.7.14`                                         |
| `busybox.
image`                                   | `busybox` image repository.
| `busybox`                                        |
| `busybox.tag`                              | `busybox` image tag.
| `1.29.3`                                         |
...
| `priorityClassName`                        | Set pod priorityClassName
| `{}`                                             |
```

Как вы видите, помимо всего прочего, схема содержит большое число настраиваемых параметров. Их можно передать во время установки приложения через файл или параметры команды helm. Установим схему без указания каких-либо параметров:

```
[user@master ~]$  helm install stable/mysql
NAME:   yucky-opossum
LAST DEPLOYED: Sun Apr 14 19:39:29 2019
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/ConfigMap
NAME                       DATA  AGE
yucky-opossum-mysql-test   1     0s

==> v1/PersistentVolumeClaim
NAME                  STATUS   VOLUME  CAPACITY  ACCESS MODES  STORAGECLASS
AGE
yucky-opossum-mysql   Pending  0s

==> v1/Pod(related)
NAME                                  READY  STATUS   RESTARTS  AGE
yucky-opossum-mysql-5c9d7c59c5-nc4d5  0/1    Pending  0         0s

==> v1/Secret
NAME                  TYPE    DATA  AGE
yucky-opossum-mysql   Opaque  2     0s

==> v1/Service
NAME                  TYPE       CLUSTER-IP    EXTERNAL-IP  PORT(S)   AGE
yucky-opossum-mysql   ClusterIP  10.97.55.203  <none>       3306/TCP  0s

==> v1beta1/Deployment
NAME                  READY  UP-TO-DATE  AVAILABLE  AGE
yucky-opossum-mysql   0/1    1           0          0s

NOTES:
MySQL can be accessed via port 3306 on the following DNS name from within
your cluster:
yucky-opossum-mysql.default.svc.cluster.local

To get your root password run:

    MYSQL_ROOT_PASSWORD=$(kubectl get secret --namespace default yucky-
opossum-mysql -o jsonpath="{.data.mysql-root-password}" | base64 --decode;
echo)

To connect to your database:

1. Run an Ubuntu pod that you can use as a client:

    kubectl run -i --tty ubuntu --image=ubuntu:16.04 --restart=Never -- bash
```

```
-il

2. Install the mysql client:

    $ apt-get update && apt-get install mysql-client -y

3. Connect using the mysql cli, then provide your password:
    $ mysql -h yucky-opossum-mysql -p

To connect to your database directly from outside the K8s cluster:
    MYSQL_HOST=127.0.0.1
    MYSQL_PORT=3306

    # Execute the following command to route the connection:
    kubectl port-forward svc/yucky-opossum-mysql 3306

    mysql -h ${MYSQL_HOST} -P${MYSQL_PORT} -u root -p${MYSQL_ROOT_PASSWORD}
```

Просмотрим список текущих установленных релизов:

```
[user@master ~]$ helm ls
NAME            REVISION        UPDATED                         STATUS
CHART           APP VERSION     NAMESPACE
yucky-opossum   1               Sun Apr 14 19:39:29 2019        DEPLOYED
mysql-0.15.0    5.7.14          default
```

При помощи команды helm status можно посмотреть на текущий статус ресурсов:

```
[user@master ~]$ helm status yucky-opossum
```

Команда helm delete удаляет релиз:

```
[user@master ~]$ helm delete yucky-opossum
release "yucky-opossum" deleted
```

Вопросы для самопроверки

1. **Какой из контроллеров подходит для запуска повторяющихся заданий?**

A. DaemonSet
B. StatefulSet
C. Job
D. CronJob

2. **Какой из контроллеров следит за тем, чтобы на каждом узле было по одному pod-модулю?**

A. DaemonSet
B. StatefulSet
C. Job
D. CronJob

3. **Какой из компонентов устанавливается локально на рабочую машину администратора?**

A. Tiller
B. Helm
C. Chartmuseum

Список ссылок

1. https://helm.sh

Глава 8.
ЗНАКОМСТВО С OPENSHIFT И OKD

В данной главе мы познакомимся с одним из популярных дистрибутивов Kubernetes, разрабатываемых компанией Red Hat. Дистрибутив имеет несколько вариантов и названий в зависимости от того, предоставляется ли он как сервис или как продукт, а также оказывается для него коммерческая поддержка или нет. В книге мы рассмотрим вариант под названием OKD (ранее OpenShift Origin)[1]. Это открытый проект, в котором происходит разработка всего семейства продуктов. В книге мы будем использовать OpenShift как синоним OKD или синоним «дистрибутив Kubernetes, разрабатываемый сообществом при поддержке Red Hat». Нужно сказать, что мы будем вести речь о OpenShift 3. Ранее существовали продукты OpenShift 2 и PaaS Makara, которые можно условно назвать «OpenShift 1». Эти продукты не основывались на Kubernetes и в книге не рассматриваются.

Сравнение OpenShift и Kubernetes

В целом OpenShift представляет собой Kubernetes с «добавками» и рядом изменений. Поскольку Red Hat создавал OpenShift до того, как в Kubernetes появились некоторые функции, необходимые корпоративному пользователю, эти функции могут быть реализованы иначе. Некоторые наработки проекта OpenShift в дальнейшем были приняты в Kubernetes, некоторые реализованы «с нуля», но иначе. В данной главе мы как раз и рассмотрим основные отличия на практике. Ключевые отличия приведены в табл. 8.1.

Функционал	OpenShift (OKD)	Kubernetes
Утилита командной строки	oc	kubectl
Разделение проектов/пользователей	Projects	Namespaces
Развертывание приложений	Deployment Configurations и Deployments	Deployments
Безопасность	Security Context Constraints	Pod Security Policy
Входящие подключения к контейнерам	Routes и Kubernetes Ingress controllers (с версии 3.10)	Kubernetes Ingress controllers
Отслеживание изменений образов	Image streams	Отсутствует
Сборка приложений	Build Config / S2I	Отсутствует
Шаблоны приложений	Templates	Helm charts

Таблица 8.1 ❖ Некоторые отличия OpenShift от Kubernetes

Установка OpenShift при помощи cluster up

Существует несколько вариантов пробной установки OpenShift, они отличаются между собой и сильно отличаются от установки OpenShift для промышленной эксплуатации, где требуется высокая доступность компонентов инфраструктуры. Мы рассмотрим установку при помощи стандартной утилиты командной строки, предназначенной для управления кластером, – oc. Данная утилита имеет команду cluster up, которая позволяет на единственном узле с установленным Docker развернуть инфраструктуру OpenShift в контейнерах. Необходимо отметить, что хотя существуют версии утилиты oc под операционные системы Windows и MacOS, команда cluster up работает только в RHEL/CentOS. Устанавливать Docker в CentOS мы научились в первой главе, поэтому сразу перейдем к установке OpenShift.

Для того чтобы установка завершилась успешно, нам необходимо разрешить обращения к встроенному внутреннему реестру OpenShift, который будет располагаться в сети 172.30.0.0/16:

```
[root@okd ~]# cat << EOF >/etc/docker/daemon.json
{
   "insecure-registries": [
     "172.30.0.0/16"
   ]
}
EOF
```

После этого перезапускаем Docker:

```
[root@okd ~]#  systemctl restart docker
```

Теперь можно установить утилиту oc, котрая используется как для установки, так и для управления кластером:

```
[root@okd ~]# yum -y install origin-clients
```

И наконец, единственная команда oc cluster up установит и запустит кластер, состоящий из одного узла:

```
[root@okd ~]# oc cluster up
Getting a Docker client ...
Checking if image openshift/origin-control-plane:v3.11 is available …
...
OpenShift server started.

The server is accessible via web console at:
    https://127.0.0.1:8443

You are logged in as:
    User:     developer
    Password: <any value>

To login as administrator:
    oc login -u system:admin
```

Для проверки функционирования кластера, как и предлагают, можно авторизоваться как администратор:

```
[root@okd ~]# oc login -u system:admin
Logged into "https://127.0.0.1:8443" as "system:admin" using existing
credentials.

You have access to the following projects and can switch between them with
'oc project <projectname>':

    default
    kube-dns
    kube-proxy
    kube-public
    kube-system
  * myproject
    openshift
    openshift-apiserver
    openshift-controller-manager
    openshift-core-operators
    openshift-infra
```

```
    openshift-node
    openshift-service-cert-signer
    openshift-web-console

Using project "myproject".
```

После чего можно вывести список рабочих узлов и убедиться, что у единственного узла статус Ready:

```
[root@okd ~]# oc get node
NAME        STATUS    ROLES     AGE       VERSION
localhost   Ready     <none>    4m        v1.11.0+d4cacc0
```

Также необходимо проверить, что все системные pod-модули в состоянии Running:

```
[root@okd ~]# oc get pods --all-namespaces
NAMESPACE                       NAME
READY     STATUS      RESTARTS   AGE
default                         docker-registry-1-5psr7
1/1       Running     0          1m
default                         persistent-volume-setup-jfk5v
0/1       Completed   0          2m
default                         router-1-d84sd
1/1       Running     0          1m
kube-dns                        kube-dns-cmskg
1/1       Running     0          4m
kube-proxy                      kube-proxy-lcpnq
1/1       Running     0          4m
kube-system                     kube-controller-manager-localhost
1/1       Running     0          4m
kube-system                     kube-scheduler-localhost
1/1       Running     0          3m
kube-system                     master-api-localhost
1/1       Running     0          3m
kube-system                     master-etcd-localhost
1/1       Running     0          4m
openshift-apiserver             openshift-apiserver-bvznn
1/1       Running     0          4m
openshift-controller-manager    openshift-controller-manager-mtgpd
1/1       Running     0          2m
openshift-core-operators        openshift-service-cert-signer-operator-
6d477f986b-hlp46   1/1       Running     0          4m
openshift-core-operators        openshift-web-console-operator-664b974ff5-
s62mt           1/1       Running     0          2m
openshift-service-cert-signer   apiservice-cabundle-injector-8ffbbb6dc-tmpsv
1/1       Running     0          4m
openshift-service-cert-signer   service-serving-cert-signer-668c45d5f-jmfh5
1/1       Running     0          4m
```

```
openshift-web-console          webconsole-7b5bc5786c-qsc5f
1/1       Running     0          1m
```

Обратите внимание, что для двух последних команд вы также могли использовать утилиту kubectl. Утилита oc в некоторой степени пересекается с kubectl как по синтаксису, так и по функционалу.

Убедившись, что кластер функционирует, можно переключиться под пользователя developer:

```
[root@okd ~]# oc login -u developer
Logged into "https://127.0.0.1:8443" as "developer" using existing
credentials.

You have one project on this server: "myproject"

Using project "myproject".
```

Если зайти браузером по адресу https://127.0.0.1:8443, то, введя имя пользователя developer и произвольный пароль (не забывайте, что речь идет о тестовой установке), вы увидите веб-консоль OpenShift. Внешний вид консоли приведен на рис. 8.1.

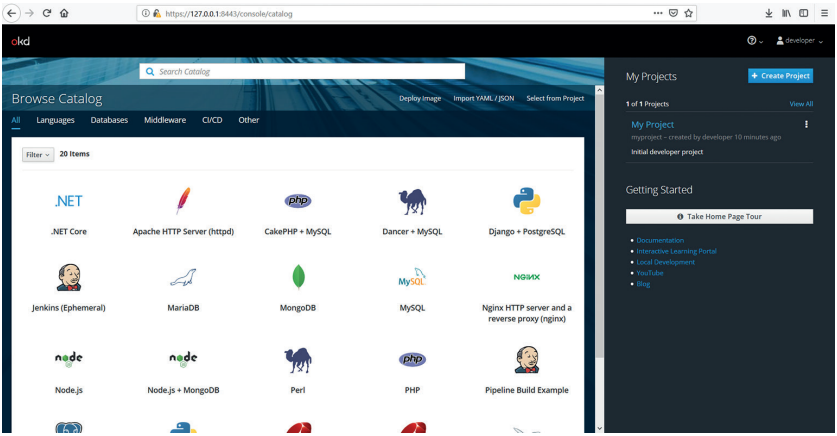


Рис. 8.1 ❖ Внешний вид веб-консоли OpenShift

Первое приложение в OpenShift

Первое, с чем мы познакомимся, – это команда oc new-app. Данная команда создает ресурсы, требуемые для сборки и развертывания приложения. Когда мы установили OpenShift при помощи

команды oc cluster up, утилита выдала сообщение, что создан и используется проект myproject. Проекты в OpenShift можно рассматривать как пространства имен Kubernetes с дополнительным функционалом, таким как, например, ролевая модель доступа. Ресурсы, создаваемые командой oc new-app, будут принадлежать текущему проекту myproject.

Зарегистрируемся как пользователь developer:

```
[root@okd ~]# oc login -u developer
Logged into "https://127.0.0.1:8443" as "developer" using existing
credentials.

You have one project on this server: "myproject"

Using project "myproject".
```

Теперь попробуем собрать демонстрационное приложение OpenShift «Hello World» на языке программирования Ruby:

```
[root@okd ~]# oc new-app https://github.com/openshift/ruby-hello-world.git
--> Found Docker image e42d0dc (9 months old) from Docker Hub for "centos/
ruby-22-centos7"

    Ruby 2.2
    --------
    Ruby 2.2 available as container is a base platform for building and
running various Ruby 2.2 applications and frameworks. Ruby is the interpreted
scripting language for quick and easy object-oriented programming. It has
many features to process text files and to do system management tasks (as in
Perl). It is simple, straight-forward, and extensible.

    Tags: builder, ruby, ruby22

    * An image stream tag will be created as "ruby-22-centos7:latest" that
will track the source image
    * A Docker build using source code from https://github.com/openshift/
ruby-hello-world.git will be created
      * The resulting image will be pushed to image stream tag "ruby-hello-
world:latest"
      * Every time "ruby-22-centos7:latest" changes a new build will be
triggered
    * This image will be deployed in deployment config "ruby-hello-world"
    * Port 8080/tcp will be load balanced by service "ruby-hello-world"
      * Other containers can access this service through the hostname "ruby-
hello-world"

--> Creating resources ...
```

```
    imagestream.image.openshift.io "ruby-22-centos7" created
    imagestream.image.openshift.io "ruby-hello-world" created
    buildconfig.build.openshift.io "ruby-hello-world" created
    deploymentconfig.apps.openshift.io "ruby-hello-world" created
    service "ruby-hello-world" created
--> Success
    Build scheduled, use 'oc logs -f bc/ruby-hello-world' to track its
progress.
    Application is not exposed. You can expose services to the outside world
by executing one or more of the commands below:
     'oc expose svc/ruby-hello-world'
    Run 'oc status' to view your app.
```

Одна команда дала нам достаточно большой и информативный вывод. Попытаемся разобраться, что произошло.

Утилита oc обратилась к заданному репозиторию кода на GitHub, где нашла Dockerfile, после этого oc использовала так называемую «стратегию» Docker. Другим вариантом стратегии может быть source, когда необходимо собрать приложение из исходного кода. Посмотрим на Dockerfile:

```
FROM centos/ruby-22-centos7
USER default
EXPOSE 8080
ENV RACK_ENV production
ENV RAILS_ENV production
COPY . /opt/app-root/src/
RUN scl enable rh-ruby22 "bundle install"
CMD ["scl", "enable", "rh-ruby22", "./run.sh"]

USER root
RUN chmod og+rw /opt/app-root/src/db
USER default
```

Важно отметить, что по умолчанию OpenShift запрещает запуск приложений в контейнерах пользователем root. В качестве базового образа используется centos/ruby-22-centos7. Видя это, утилита oc создает специальный объект под названием Image Stream, указывающий на образ, и загружает базовый образ во встроенный реестр OpenShift. Посмотрим описание этого объекта:

```
[root@okd ~]# oc describe is ruby-22-centos7
Name:                   ruby-22-centos7
Namespace:              myproject
Created:                32 minutes ago
Labels:                 app=ruby-hello-world
Annotations:            openshift.io/generated-by=OpenShiftNewApp
```

```
                        openshift.io/image.dockerRepositoryCheck=2019-03-
02T16:04:10Z
Docker Pull Spec:       172.30.1.1:5000/myproject/ruby-22-centos7
Image Lookup:           local=false
Unique Images:          1
Tags:                   1

latest
  tagged from centos/ruby-22-centos7

  * centos/ruby-22-centos7@
sha256:a18c8706118a5c4c9f1adf045024d2abf06ba632b5674b23421019ee4d3edcae
      32 minutes ago
```

Объекты Image Stream используются для отслеживания версий образов, и в случае обновления базового образа OpenShift может автоматически пересобрать и развернуть приложение, которое на нем основано. Установщик OpenShift создает несколько is в пространстве имен openshift во время установки:

```
[root@okd ~]# oc get is -n openshift
NAME         DOCKER REPO                            TAGS
UPDATED
dotnet       172.30.1.1:5000/openshift/dotnet       2.0,latest
About an hour ago
httpd        172.30.1.1:5000/openshift/httpd        2.4,latest
About an hour ago
jenkins      172.30.1.1:5000/openshift/jenkins      latest,1,2
About an hour ago
mariadb      172.30.1.1:5000/openshift/mariadb      10.1,10.2,latest
About an hour ago
...
wildfly      172.30.1.1:5000/openshift/wildfly      10.1,11.0,12.0 + 5
more...        About an hour ago
```

Для сборки образа контейнера был создан объект Build Config (bc), который отвечает за этот процесс. Посмотрим его описание:

```
[root@okd ~]# oc describe bc ruby-hello-world
Name:           ruby-hello-world
Namespace:      myproject
Created:        37 minutes ago
Labels:         app=ruby-hello-world
Annotations:    openshift.io/generated-by=OpenShiftNewApp
Latest Version: 1

Strategy:       Docker
URL:            https://github.com/openshift/ruby-hello-world.git
```

```
From Image:     ImageStreamTag ruby-22-centos7:latest
Output to:      ImageStreamTag ruby-hello-world:latest
...
```

Выбранная стратегия – Docker, и для собранного образа будет создан Image Stream под названием ruby-hello-world. Проверяем, что Image Stream под указанным именем был создан:

```
[root@okd ~]# oc get is
NAME               DOCKER REPO                                    TAGS
UPDATED
ruby-22-centos7    172.30.1.1:5000/myproject/ruby-22-centos7    latest    43
minutes ago
ruby-hello-world   172.30.1.1:5000/myproject/ruby-hello-world   latest    42
minutes ago
```

Если вернуться к последним строкам вывода после команды oc new-app, то можно увидеть, что на основании конфигурации сборки (Build Config) была запланирована сборка (Build). Вот эта строка:

```
Build scheduled, use 'oc logs -f bc/ruby-hello-world' to track its progress.
```

OpenShift предлагает проследить за ходом выполнения сборки при помощи команды oc logs -f bc/ruby-hello-world, чем мы и воспользуемся. Информация при этом выводится на консоль в реальном времени:

```
[root@okd ~]# oc logs -f bc/ruby-hello-world
Cloning "https://github.com/openshift/ruby-hello-world.git" ...
        Commit: 787f1beae9956c959c6af62ee43bfdda73769cf7 (Merge pull request
#78 from bparees/v22)
        Author: Ben Parees <bparees@users.noreply.github.com>
        Date:   Thu Jan 17 17:21:03 2019 -0500
Replaced Dockerfile FROM image centos/ruby-22-centos7
Step 1/13 : FROM centos/ruby-22-centos7@
sha256:a18c8706118a5c4c9f1adf045024d2abf06ba632b5674b23421019ee4d3edcae
 ---> e42d0dccf073
Step 2/13 : USER default
 ---> Using cache
 ---> 012fe48626f4
Step 3/13 : EXPOSE 8080
 ---> Using cache
 ---> 35f93ff4038d
...
Step 11/13 : USER default
 ---> Using cache
 ---> ddcb4ea6ea97
Step 12/13 : ENV "OPENSHIFT_BUILD_NAME" "ruby-hello-world-1" "OPENSHIFT_
```

```
BUILD_NAMESPACE" "myproject" "OPENSHIFT_BUILD_SOURCE" "https://
github.com/openshift/ruby-hello-world.git" "OPENSHIFT_BUILD_COMMIT"
"787f1beae9956c959c6af62ee43bfdda73769cf7"
 ---> Using cache
 ---> e9a4ed58d09a
Step 13/13 : LABEL "io.openshift.build.commit.author" "Ben Parees \
u003cbparees@users.noreply.github.com\u003e" "io.openshift.build.commit.
date" "Thu Jan 17 17:21:03 2019 -0500" "io.openshift.build.commit.id"
"787f1beae9956c959c6af62ee43bfdda73769cf7" "io.openshift.build.commit.
message" "Merge pull request #78 from bparees/v22" "io.openshift.build.
commit.ref" "master" "io.openshift.build.name" "ruby-hello-world-1" "io.
openshift.build.namespace" "myproject" "io.openshift.build.source-location"
"https://github.com/openshift/ruby-hello-world.git"
 ---> Using cache
 ---> 432acc07be01
Successfully built 432acc07be01
Pushing image 172.30.1.1:5000/myproject/ruby-hello-world:latest ...
Pushed 0/12 layers, 17% complete
...
Pushed 12/12 layers, 100% complete
Push successful
```

Сборка отработала успешно, и собранный образ попал во внутренний реестр, на который ссылается объект типа Image Stream под названием ruby-hello-world. Посмотрим на список сборок:

```
[root@okd ~]# oc get build
NAME                 TYPE      FROM          STATUS     STARTED
DURATION
ruby-hello-world-1   Docker    Git@787f1be   Complete   About an hour ago
58s
```

Один-единственный Build, завершившийся успешно. Возвращаемся к выводу команды oc new-app и смотрим на следующие строки:

```
    * This image will be deployed in deployment config "ruby-hello-world"
    * Port 8080/tcp will be load balanced by service "ruby-hello-world"
      * Other containers can access this service through the hostname "ruby-
hello-world"
```

После сборки приложения был создан новый объект Deployment Config с именем ruby-hello-world. Deployment Config (dc) – это объект, который с неким приближением можно назвать аналогом Deployment в Kubernetes. Объект Deployment Configuration (dc) представляет собой описание pod-модулей, создаваемых из одного и того же образа контейнера. Для созданных pod-модулей dc

поддерживает простейший механизм continuous delivery (непрерывной доставки) и обновлений. Таким образом, Deployment Config обладает бóльшим функционалом, чем Deployment. Посмотрим на этот объект:

```
[root@okd ~]# oc describe dc ruby-hello-world
Name:           ruby-hello-world
Namespace:      myproject
Created:        About an hour ago
Labels:         app=ruby-hello-world
Annotations:    openshift.io/generated-by=OpenShiftNewApp
Latest Version: 1
Selector:       app=ruby-hello-world,deploymentconfig=ruby-hello-world
Replicas:       1
Triggers:       Config, Image(ruby-hello-world@latest, auto=true)
Strategy:       Rolling
Template:
Pod Template:
  Labels:       app=ruby-hello-world
                deploymentconfig=ruby-hello-world
  Annotations:  openshift.io/generated-by=OpenShiftNewApp
  Containers:
   ruby-hello-world:
    Image:              172.30.1.1:5000/myproject/ruby-hello-world@
sha256:4dd3cc727e3c626f502f1346b09c9a381a68904421f066736271e2a245aed87a
    Port:               8080/TCP
    Host Port:          0/TCP
    Environment:        <none>
    Mounts:             <none>
  Volumes:              <none>

Deployment #1 (latest):
        Name:           ruby-hello-world-1
        Created:        about an hour ago
        Status:         Complete
        Replicas:       1 current / 1 desired
        Selector:       app=ruby-hello-world,deployment=ruby-hello-world-
1,deploymentconfig=ruby-hello-world
        Labels:         app=ruby-hello-world,openshift.io/deployment-config.
name=ruby-hello-world
        Pods Status:    1 Running / 0 Waiting / 0 Succeeded / 0 Failed

Events:
  Type          Reason                  Age     From
Message
  ----          ------                  ----    ----
-------
```

```
  Normal        DeploymentCreated       57m     deploymentconfig-controller
Created new replication controller "ruby-hello-world-1" for version 1
```

Как мы видим, у нас должна быть одна реплика, то есть один pod-модуль с собранным приложением:

```
[root@okd ~]# oc get pod
NAME                       READY     STATUS      RESTARTS   AGE
ruby-hello-world-1-7vd59   1/1       Running     0          59m
ruby-hello-world-1-build   0/1       Completed   0          1h
```

Помимо развернутого приложения, в pod-модуле ruby-hello-world-1-7vd59 мы видим отработавшую сборку ruby-hello-world-1-build. Возвращаемся к выводу oc new-app:

```
    service "ruby-hello-world" created
...
    Application is not exposed. You can expose services to the outside world
by executing one or more of the commands below:
     'oc expose svc/ruby-hello-world'
```

Был создан сервис ruby-hello-world, и, для того чтобы дать к нему доступ снаружи кластера, необходимо дать команду oc expose svc/ruby-hello-world. Проверим наличие сервиса:

```
[root@okd ~]# oc get svc ruby-hello-world
NAME               TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
ruby-hello-world   ClusterIP   172.30.140.35   <none>        8080/TCP   1h
```

Посмотреть на все созданные в проекте ресурсы можно командой oc get all:

```
[root@okd ~]# oc get all
NAME                              READY     STATUS      RESTARTS   AGE
pod/ruby-hello-world-1-7vd59      1/1       Running     0          1h
pod/ruby-hello-world-1-build      0/1       Completed   0          1h

NAME                                        DESIRED   CURRENT   READY     AGE
replicationcontroller/ruby-hello-world-1    1         1         1         1h

NAME                          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
AGE
service/ruby-hello-world      ClusterIP   172.30.140.35   <none>        8080/
TCP   1h
```

```
NAME                                                    REVISION   DESIRED
CURRENT   TRIGGERED BY
deploymentconfig.apps.openshift.io/ruby-hello-world   1          1         1
config,image(ruby-hello-world:latest)

NAME                                              TYPE      FROM      LATEST
buildconfig.build.openshift.io/ruby-hello-world   Docker    Git       1

NAME                                          TYPE      FROM          STATUS
STARTED             DURATION
build.build.openshift.io/ruby-hello-world-1   Docker    Git@787f1be
Complete   About an hour ago   58s

NAME                                              DOCKER REPO
TAGS      UPDATED
imagestream.image.openshift.io/ruby-22-centos7    172.30.1.1:5000/myproject/
ruby-22-centos7    latest    About an hour ago
imagestream.image.openshift.io/ruby-hello-world   172.30.1.1:5000/myproject/
ruby-hello-world   latest    About an hour ago
```

Последуем совету в конце вывода утилиты oc new-app и откроем приложение внешнему миру:

```
[root@okd ~]# oc expose svc/ruby-hello-world
route.route.openshift.io/ruby-hello-world exposed
```

При этом создастся объект, также отсутствующий в Kubernetes, – Route. Его назначение аналогично Kubernetes Ingress controller:

```
[root@okd ~]# oc get route
NAME               HOST/PORT                                    PATH
SERVICES           PORT       TERMINATION   WILDCARD
ruby-hello-world   ruby-hello-world-myproject.127.0.0.1.nip.io
ruby-hello-world   8080-tcp                 None
```

Осталось проверить работу приложения. Адрес, который мы узнали из вывода oc get route:

```
[root@okd ~]# curl http://ruby-hello-world-myproject.127.0.0.1.nip.io/
<!DOCTYPE html>
<html>
...
  <title>Hello from OpenShift v3!</title>
</head>
<body>
  <div class="page-header" align=center>
    <h1> Welcome to an OpenShift v3 Demo App! </h1>
...
```

На рис. 8.2 показано, как выглядит наше приложение в веб-консоли.

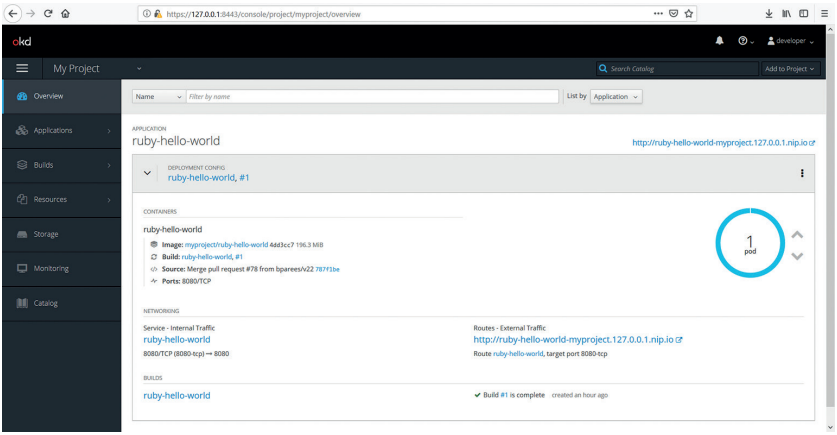


Рис. 8.2 ❖ Приложение ruby-hello-world в веб-консоли OpenShift

Удалим все ресурсы нашего тестового проекта. Воспользуемся тем, что OpenShift автоматически присваивает ресурсам метку с именем приложения. В нашем случае это ruby-hello-world:

```
[root@okd ~]# oc delete all -l app=ruby-hello-world
pod "ruby-hello-world-1-7vd59" deleted
replicationcontroller "ruby-hello-world-1" deleted
service "ruby-hello-world" deleted
deploymentconfig.apps.openshift.io "ruby-hello-world" deleted
buildconfig.build.openshift.io "ruby-hello-world" deleted
build.build.openshift.io "ruby-hello-world-1" deleted
imagestream.image.openshift.io "ruby-22-centos7" deleted
imagestream.image.openshift.io "ruby-hello-world" deleted
route.route.openshift.io "ruby-hello-world" deleted
```

Убедимся, что в проекте не осталось ресурсов:

```
[root@okd ~]# oc get all
No resources found.
```

Сборка приложений

В предыдущем разделе данной главы мы на практике познакомились с несколькими новыми объектами OpenShif, отсутствующими в Kubernetes. Рассмотрим их более подробно. Начнем с ресурса

Build Config (bc), описывающего то, как должна происходить сборка приложения. Данный ресурс используется встроенным в OpenShift компонентом Source-to-Image (S2I) [2].

Как было сказано выше, при этом применяется один из вариантов стратегии сборки:

- **Docker** – в рамках этой стратегии запускается команда docker build для создания нового образа. Для сборки требуется, чтобы Dockerfile и все необходимые артефакты были доступны в git-репозитории. Команда docker build отрабатывает в специально запускаемом pod-модуле с именем вида имя_приложения-номер_сборки-build;
- **Source** – данная стратегия предназначена для создания образа контейнера из исходного кода приложения. При этом OpenShift помещает код в совместимый с языком программирования или фреймворком сборочный образ;
- **Pipeline** – новый образ создается при помощи подключаемого модуля Jenkins pipeline;
- **Custom** – разработчик может определить свой собственный образ для сборки приложения. Это самая гибкая стратегия сборки.

Как правило, конфигурация сборки создается либо командой oc new-app, либо через веб-консоль, либо из командной строки командой oc create -f и заданием пути к файлу в формате YAML или JSON.

Для работы с Build Config создадим еще одно приложение, на этот раз при помощи стратегии source:

```
[root@okd ~]# oc new-app https://github.com/openshift/nodejs-ex
--> Found image 93de123 (4 months old) in image stream "openshift/nodejs"
under tag "10" for "nodejs"

    Node.js 10.12.0
...

    Tags: builder, nodejs, nodejs-10.12.0

    * The source repository appears to match: nodejs
    * A source build using source code from https://github.com/openshift/
nodejs-ex will be created
      * The resulting image will be pushed to image stream tag "nodejs-
ex:latest"
      * Use 'start-build' to trigger a new build
```

```
    * This image will be deployed in deployment config "nodejs-ex"
    * Port 8080/tcp will be load balanced by service "nodejs-ex"
      * Other containers can access this service through the hostname
"nodejs-ex"

--> Creating resources ...
    imagestream.image.openshift.io "nodejs-ex" created
    buildconfig.build.openshift.io "nodejs-ex" created
    deploymentconfig.apps.openshift.io "nodejs-ex" created
    service "nodejs-ex" created
--> Success
    Build scheduled, use 'oc logs -f bc/nodejs-ex' to track its progress.
    Application is not exposed. You can expose services to the outside world
by executing one or more of the commands below:
     'oc expose svc/nodejs-ex'
    Run 'oc status' to view your app.
```

Проверим, что Build Config отработал успешно:

```
[root@okd ~]# oc logs -f bc/nodejs-ex
Cloning "https://github.com/openshift/nodejs-ex" ...
        Commit: e59fe7571f883db2ae2e53d555aef6d145c6f032 (Merge pull request
#206 from liangxia/okd)
        Author: Honza Horak <hhorak@redhat.com>
        Date:   Tue Oct 16 15:45:10 2018 +0200
Using 172.30.1.1:5000/openshift/nodejs@
sha256:3cc041334eef8d5853078a0190e46a2998a70ad98320db512968f1de0561705e as
the s2i builder image
tar: scripts: time stamp 2019-03-06 14:46:44 is 0.098446386 s in the future
...
Pushing image 172.30.1.1:5000/myproject/nodejs-ex:latest ...
..
Push successful
```

Убедимся, что pod-модуль запущен и что Build отработал успешно:

```
[root@okd ~]# oc get pod
NAME                READY     STATUS      RESTARTS   AGE
nodejs-ex-1-build   0/1       Completed   0          2m
nodejs-ex-1-mgv2d   1/1       Running     0          1m
```

Пока присутствует только один объект Build Config, и его последний номер – единица:

```
[root@okd ~]# oc get bc
NAME        TYPE      FROM      LATEST
nodejs-ex   Source    Git       1
```

Посмотрим на сам объект Build Config. Для удобства обращения к строкам перенаправим вывод команды oc в утилиту nl, которая добавит номера строк:

```
[root@okd nodejs-ex]# oc get bc nodejs-ex -o yaml | nl
     1  apiVersion: build.openshift.io/v1
     2  kind: BuildConfig
     3  metadata:
     4    annotations:
     5      openshift.io/generated-by: OpenShiftNewApp
     6    creationTimestamp: 2019-03-07T08:28:37Z
     7    labels:
     8      app: nodejs-ex
     9    name: nodejs-ex
    10    namespace: myproject
    11    resourceVersion: "6414"
    12    selfLink: /apis/build.openshift.io/v1/namespaces/myproject/
buildconfigs/nodejs-ex
    13    uid: 01545677-40b3-11e9-967c-08002766cc3f
    14  spec:
    15    failedBuildsHistoryLimit: 5
    16    nodeSelector: null
    17    output:
    18      to:
    19        kind: ImageStreamTag
    20        name: nodejs-ex:latest
    21    postCommit: {}
    22    resources: {}
    23    runPolicy: Serial
    24    source:
    25      git:
    26        uri: https://github.com/openshift/nodejs-ex
    27      type: Git
    28    strategy:
    29      sourceStrategy:
    30        from:
    31          kind: ImageStreamTag
    32          name: nodejs:10
    33          namespace: openshift
    34      type: Source
```

```
    35    successfulBuildsHistoryLimit: 5
    36    triggers:
    37    - github:
    38        secret: VJhurL_mWDZsUJev8fT-
    39      type: GitHub
    40    - generic:
    41        secret: Jl9fu76lVuiZQocrzqrZ
    42      type: Generic
    43    - type: ConfigChange
    44    - imageChange:
    45        lastTriggeredImageID: 172.30.1.1:5000/openshift/nodejs@
sha256:3cc041334eef8d5853078a0190e46a2998a70ad98320db512968f1de0561705e
    46      type: ImageChange
    47  status:
    48    lastVersion: 1
```

Прокомментируем листинг:

- строка 2 – тип ресурса;
- строка 9 – имя ресурса Build Config;
- строка 10 – проект, в котором создан ресурс;
- строка 23 – runPolicy определяет, может ли сборка компонентов стартовать параллельно. Serial означает, что сборка должна происходить последовательно;
- строки 24–27 – где располагается исходный код;
- строки 36–46 – определяют триггеры, запускающие новую сборку. В нашем случае их три. Два – с секретами, сгенерированными OpenShift. Внешние приложения могут использовать эти секреты как часть webhook URL для запуска сборки. Третий триггер – это отслеживание изменения образа контейнера.

Пока у нас присутствует только одна сборка:

```
[root@okd ~]# oc get builds
NAME          TYPE      FROM          STATUS     STARTED         DURATION
nodejs-ex-1   Source    Git@e59fe75   Complete   3 minutes ago   1m18s
```

На рис. 8.3 и 8.4 приведено, как выглядит Build Config и последний Build в веб-консоли OpenShift.

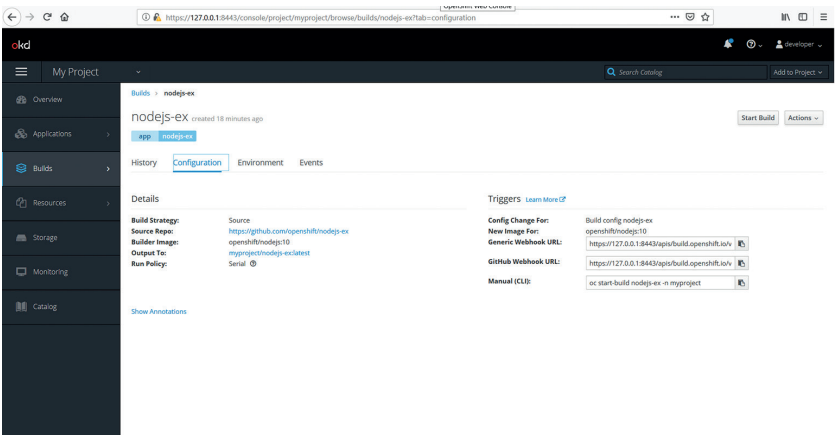


Рис. 8.3 ❖ Объект Build Config в веб-консоли OpenShift

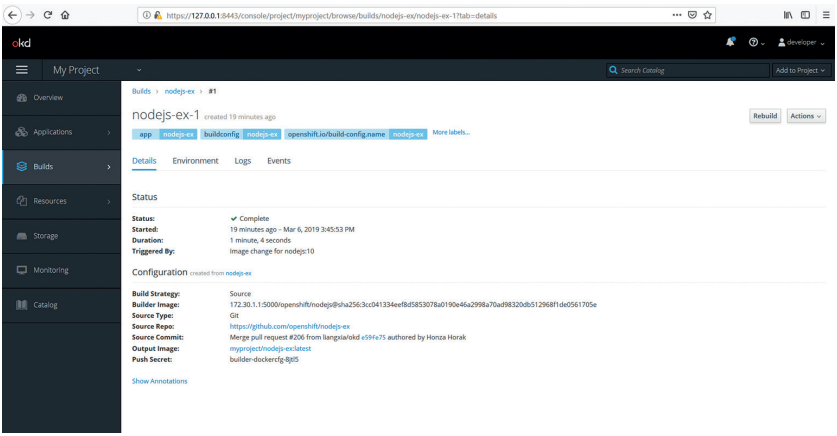


Рис. 8.4 ❖ Объект Build в веб-консоли OpenShift

Изменим код нашего приложения и запустим повторную сборку вручную. Для этого скачаем исходный код приложения:

```
[root@okd ~]# git clone  https://github.com/openshift/nodejs-ex
Cloning into 'nodejs-ex'...
remote: Enumerating objects: 631, done.
remote: Total 631 (delta 0), reused 0 (delta 0), pack-reused 631
Receiving objects: 100% (631/631), 233.74 KiB | 0 bytes/s, done.
Resolving deltas: 100% (244/244), done.
```

Поменяем текст приветствия с «Welcome to OpenShift» на «Welcome to OKD»:

```
[root@okd ~]# cd nodejs-ex/
[root@okd nodejs-ex]# sed -i 's/Welcome to OpenShift/Welcome to OKD/g' views/
index.html
```

Поскольку мы не можем загрузить код обратно на Git, воспользуемся опцией --from-dir=, которая позволяет стартовать сборку из заданной директории:

```
[root@okd nodejs-ex]# oc start-build nodejs-ex --from-dir="." --follow
Uploading directory "." as binary input for the build ...

Uploading finished
build.build.openshift.io/nodejs-ex-2 started
Receiving source from STDIN as archive ...
Using 172.30.1.1:5000/openshift/nodejs@
sha256:3cc041334eef8d5853078a0190e46a2998a70ad98320db512968f1de0561705e as
the s2i builder image
---> Installing application source
---> Building your Node application from source
...
Pushing image 172.30.1.1:5000/myproject/nodejs-ex:latest …
...
Push successful
```

Проверим, что последний номер сборки – теперь два:

```
[root@okd nodejs-ex]# oc get bc
NAME        TYPE      FROM      LATEST
nodejs-ex   Source    Git       2
```

В списке сборок у нас добавилась новая строка:

```
[root@okd nodejs-ex]# oc get builds
NAME          TYPE      FROM             STATUS     STARTED          DURATION
nodejs-ex-1   Source    Git@e59fe75      Complete   5 minutes ago    1m18s
nodejs-ex-2   Source    Binary@e59fe75   Complete   54 seconds ago   33s
```

Дождемся, когда pod-модуль с приложением запустится:

```
[root@okd nodejs-ex]# oc get pods
NAME                READY     STATUS      RESTARTS   AGE
nodejs-ex-1-build   0/1       Completed   0          5m
nodejs-ex-2-build   0/1       Completed   0          1m
nodejs-ex-2-klbwx   1/1       Running     0          32s
```

Откроем доступ к приложению, создав объект Route:

```
[root@okd nodejs-ex]# oc expose svc nodejs-ex
route.route.openshift.io/nodejs-ex exposed
```

Данный объект представляет собой имя хоста, которое OpenShift распознает как точку подключения к микросервису. Определим URL, по которому доступно приложение, и проверим, что выдается наша измененная строка «Welcome to OKD»:

```
[root@okd nodejs-ex]# oc get route
NAME        HOST/PORT                                PATH      SERVICES    PORT
TERMINATION   WILDCARD
nodejs-ex   nodejs-ex-myproject.127.0.0.1.nip.io             nodejs-ex
8080-tcp                  None
[root@okd nodejs-ex]# curl nodejs-ex-myproject.127.0.0.1.nip.io | grep OKD
...
  <title>Welcome to OKD</title>
...
```

Маршрут (Route) соединяет внешний IP-адрес и имя хоста с внутренним IP-адресом сервиса. Сам маршрутизатор реализован на базе HAProxy. Вы можете найти pod-модуль с именем router в проекте default. Для корректного разрешения имени на DNS-сервере должен быть настроен так называемый wildcard domain. Иными словами, все имена определенного поддомена должны разрешаться в IP-адреса балансировщика нагрузки OpenShift. В нашем тестовом кластере это реализовано проще, при помощи сервиса nip.io [3].

Продемонстрируем, как можно запустить сборку извне OpenShift при помощи webhook. Определим URL, по которому необходимо обратиться для запуска новой сборки. Искомый Webhook Generic выделен в выводе:

```
[root@okd nodejs-ex]# oc describe bc nodejs-ex
Name:           nodejs-ex
Namespace:      myproject
Created:        2 hours ago
Labels:         app=nodejs-ex
```

```
Annotations:    openshift.io/generated-by=OpenShiftNewApp
Latest Version: 2

Strategy:       Source
URL:            https://github.com/openshift/nodejs-ex
From Image:     ImageStreamTag openshift/nodejs:10
Output to:      ImageStreamTag nodejs-ex:latest

Build Run Policy:       Serial
Triggered by:           Config, ImageChange
Webhook GitHub:
        URL:    https://127.0.0.1:8443/apis/build.openshift.io/v1/namespaces/
myproject/buildconfigs/nodejs-ex/webhooks/<secret>/github
Webhook Generic:
        URL:            https://127.0.0.1:8443/apis/build.openshift.io/v1/
namespaces/myproject/buildconfigs/nodejs-ex/webhooks/<secret>/generic
        AllowEnv:       false
Builds History Limit:
        Successful:     5
        Failed:         5

Build           Status          Duration        Creation Time
nodejs-ex-2     complete        33s             2019-03-07 09:33:22 +0100 CET
nodejs-ex-1     complete        1m18s           2019-03-07 09:28:38 +0100 CET
```

Секрет, который необходимо подставить вместо <secret> в URL, возьмем из строки под номером 41 в вышеприведенной распечатке Build Config. Выполняем команду POST по сформированному таким образом URL:

```
[root@okd nodejs-ex]# curl -X POST -k https://127.0.0.1:8443/apis/build.
openshift.io/v1/namespaces/myproject/buildconfigs/nodejs-ex/webhooks/
Jl9fu76lVuiZQocrzqrZ/generic
```

Проверяем, что стартовала новая сборка:

```
[root@okd nodejs-ex]#oc get builds
NAME          TYPE      FROM             STATUS     STARTED       DURATION
nodejs-ex-1   Source    Git@e59fe75      Complete   2 hours ago   1m18s
nodejs-ex-2   Source    Binary@e59fe75   Complete   2 hours ago   33s
nodejs-ex-3   Source    Git              Pending
```

Спустя некоторое время будет запущен новый pod-модуль приложения:

```
[root@okd nodejs-ex]# oc get pod
NAME                READY     STATUS      RESTARTS   AGE
nodejs-ex-1-build   0/1       Completed   0          1h
nodejs-ex-2-build   0/1       Completed   0          1h
```

```
nodejs-ex-3-build   0/1       Completed   0          1m
nodejs-ex-3-hzc2k   1/1       Running     0          19s
```

На рис. 8.5 приведено, как история сборок выглядит в веб-консоли OpenShift.

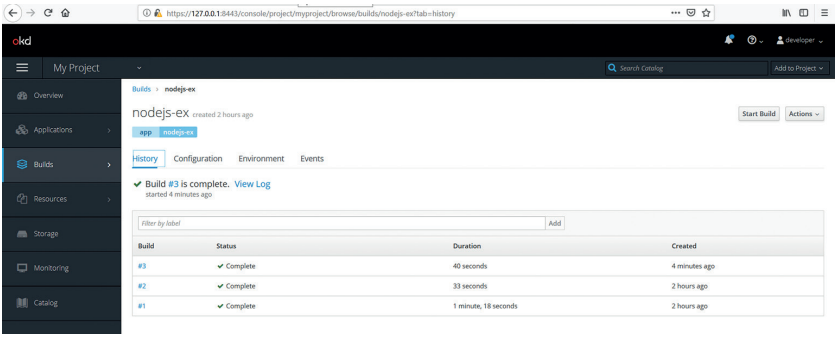


Рис. 8.5 ❖ История сборок в веб-консоли OpenShift

Работа с шаблонами OpenShift

Объект под названием Template (шаблон) определяет набор параметризированных объектов, на выходе создающий список объектов в OpenShift. Таким образом, можно объединить все объекты, необходимые для создания приложения, и вызывать их с необходимыми параметрами, получая различные экземпляры одного и того же приложения. Запускать шаблоны, иными словами, инициировать создание описанных в них объектов, можно как из командной строки, так и из веб-консоли. Шаблоны можно создавать в YAML или JSON, а далее загружать в выбранный проект при помощи команды oc create -f.

В комплекте с OpenShift уже идет набор шаблонов, который может помочь разработчикам быстро создавать приложения на различных языках программирования. По умолчанию шаблоны загружены в проект openshift:

```
[root@okd ~]# oc get templates -n openshift
NAME                        DESCRIPTION
PARAMETERS        OBJECTS
cakephp-mysql-persistent    An example CakePHP application with a MySQL
database. For more information ab...   20 (4 blank)      9
dancer-mysql-persistent     An example Dancer application with a MySQL
```

```
database. For more information abo...   17 (5 blank)      9
django-psql-persistent     An example Django application with a PostgreSQL
database. For more informatio...   20 (5 blank)      9
jenkins-ephemeral          Jenkins service, without persistent storage....
7 (all set)       6
jenkins-pipeline-example   This example showcases the new Jenkins Pipeline
integration in OpenShift,...       16 (4 blank)      8
mariadb-persistent         MariaDB database service, with persistent storage.
For more information about...   9 (3 generated)   4
mongodb-persistent         MongoDB database service, with persistent storage.
For more information about...   9 (3 generated)   4
mysql-persistent           MySQL database service, with persistent storage.
For more information about u...   9 (3 generated)   4
nodejs-mongo-persistent    An example Node.js application with a MongoDB
database. For more information...    19 (4 blank)      9
postgresql-persistent      PostgreSQL database service, with persistent
storage. For more information ab...   8 (2 generated)   4
rails-pgsql-persistent     An example Rails application with a PostgreSQL
database. For more information...   21 (4 blank)      9
```

Скачаем с GitHub пример более простого шаблона, чем загруженные автоматически:

```
[root@okd ~]#wget https://raw.githubusercontent.com/openshift/origin/master/
examples/db-templates/mysql-ephemeral-template.json
```

Данный пример представляет собой шаблон для запуска базы данных MySQL без постоянного хранилища. Создадим новый проект и загрузим в него скачанный шаблон в формате JSON:

```
[root@okd ~]# oc new-project mytemplates
Now using project "mytemplates" on server "https://127.0.0.1:8443".
...
[root@okd ~]# oc create -f mysql-ephemeral-template.json
template.template.openshift.io/mysql-ephemeral created
```

Проверим, что шаблон загружен:

```
[root@okd ~]# oc get template
NAME              DESCRIPTION
PARAMETERS        OBJECTS
mysql-ephemeral   MySQL database service, without persistent storage. For
more information abou...   8 (3 generated)   3
```

Как видно из описания, данный шаблон содержит 8 параметров, из которых три могут быть сгенерированы автоматически, и создает три объекта. Заново экспортируем шаблон, но уже в формат

YAML, и посмотрим на его основные части. Из-за большого объема вывод листинга приведен в приложении 2:

```
[root@okd ~]# oc get template mysql-ephemeral -o yaml | nl
...
```

Прокомментируем листинг:

- строка 2 – тип ресурса;
- строки 5–24 – опциональные описания и аннотации. Эти данные используются для вывода в веб-консоли и в других утилитах;
- строка 29 – имя ресурса template;
- строка 30 – имя проекта (пространства имен), в котором создан шаблон;
- строка 34 – тут начинается описание объектов, входящих в шаблон. Всего их три;
- строка 35 – начало описания объекта Secret;
- строка 49 – начало описания объекта Service;
- строка 68 – начало описания объекта Deployment Config, с которым мы познакомились в первом разделе;
- строка 158 – начало определения параметров.

Обратите внимание, что во всех остальных секциях используются параметры, определенные в секции parameters. Рассмотрим в качестве примера пароль для базы данных (строки 185–190):

```
185  - description: Password for the MySQL root user.
186    displayName: MySQL root user Password
187    from: '[a-zA-Z0-9]{16}'
188    generate: expression
189    name: MYSQL_ROOT_PASSWORD
190    required: true
```

Пароль используется для создания секрета в строке 47. В первых двух строках приводится описание и отображаемое имя параметра. OpenShift может самостоятельно для вас сгенерировать содержимое параметра. За это отвечают следующие две строки. И наконец, последние две строки – это имя параметра и признак того, что он обязательный.

На рис. 8.6–8.8 даны снимки с экрана веб-консоли OpenShift, показывающие запуск приложения из загруженного шаблона.

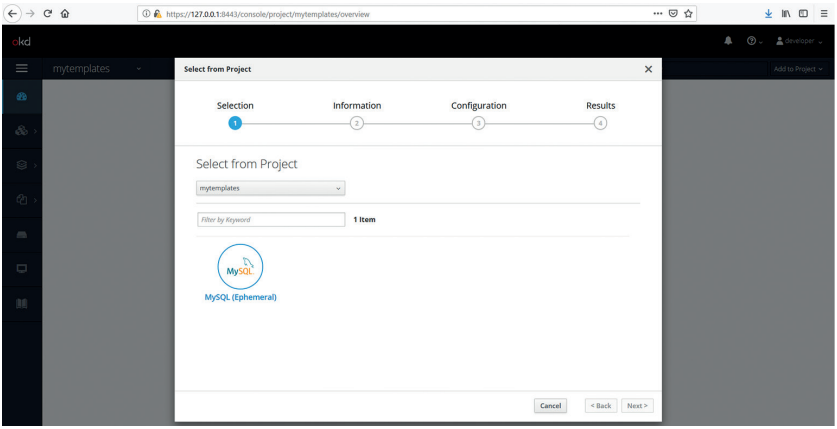


Рис. 8.6 ❖ Выбор шаблона в веб-консоли OpenShift

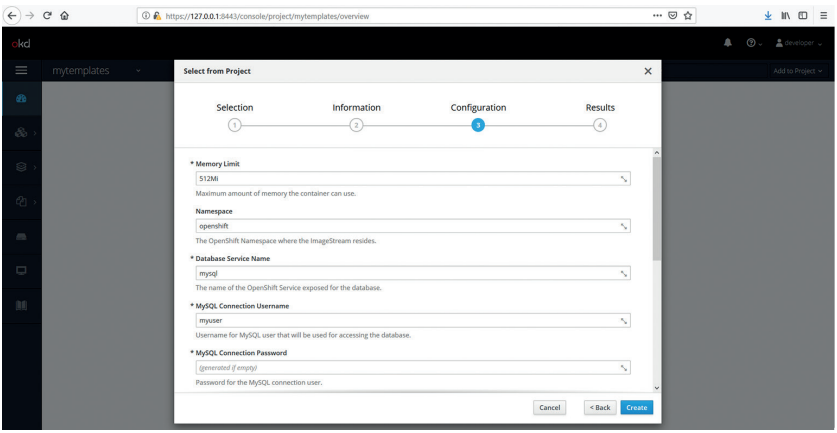


Рис. 8.7 ❖ Заполнение параметров шаблона в веб-консоли OpenShift

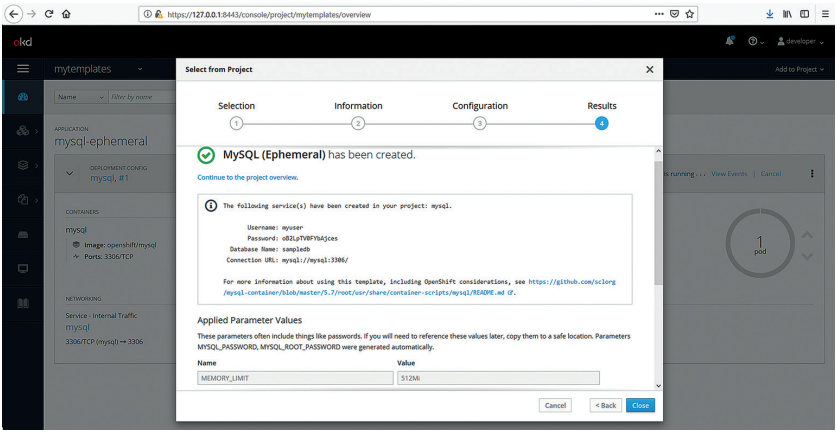


Рис. 8.8 ❖ Результат работы шаблона в веб-консоли OpenShift

Продемонстрируем создание приложения из командной строки при помощи ос new-app:

```
[root@okd ~]# oc new-app --template=mysql-ephemeral -p MYSQL_USER=myuser -p
MYSQL_PASSWORD=myP@ssw0rd
--> Deploying template "mytemplates/mysql-ephemeral" to project mytemplates

     MySQL (Ephemeral)
     ---------
     MySQL database service, without persistent storage.
...
     The following service(s) have been created in your project: mysql.

            Username: myuser
            Password: myP@ssw0rd
       Database Name: sampledb
      Connection URL: mysql://mysql:3306/

     For more information about using this template, including OpenShift
considerations, see https://github.com/sclorg/mysql-container/blob/
master/5.7/root/usr/share/container-scripts/mysql/README.md.
...
```

```
--> Creating resources ...
    secret "mysql" created
    service "mysql" created
    deploymentconfig.apps.openshift.io "mysql" created
--> Success
    Application is not exposed. You can expose services to the outside world
by executing one or more of the commands below:
     'oc expose svc/mysql'
    Run 'oc status' to view your app.
```

Проверим подключение к базе данных при помощи заданных во время запуска шаблона параметров. Узнаем имя Pod-модуля, после чего при помощи oc port-forward перенаправим порт 3306 из контейнера на порт 10 000 локального узла:

```
[root@okd ~]# oc get pod
NAME            READY     STATUS    RESTARTS   AGE
mysql-1-z9lf7   1/1       Running   0          14m
[root@okd ~]oc port-forward  mysql-1-z9lf7 10000:3306
Forwarding from 127.0.0.1:10000 -> 3306
Forwarding from [::1]:10000 -> 3306
```

Теперь можно подключиться к базе данных:

```
[root@okd ~]# mysql -umyuser -pmyP@ssw0rd -h127.0.0.1 -P10000
Handling connection for 10000
Welcome to the MariaDB monitor.  Commands end with ; or \g.
...
MySQL [(none)]>
```

На этом мы заканчиваем знакомство с OpenShift. Для дальнейшего изучения автор рекомендует ресурс [4], содержащий бесплатные интерактивные курсы по OpenShift.

Вопросы для самопроверки

1. **Для чего используется Image Stream (укажите все правильные ответы)?**

A. Для отслеживания версий образов
B. Для хранения нескольких версий образов
C. Для сборки образов контейнеров
D. Все перечисленное выше

2. **Для чего предназначен объект Deployment Config (укажите все правильные ответы)?**

A. Он представляет собой описание pod-модулей
B. Поддерживает механизм непрерывной доставки для pod-модулей
C. Поддерживает механизм входящих подключений к контейнерам
D. Все перечисленное выше

3. **Какие ресурсы будут созданы после применения команды oc new-app (укажите все правильные ответы)?**

A. Service
B. Route
C. Pod
D. Deployment Config

Список ссылок

1. https://www.okd.io/
2. https://github.com/openshift/source-to-image
3. http://nip.io/
4. https://learn.openshift.com/

ЗАКЛЮЧЕНИЕ

Данная книга знакомит лишь с основами соответствующих технологий и, как и следует из названия, призвана быть только введением в мир контейнеров и Kubernetes. Надеюсь, что она оказалась полезным подспорьем в этом начинании. Информацию об обнаруженных ошибках и опечатках можно отправлять автору по электронной почте amarkelov@yandex.ru.

ОТВЕТЫ К ВОПРОСАМ ДЛЯ САМОПРОВЕРКИ

Глава 1
1. A, C
2. B, D
3. A, C

Глава 2
1. A
2. B
3. B
4. B

Глава 3
1. B
2. C
3. B

Глава 4
1. A, C, D
2. A, B, C

Глава 5
1. C
2. C

Глава 6
1. A
2. A, C

Глава 7
1. D
2. A
3. B

Глава 8
1. A
2. A, B
3. A, C, D

ПРИЛОЖЕНИЯ

Приложение 1

Листинг внедрения nginx

```
 1  apiVersion: v1
 2  items:
 3  - apiVersion: extensions/v1beta1
 4    kind: Deployment
 5    metadata:
 6      annotations:
 7        deployment.kubernetes.io/revision: "1"
 8      creationTimestamp: "2019-04-07T10:06:29Z"
 9      generation: 1
10      labels:
11        app: nginx-deploy
12      name: nginx-deploy
13      namespace: new-deploy
14      resourceVersion: "4376"
15      selfLink: /apis/extensions/v1beta1/namespaces/new-deploy/
deployments/nginx-deploy
16      uid: cffc5bbe-591c-11e9-8edc-628057a8c709
17    spec:
18      progressDeadlineSeconds: 600
19      replicas: 1
20      revisionHistoryLimit: 10
21      selector:
22        matchLabels:
23          app: nginx-deploy
24      strategy:
25        rollingUpdate:
26          maxSurge: 25%
27          maxUnavailable: 25%
28        type: RollingUpdate
29      template:
30        metadata:
31          creationTimestamp: null
32          labels:
33            app: nginx-deploy
34        spec:
35          containers:
36          - image: nginx
37            imagePullPolicy: Always
```

```
    38            name: nginx
    39            resources: {}
    40            terminationMessagePath: /dev/termination-log
    41            terminationMessagePolicy: File
    42          dnsPolicy: ClusterFirst
    43          restartPolicy: Always
    44          schedulerName: default-scheduler
    45          securityContext: {}
    46          terminationGracePeriodSeconds: 30
    47    status:
    48      availableReplicas: 1
    49      conditions:
    50      - lastTransitionTime: "2019-04-07T10:06:40Z"
    51        lastUpdateTime: "2019-04-07T10:06:40Z"
    52        message: Deployment has minimum availability.
    53        reason: MinimumReplicasAvailable
    54        status: "True"
    55        type: Available
    56      - lastTransitionTime: "2019-04-07T10:06:29Z"
    57        lastUpdateTime: "2019-04-07T10:06:40Z"
    58        message: ReplicaSet "nginx-deploy-6d78f8cf68" has successfully
progressed.
    59        reason: NewReplicaSetAvailable
    60        status: "True"
    61        type: Progressing
    62      observedGeneration: 1
    63      readyReplicas: 1
    64      replicas: 1
    65      updatedReplicas: 1
    66  - apiVersion: extensions/v1beta1
    67    kind: ReplicaSet
    68    metadata:
    69      annotations:
    70        deployment.kubernetes.io/desired-replicas: "1"
    71        deployment.kubernetes.io/max-replicas: "2"
    72        deployment.kubernetes.io/revision: "1"
    73      creationTimestamp: "2019-04-07T10:06:29Z"
    74      generation: 1
    75      labels:
    76        app: nginx-deploy
    77        pod-template-hash: "2834947924"
    78      name: nginx-deploy-6d78f8cf68
    79      namespace: new-deploy
    80      ownerReferences:
    81      - apiVersion: apps/v1
    82        blockOwnerDeletion: true
    83        controller: true
    84        kind: Deployment
```

```
    85        name: nginx-deploy
    86        uid: cffc5bbe-591c-11e9-8edc-628057a8c709
    87      resourceVersion: "4375"
    88      selfLink: /apis/extensions/v1beta1/namespaces/new-deploy/
replicasets/nginx-deploy-6d78f8cf68
    89      uid: d00f04ff-591c-11e9-8edc-628057a8c709
    90    spec:
    91      replicas: 1
    92      selector:
    93        matchLabels:
    94          app: nginx-deploy
    95          pod-template-hash: "2834947924"
    96      template:
    97        metadata:
    98          creationTimestamp: null
    99          labels:
   100            app: nginx-deploy
   101            pod-template-hash: "2834947924"
   102        spec:
   103          containers:
   104          - image: nginx
   105            imagePullPolicy: Always
   106            name: nginx
   107            resources: {}
   108            terminationMessagePath: /dev/termination-log
   109            terminationMessagePolicy: File
   110          dnsPolicy: ClusterFirst
   111          restartPolicy: Always
   112          schedulerName: default-scheduler
   113          securityContext: {}
   114          terminationGracePeriodSeconds: 30
   115    status:
   116      availableReplicas: 1
   117      fullyLabeledReplicas: 1
   118      observedGeneration: 1
   119      readyReplicas: 1
   120      replicas: 1
```

Приложение 2
Листинг шаблона OpenShift mysql-ephemeral

```
apiVersion: template.openshift.io/v1
kind: Template
labels:
  template: mysql-ephemeral-template
message: |-
  The following service(s) have been created in your project:
${DATABASE_SERVICE_NAME}.
         Username: ${MYSQL_USER}
         Password: ${MYSQL_PASSWORD}
    Database Name: ${MYSQL_DATABASE}
   Connection URL: mysql://${DATABASE_SERVICE_NAME}:3306/
  For more information about using this template, including
OpenShift considerations, see https://github.com/sclorg/mysql-container/blob/
master/5.7/root/usr/share/container-scripts/mysql/README.md.
metadata:
  annotations:
    description: |-
      MySQL database service, without persistent storage. For more
information about using this template, including OpenShift considerations,
see https://github.com/sclorg/mysql-container/blob/master/5.7/root/usr/share/
container-scripts/mysql/README.md.
      WARNING: Any data stored will be lost upon pod destruction.
Only use this template for testing
    iconClass: icon-mysql-database
    openshift.io/display-name: MySQL (Ephemeral)
    openshift.io/documentation-url: https://docs.okd.io/latest/using_
images/db_images/mysql.html
    openshift.io/long-description: This template provides a
standalone MySQL server
      with a database created.  The database is not stored on
persistent storage,
      so any restart of the service will result in all data being
lost.  The database
      name, username, and password are chosen via parameters when
provisioning this
      service.
    openshift.io/provider-display-name: Red Hat, Inc.
    openshift.io/support-url: https://access.redhat.com
    tags: database,mysql
  creationTimestamp: 2019-03-24T13:01:44Z
  name: mysql-ephemeral
  namespace: mytemplates
  resourceVersion: "9603"
  selfLink: /apis/template.openshift.io/v1/namespaces/mytemplates/
```

```
templates/mysql-ephemeral
    33    uid: f960574c-4e34-11e9-8497-08002766cc3f
    34  objects:
    35  - apiVersion: v1
    36    kind: Secret
    37    metadata:
    38      annotations:
    39        template.openshift.io/expose-database_name: '{.data[''database-
name'']}'
    40        template.openshift.io/expose-password: '{.data[''database-
password'']}'
    41        template.openshift.io/expose-root_password: '{.data[''database-
root-password'']}'
    42        template.openshift.io/expose-username: '{.data[''database-
user'']}'
    43      name: ${DATABASE_SERVICE_NAME}
    44    stringData:
    45      database-name: ${MYSQL_DATABASE}
    46      database-password: ${MYSQL_PASSWORD}
    47      database-root-password: ${MYSQL_ROOT_PASSWORD}
    48      database-user: ${MYSQL_USER}
    49  - apiVersion: v1
    50    kind: Service
    51    metadata:
    52      annotations:
    53        template.openshift.io/expose-uri: mysql://{.spec.clusterIP}:{.
spec.ports[?(.name=="mysql")].port}
    54      name: ${DATABASE_SERVICE_NAME}
    55    spec:
    56      ports:
    57      - name: mysql
    58        nodePort: 0
    59        port: 3306
    60        protocol: TCP
    61        targetPort: 3306
    62      selector:
    63        name: ${DATABASE_SERVICE_NAME}
    64      sessionAffinity: None
    65      type: ClusterIP
    66    status:
    67      loadBalancer: {}
    68  - apiVersion: v1
    69    kind: DeploymentConfig
    70    metadata:
    71      annotations:
    72        template.alpha.openshift.io/wait-for-ready: "true"
    73      name: ${DATABASE_SERVICE_NAME}
    74    spec:
```

```
    replicas: 1
    selector:
      name: ${DATABASE_SERVICE_NAME}
    strategy:
      type: Recreate
    template:
      metadata:
        labels:
          name: ${DATABASE_SERVICE_NAME}
      spec:
        containers:
        - capabilities: {}
          env:
          - name: MYSQL_USER
            valueFrom:
              secretKeyRef:
                key: database-user
                name: ${DATABASE_SERVICE_NAME}
          - name: MYSQL_PASSWORD
            valueFrom:
              secretKeyRef:
                key: database-password
                name: ${DATABASE_SERVICE_NAME}
          - name: MYSQL_ROOT_PASSWORD
            valueFrom:
              secretKeyRef:
                key: database-root-password
                name: ${DATABASE_SERVICE_NAME}
          - name: MYSQL_DATABASE
            valueFrom:
              secretKeyRef:
                key: database-name
                name: ${DATABASE_SERVICE_NAME}
          image: ' '
          imagePullPolicy: IfNotPresent
          livenessProbe:
            initialDelaySeconds: 30
            tcpSocket:
              port: 3306
            timeoutSeconds: 1
          name: mysql
          ports:
          - containerPort: 3306
            protocol: TCP
          readinessProbe:
            exec:
              command:
              - /bin/sh
```

```
   123                - -i
   124                - -c
   125                - MYSQL_PWD="$MYSQL_PASSWORD" mysql -h 127.0.0.1 -u
$MYSQL_USER -D $MYSQL_DATABASE
   126                  -e 'SELECT 1'
   127              initialDelaySeconds: 5
   128              timeoutSeconds: 1
   129            resources:
   130              limits:
   131                memory: ${MEMORY_LIMIT}
   132            securityContext:
   133              capabilities: {}
   134              privileged: false
   135            terminationMessagePath: /dev/termination-log
   136            volumeMounts:
   137            - mountPath: /var/lib/mysql/data
   138              name: ${DATABASE_SERVICE_NAME}-data
   139          dnsPolicy: ClusterFirst
   140          restartPolicy: Always
   141          volumes:
   142          - emptyDir:
   143              medium: ""
   144            name: ${DATABASE_SERVICE_NAME}-data
   145      triggers:
   146      - imageChangeParams:
   147          automatic: true
   148          containerNames:
   149          - mysql
   150          from:
   151            kind: ImageStreamTag
   152            name: mysql:${MYSQL_VERSION}
   153            namespace: ${NAMESPACE}
   154          lastTriggeredImage: ""
   155        type: ImageChange
   156      - type: ConfigChange
   157    status: {}
   158  parameters:
   159  - description: Maximum amount of memory the container can use.
   160    displayName: Memory Limit
   161    name: MEMORY_LIMIT
   162    required: true
   163    value: 512Mi
   164  - description: The OpenShift Namespace where the ImageStream resides.
   165    displayName: Namespace
   166    name: NAMESPACE
   167    value: openshift
   168  - description: The name of the OpenShift Service exposed for the
database.
```

```
   169    displayName: Database Service Name
   170    name: DATABASE_SERVICE_NAME
   171    required: true
   172    value: mysql
   173  - description: Username for MySQL user that will be used for
accessing the database.
   174    displayName: MySQL Connection Username
   175    from: user[A-Z0-9]{3}
   176    generate: expression
   177    name: MYSQL_USER
   178    required: true
   179  - description: Password for the MySQL connection user.
   180    displayName: MySQL Connection Password
   181    from: '[a-zA-Z0-9]{16}'
   182    generate: expression
   183    name: MYSQL_PASSWORD
   184    required: true
   185  - description: Password for the MySQL root user.
   186    displayName: MySQL root user Password
   187    from: '[a-zA-Z0-9]{16}'
   188    generate: expression
   189    name: MYSQL_ROOT_PASSWORD
   190    required: true
   191  - description: Name of the MySQL database accessed.
   192    displayName: MySQL Database Name
   193    name: MYSQL_DATABASE
   194    required: true
   195    value: sampledb
   196  - description: Version of MySQL image to be used (5.7, or latest).
   197    displayName: Version of MySQL Image
   198    name: MYSQL_VERSION
   199    required: true
   200    value: "5.7"
```

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс»
наложенным платежом, выслав открытку или письмо по почтовому
адресу: **115487, г. Москва, 2-й Нагатинский пр-д, д. 6А**.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине:
www.alians-kniga.ru.

Оптовые закупки: тел. +7(499) 782-38-89
Электронный адрес: **books@alians-kniga.ru**.

Маркелов А.А.

Введение в технологии контейнеров и Kubernetes

Главный редактор *Мовчан Д.А.*
dmkpress@gmail.com
Корректор *Синяева Г.И.*
Верстка *Антонова А.И.*
Дизайн обложки *Мовчан А.Г.*

Формат 60×90 $^{1}/_{16}$.
Печать цифровая. Усл. печ. л. 12,13.
Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**