Парминдер Сингх Кочер

Микросервисы и контейнеры Docker

ДМК Пресс
издательство
Москва, 2019

Parminder Singh Kocher

Microservices and Containers

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Парминдер Сингх Кочер

Микросервисы и контейнеры Docker

УДК 004.451Docker
ББК 32.972.1
К55

Кочер П. С.
К55 Микросервисы и контейнеры Docker / пер. с анг. А. Н. Киселева. – М.: ДМК Пресс, 2019. – 240 с.: ил.

ISBN 978-5-97060-739-8

В данной книге анализируется два самых горячих направления развития информационных технологий: микросервисы и контейнеры Docker. Вместе они способны обеспечить беспрецедентную гибкость в разработке и развертывании приложений, а также в их масштабировании. Вы узнаете, как использовать микросервисы и Docker для создания модульной архитектуры, увеличения производительности и надежности приложений, уменьшения времени до выхода на рынок, повторного использования кода и экспоненциального повышения эффективности в DevOps.

Издание рекомендовано архитекторам и разработчикам ПО, а также будет полезно руководителям, стремящимся уйти от устаревших подходов и максимизировать успех своего бизнеса.

УДК 004.451Docker
ББК 32.972.1

ISBN 978-0-13-459838-3 (анг.)
ISBN 978-5-97060-739-8 (рус.)

Посвящается моим маме и папе.
Без их любви и родительского благословения
появление этой книги было бы невозможно.

Содержание

Введение

Как всегда, технологический сектор находится в гуще важных перемен. Вот лишь некоторые примеры, связанные с возможностями интернета: сети с программной поддержкой и предоставление программного обеспечения как услуги (Software as a Service, SaaS). Благодаря этим инновациям возник большой спрос на платформы и архитектуры, способные улучшить процесс разработки и развертывания приложений. И небольшие, и крупные компании нуждаются во фреймворках и архитектурах, упрощающих процесс обновления их приложений и позволяющих чаще выводить на рынок новые версии.

Эти и многие другие перемены остаются пока достаточно новыми, и все же за время их существования успело появиться и исчезнуть множество технологий и фреймворков. Однако наиболее успешные остаются и продолжают помогать совершенствовать мир программного обеспечения, позволяя разработчикам – нам с вами – создавать новые и обновлять существующие приложения с еще большей гибкостью, чем прежде. К двум таким успешным технологиям относятся микросервисы и контейнеры. По сравнению с широко используемым монолитным подходом к разработке и развертыванию приложений, микросервисы упрощают эти процессы, особенно в крупных проектах, требующих совместной работы нескольких групп и все более длинного кода. В таких проектах даже небольшое изменение в коде может вызвать серьезные задержки. Современные микросервисы способны объединять большие объемы кода, обеспечивая гибкость и масштабируемость разработки и развертывания приложений, и все это в рамках проверенной парадигмы.

Когда я впервые начал знакомиться с микросервисами, существовало несколько ценных интернет-ресурсов, таких как microservices.io Криса Ричардсона (Chris Richardson) и martinfowler.com Джеймса Льюиса (James Lewis) и Мартина Фаулера (Martin Fowler), но почти не было книг, доходчиво объясняющих, почему технический директор или руководитель группы разработчиков должен (или не должен) перейти к использованию микросервисов. На рынке явно наблюдался пробел. Чем больше я овладевал предметом, тем больше думал: «Почему бы мне самому не попробовать восполнить этот пробел?» Вскоре я начал обдумывать идею написания своей собственной книги.

Эта книга для вас?

Я писал эту книгу, ориентируясь на две аудитории специалистов. В первую входят студенты, дизайнеры и архитекторы с опытом разработки программного обеспечения и конструирования систем. Даже если вы знакомы с микросервисами и/или контейнерами, эта книга, вероятно, первая на вашем пути, полностью посвященная им. Ее цель – не только представить исчерпывающий обзор обозначенных тем, но и дать достаточный объем информации, чтобы помочь решить, стоит или не стоит использовать эти технологии в вашем конкретном случае. Те из вас,

кто уже имеет практический опыт работы с микросервисами и/или контейнерами, возможно, захотят перелистнуть части I и II и погрузиться прямо в часть III, демонстрирующую развернутый пример центра обслуживания, написанный в соответствии со стандартными методологиями создания сервис-ориентированных архитектур (Service-Oriented Architecture, SOA). Данный пример показывает, как преобразовать монолитную архитектуру приложения в архитектуру на основе микросервисов и как в общую картину вписываются контейнеры Docker. Я думаю, что это глубокое погружение окажется для вас достаточно полезным и интересным, чтобы вызвать желание продолжить самостоятельное исследование мира микросервисов и контейнеров.

Вторая аудитория моих потенциальных читателей – это люди, рассматривающие данную тему с точки зрения бизнеса (руководители или менеджеры проектов, которым интересно познакомиться с основами). Возможно, вы прочитали интригующую статью в блоге о микросервисах. Может быть, вы давно присматривались к этому решению, но не могли найти хорошую книгу, описывающую последовательность шагов, которые нужно выполнить. Вероятно, вы подслушали, как инженеры обсуждают контейнеры Docker, и теперь хотите узнать больше, чтобы принять участие в последующих обсуждениях. Независимо от причин, побудивших вас обратиться к этой книге, по сути, она представляет собой букварь, полный простых для понимания примеров и содержащий минимальное количество жаргона, и послужит идеальным руководством для любого менеджера, ищущего новые пути увеличения эффективности разработки и развертывания приложений.

Эта книга для всех, кто:

- стремится повысить эффективность разработки промышленного программного обеспечения в своей организации;
- предполагает перейти к использованию микросервисов и контейнеров Docker и хочет понять, чем они отличаются от SOA;
- желает получить представление о микросервисах и Docker, чтобы обрести новые навыки, пользующиеся спросом на рынке.

Проще говоря, эта книга для тех, кто хочет узнать больше о микросервисах и контейнерах Docker. Я надеюсь, что вы один из них!

Зарегистрируйте свою копию книги «Микросервисы и контейнеры Docker» на сайте InformIT, чтобы получить доступ к обновлениям и/или исправлениям по мере их появления. Для этого откройте в браузере страницу informit.com/register и выполните вход или создайте новую учетную запись. Введите код ISBN книги (9780134598383) и щелкните кнопку **Submit** (Отправить). На вкладке **Registered Products** (Зарегистрированные продукты) найдите ссылку **Access Bonus Content** (Дополнительные материалы) рядом с этим продуктом и перейдите по ней, чтобы получить доступ к любым имеющимся дополнительным материалам. Если вы хотите получать уведомления об эксклюзивных предложениях на новые издания или обновления, установите флажок, чтобы сообщить о своем желании.

Благодарности

Как человек, который на протяжении всей своей карьеры занимался технологиями, я не думал, что когда-нибудь напишу книгу. Я инженер, а не писатель. Поэтому, пока я вплотную не занялся этим, я не имел ни малейшего представления, как пишутся книги, и насколько это трудно. Скажем так: я понимал, что предстоит многое сделать, но не думал, что *настолько* много. Писать книгу было бы не так трудно, если бы я мог посвятить ей все свое рабочее время. Но я писал ее, продолжая трудиться на основной работе, поэтому временами мне казалось, что никогда не смогу закончить ее! И так бы и было, если бы не множество талантливых и великодушных людей, направлявших и поддерживавших меня на каждом шагу.

Спасибо прежде всего всей команде издательства Pearson, что приняли мое предложение и помогли пройти весь процесс издания книги. Особенно я хочу поблагодарить Кристофера Гузиковски (Christopher Guzikowski) за его помощь, веру, что я смогу это сделать, и за терпение. Также большое спасибо Майклу Терстону (Michael Thurston), который быстро и качественно исправлял мои ошибки.

Я не смог бы закончить эту книгу без помощи и поддержки многих друзей, особенно Ленина Лакшминараянана (Lenin Lakshminarayanan) и Анужа Сингха (Anuj Singh), которые потратили бесчисленное количество вечеров и выходных, помогая мне с примерами программного кода, особенно важными для этой книги. Огромное спасибо Джеральду Кантору (Gerald Cantor), который снова и снова читал рукопись и делился своим ценным мнением. Спасибо Рави Паписетти (Ravi Papisetti), Навазу Актеру (Nawaz Akther), Самиру Наиру (Sameer Nair) и Гурвиндеру Сингху (Gurvinder Singh) за ценные советы и предложения, а также Майклу Волману (Michael Wolman), который пропустил через себя каждое слово в этой книге.

Эта книга также была бы невозможна без воодушевляющих советов. Всякий раз, когда у меня возникали сомнения, я обращался к своим наставникам и неизменно получал ценные советы, которые сыграли важную роль в достижении мною этого момента в карьере. Особенно я хочу поблагодарить Грега Картера (Greg Carter), моего наставника на протяжении последних 12 лет, за его безоговорочную поддержку и помощь; Сунила Крипалани (Sunil Kripalani) за его доверие и за то, что подталкивал меня к инновациям и стремлению оказывать влияние; Антонио Нуччи (Antonio Nucci), настоящему провидцу, даже простые беседы с которым мотивируют меня на большие свершения.

Не в последнюю очередь я хочу поблагодарить мою семью за терпение, проявленное ко мне, пока я занимался этой благородной, но иногда очень нервной работой! Спасибо моим детям, Прабхлин (Prabhleen), Джашминдер (Jashminder) и Джаслин (Jasleen), которые провели без меня множество выходных и с пониманием отнеслись к моей занятости. Наконец, большое спасибо моей прекрасной жене Раман (Raman) за ее вдохновение, поддержку и доверие. Если бы не ее поддержка, эта книга осталась бы мечтой.

Большое спасибо всем вам!

Об авторе

Парминдер Сингх Кочер (Parminder Singh Kocher) родился и вырос в Индии. Вот уже более двух десятилетий он занимается созданием программных систем корпоративного класса. С 2005 г. работает в Cisco Systems, где в свое время руководил платформой Cisco Managed Services (CMS) и возглавлял несколько проектов программного обеспечения. В настоящее время занимает пост технического директора платформы Cisco Networking Academy и возглавляет инженерные группы, отвечающие за разработку платформы доступа следующего поколения в 180 странах. Помимо степеней бакалавра и магистра в области информатики, автор имеет степень магистра в области администрирования, полученную в Школе бизнеса имени Хэнкамера в Бэйлоре (Baylor Hankamer School of Business), и сертификат руководителя в области стратегии и инноваций, полученный в Школе менеджмента Слоана Массачусетского технологического института (MIT Sloan School of Management). Живет в городе Остин, расположенном в штате Техас, со своей женой и тремя детьми.

Часть I

МИКРОСЕРВИСЫ

Глава 1

Введение в микросервисы

Технологии меняются и влияют на развитие промышленности, которая, в свою очередь, предъявляет новые сложные требования к технологиям. Менее чем за 2 десятилетия мы перешли от эры коммутируемых модемов со скоростью 56 Кбит до 100-гигабитных сетей Ethernet. С увеличением скорости передачи данных выросли требования к скорости работы программного обеспечения, для разработки которого были созданы более совершенные и высокоуровневые языки программирования. Аналогично в области систем мы перешли от мейнфреймов к высокоскоростным серверам, а затем воплотили на этих серверах облачные технологии и технологии виртуализации. Теперь все чаще в разговорах специалистов стал встречаться термин «контейнеризация», обозначающий новую технологию, помогающую более эффективно использовать ресурсы.

Попутно появились новые парадигмы, такие как «модель – представление – контроллер» (Model – View – Controller, MVC), шаблоны интеграции корпоративных приложений (Enterprise Integration Patterns, EIP) и сервис-ориентированные архитектуры (Service-Oriented Architectures, SOA). В настоящее время в техническом мире активно обсуждаются архитектуры на основе микросервисов. Давайте попробуем выяснить, почему.

Что такое «микросервис»?

Микросервис – это независимый, автономный ресурс, спроектированный как отдельный выполняемый файл или процесс и взаимодействующий с другими микросервисами через стандартные, но легковесные межпроцессные связи, такие как протокол передачи гипертекста (HTTP), веб-службы RESTful (построенные на архитектуре репрезентативной передачи состояния – Representational State Transfer, REST), очереди сообщений и т. п. Уникальность микросервисов обусловлена тем, что каждый из них разрабатывается, тестируется, развертывается и масштабируется независимо от других микросервисов.

Идея использования микросервисов основана на лучших принципах разработки программного обеспечения, в том числе таких, как слабая взаимозависимость, высокая масштабируемость и ориентированность на службы.

Что подразумевается под словами «автономный ресурс»? А подразумевается под ними, что каждый микросервис выполняет ровно одну функцию, которая ведет себя одинаково для всех потребителей. Возьмем, к примеру, службу управления заказами, которая только обрабатывает заказы и больше ничего (даже уведомлений не отправляет). Но она может вызвать другой микросервис, отвечающий за отправку уведомлений об обработке. Такое разделение функций обеспечивает достаточную гибкость, т. к. каждый микросервис можно развивать, поддерживать, масштабировать, расширять и замещать независимо от других микросервисов.

Согласно этому определению, приложение на основе микросервисов – это просто группа из нескольких независимых и автономных микросервисов, каждый из которых реализует четко определенную функцию и для обеспечения общей функциональности приложения взаимодействует с другими микросервисами через четко определенные протоколы. Эту парадигму можно описать как архитектуру, в которой каждый микросервис выполняется в отдельном процессе.

Возможно, вам интересно узнать, чем приложения на основе микросервисов отличаются от монолитных приложений на основе сервис-ориентированной архитектуры (SOA). Разница в том, что в монолитном приложении, известном также как *монолитная реализация*, все службы упакованы в один большой выполняемый файл, или файл WAR.

Рассмотрим простой пример: веб-приложение калькулятора. В монолитном приложении все операции калькулятора (сложение, вычитание и т. д.) могут быть написаны как отдельные функции в программе, причем для выполнения своей операции одна функция может вызывать другую непосредственно. Все они действуют в пределах одного процесса и взаимодействуют друг с другом посредством стандартного механизма вызова подпрограмм. Примерная конструкция такой программы показана на рис. 1.1.

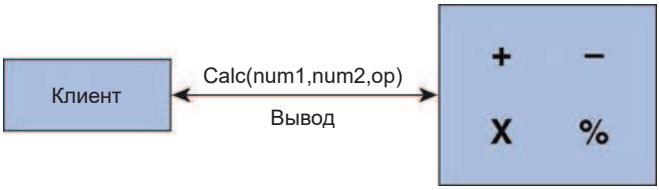


Рис. 1.1 ❖ Простая программа-калькулятор с монолитной архитектурой

Это очень простое приложение, реализация которого на основе микросервисов была бы излишеством. Однако только для того, чтобы понять суть, предположим, что разработчик, следуя парадигме микросервисов, сконструировал приложение калькулятора, реализовав каждую операцию в виде отдельной, автономной службы, как показано на рис. 1.2. В этом случае микросервисы будут вызывать друг друга по HTTP или другому протоколу. Если в какой-либо из функций в монолитном приложении возникнет ошибка (например, переполнение), это может привести к отказу всего приложения. Однако в случае с микросервисами проблема коснется только службы, где возникла ошибка, а остальные по-прежнему будут доступны для пользователей.

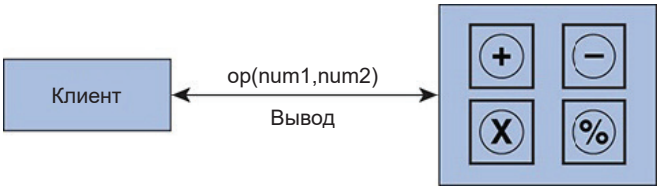


Рис. 1.2 ❖ Простая программа-калькулятор с архитектурой на основе микросервисов

Цель этого простого примера – подчеркнуть самое большое преимущество парадигмы микросервисов: она позволяет упростить реализацию сложного приложения, разделив его на более простые и автономные компоненты. Благодаря этой простоте можно, например, добавлять новые возможности, не влияя на другие службы.

Кроме того, каждый микросервис может развиваться и масштабироваться независимо. Например, предположим, что потребовалось добавить в приложение-калькулятор новую операцию, основанную на уже доступной функциональности: вычисление квадрата числа. Это очень простая операция и не требует изменения существующего кода. Мы создадим новый микросервис, который через стандартный документированный API вызовет микросервис «умножения» (см. рис. 1.3). Следовательно, нам потребуется написать, скомпилировать и развернуть только один микросервис, в отличие от монолитного приложения, требующего полной перекомпиляции и повторного развертывания с возможной приостановкой обслуживания клиентов.

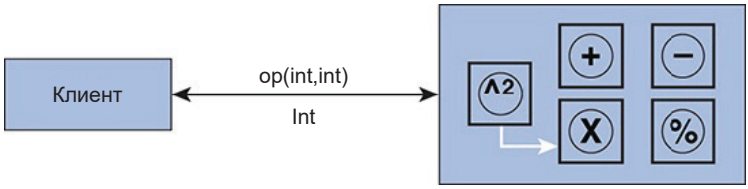


Рис. 1.3 ❖ В архитектуре на основе микросервисов легко добавить новую функцию «квадрат числа»

Также можно создавать микросервисы, которые вызываются только другими микросервисами и не используются непосредственно клиентским приложением. Например, как показано на рис. 1.4, клиент может вызвать лишь три микросервиса на уровне 1, тогда как первый микросервис на уровне 1 может вызвать два микросервиса на уровнях 2 и 3, как показано стрелками. Микросервисы, подобные этим двум, часто называют *вспомогательными*.

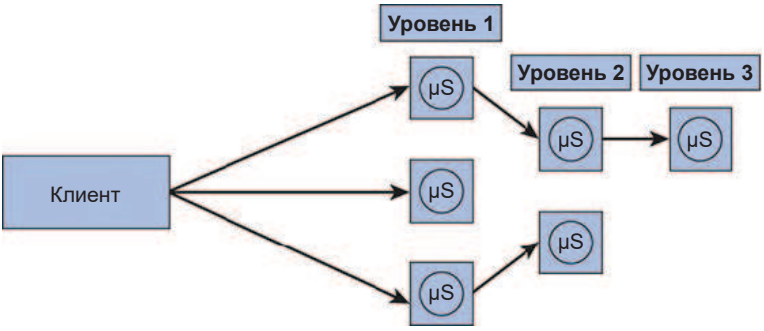


Рис. 1.4 ❖ Микросервис, вызывающий другие микросервисы[1]

Идея микросервисов не нова, но в последнее время она набирает популярность, так как избавляет от проблем, характерных для монолитных приложений.

Давайте рассмотрим другой пример и обсудим эти проблемы. Представьте систему электронной коммерции и ее компоненты на верхнем уровне, как показано на рис. 1.5.

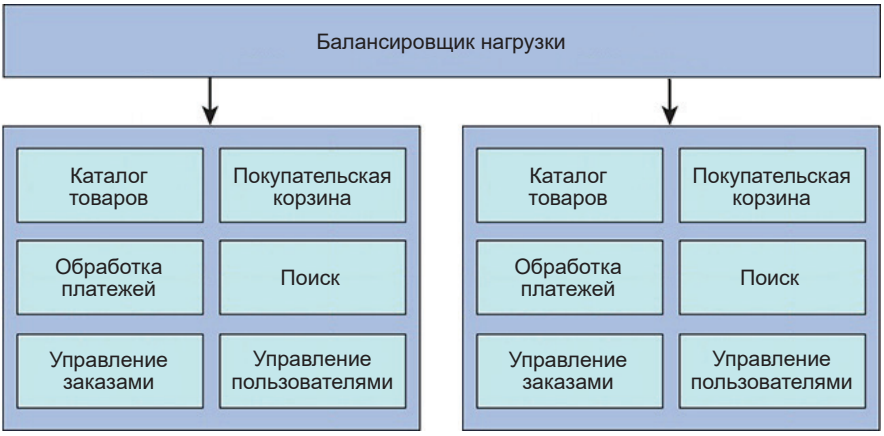


Рис. 1.5 ❖ Основные компоненты монолитной системы электронной коммерции

Эта система может прекрасно подходить для малых и средних компаний. Отдел технического обеспечения создает и развертывает в промышленном окружении единственный пакет, а горизонтальная масштабируемость системы легко обеспечивается развертыванием нескольких копий приложения и размещением балансировщика нагрузки перед ними. С ростом бизнеса растут и требования к возможностям, что влечет расширение кода, увеличение численности специалистов и, в свою очередь, сложность выпуска новых версий, развертывания и сопровождения приложения. Со временем приложение становится все более сложным, что затрудняет разграничение ответственности за код и функциональные

[1] Здесь и далее μS – микросервис. – *Прим. перев.*

возможности между разработчиками. В этот момент, как правило, все начинает рушиться, и организация сталкивается со следующими проблемами:

- низкой производительностью;
- плохой масштабируемостью;
- долгими циклами регрессионного тестирования;
- долгими циклами обновления и повторного развертывания, влекущими невозможность быстро внедрить мелкие исправления и улучшения;
- незапланированными простоями;
- возможными простоями на время, пока производится обновление;
- невозможностью внедрения новых технологий и языков программирования;
- невозможностью выборочного масштабирования необходимых компонентов или функций.

Одними из множества последствий, вытекающих из этих проблем, которые обычно остаются незамеченными, являются разочарование, испытываемое инженерами, и нарастание конфликтных ситуаций в коллективе.

В таких ситуациях очень может пригодиться парадигма микросервисов. Применение этой парадигмы оправдано только в отношении больших монолитных приложений, поскольку влечет за собой затраты, которые могут оказаться нецелесообразными, если приложение невелико или поддерживает малый бизнес. Чтобы на этом этапе зрелости разложить монолитное приложение на микросервисы, может потребоваться инвестировать кругленькую сумму. Поэтому организации обычно начинают добавлять новые возможности, реализуя их как микросервисы, а затем, уже получая отдачу от инвестиций, постепенно преобразовывают старое приложение.

Представьте, что нам нужно обновить компонент «покупательская корзина» из предыдущего примера. В зависимости от архитектуры старого программного обеспечения для этого может потребоваться не только добавление или обновление кода, но и выполнение регрессионного тестирования всего кода или только той его части, которая так или иначе связана с покупательской корзиной. Также потребуется перекомпилировать, протестировать и развернуть приложение целиком, что может привести к простою или замедлить работу приложения. Кроме того, разработчик может посчитать, что конкретную функциональность проще и эффективнее было бы реализовать на каком-то новом языке, таком как Scala. Это его желание, вероятно, останется невыполнимым, если не выделить средства, чтобы переписать все приложение на этом новом языке. Фактически разработчик окажется привязанным к выбору своих предшественников, который, возможно, был правильным в то время, но в данный момент уже не является оптимальным.

Давайте посмотрим, как микросервисы могут помочь в такой ситуации. Как уже обсуждалось, выделим компоненты монолитного приложения в отдельные микросервисы, как показано на рис. 1.6.

Эти микросервисы развертываются по отдельности, и каждый реализует только одну функцию. Если понадобится изменить микросервис, реализующий покупательскую корзину, мы должны будем изменить только код этого микросервиса. Сделать это будет намного проще, так же как проще будет провести тестирование и развертывание. Микросервисы не только решают проблемы, характерные для монолитных приложений, но и предлагают ряд преимуществ, которые способствуют внедрению технологии непрерывного развертывания.

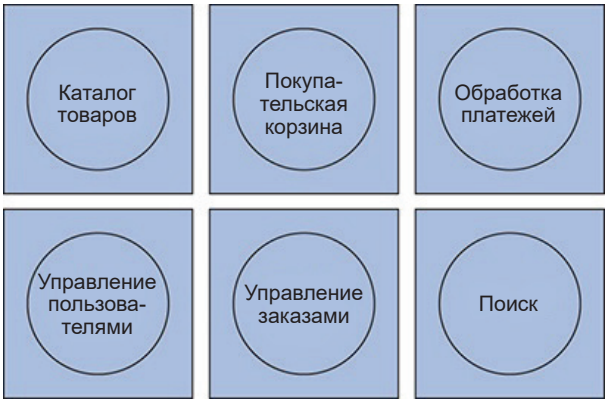


Рис. 1.6 ❖ Компоненты системы электронной коммерции, выделенные в отдельные микросервисы

Модульная архитектура

Согласно отчету The Standish Group за 2016 г. («CHAOS Report 2016»), только 29 % программных проектов во всей отрасли уложились в отведенное время и смету. Это означает, что 71 % проектов потерпел неудачу или их успех оказался весьма спорным. Неудачи в основном были обусловлены проблемами качества, незавершенностью, перерасходом бюджета и т. д. Вследствие этого было введено много новых практик и стандартов управления проектами, которым должны были следовать организации, занимающиеся разработкой программного обеспечения (например, стандартов разработки программного обеспечения IEEE, стандартов тестирования программного обеспечения). Основная цель этих стандартов – управление сложностью с применением передовых практик. В результате организации, принявшие на вооружение эти стандарты, улучшили шансы на завершение проектов и увеличили срок службы приложений.

Средний срок службы программных приложений или платформ составляет от 4 до 6 лет, после чего они устаревают по различным причинам. В числе причин могут быть изменение требований с течением времени, невозможность масштабирования из-за устаревшей архитектуры, устаревание технологий и т. д. Индустрия постоянно развивается и все время требует создания приложений нового поколения, что означает необходимость переписывать программное обеспечение с использованием новейших технологий, архитектур и передовых практик. Но в какой-то момент необходимо задать вопрос: так ли необходимо изменять каждый компонент, т. е. весь пакет? Оказывается, это требуется не всегда. Некоторые компоненты или части действительно можно усовершенствовать, задействовав новые технологии, но обычно этот вариант не подходит, потому что архитектура не обеспечивает модульности и не позволяет заменять отдельные компоненты или части программного обеспечения новым кодом.

Мы разрабатываем монолитные приложения, а значит, должны следовать стандартам, помогающим преодолеть сложности. Но если устранить саму сложность с помощью парадигмы микросервисов, мы получим модульную архитектуру, значительно увеличивающую срок службы программного обеспечения, и сможем

уменьшить нашу зависимость от большого количества стандартов, избавиться от громоздкого процесса разработки и сэкономить время, а значит, ускорить цикл создания программного обеспечения.

Помимо эффективности процесса, модульная архитектура также позволит добиться существенной экономии в будущем, когда потребуется обновить платформу. Вместо того чтобы все начать сначала, мы сможем хирургическим путем удалить устаревшие микросервисы и заменить их новыми, реализованными с использованием более совершенных технологий и архитектур. Это одно из ключевых долгосрочных преимуществ парадигмы микросервисов, отличающих ее от других парадигм. Но даже простое увеличение модульности делает внедрение подхода на основе микросервисов достойным инвестиций.

Другие преимущества микросервисов

Помимо достоинств, перечисленных выше, микросервисовы могут предложить следующие выгоды:

- **простота.** Каждый микросервис выполняет только одну четко определенную функцию, поэтому требуется меньше кода, меньше зависимостей от другого кода и уменьшается вероятность ошибок;
- **масштабируемость.** Для масштабирования монолитного приложения его необходимо развернуть на нескольких серверах и настроить балансировщик нагрузки. Невозможно масштабировать только часть приложения. Здесь действует принцип «все или ничего». С микросервисами можно масштабировать только компоненты, подвергающиеся высокой нагрузке, как показано на рис. 1.7. Возможность дифференцированной масштабируемости является очень простой и важной особенностью микросервисов;

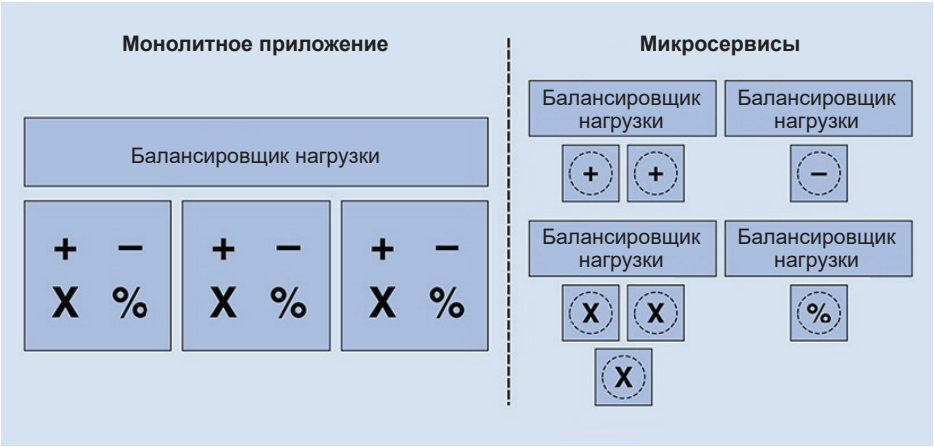


Рис. 1.7 ❖ Сравнение масштабируемости

- **непрерывное развертывание.** Благодаря меньшему количеству взаимозависимостей в коде и более быстрому циклу разработки парадигма микросервисов поддерживает культуру непрерывного развертывания и интегра-

ции разработки и эксплуатации (DevOps) и фактически подталкивает к ее использованию;

- **больше свободы и меньше зависимости.** Микросервисы по определению автономны и независимы. Команда разработчиков может сосредоточиться на своем микросервисе и свободно расширять его возможности, не опасаясь нарушить работу другого микросервиса, пока они гарантируют неизменность интерфейса или реализуют новый интерфейс, обратно совместимый с прежним;
- **изоляция отказов.** Изоляция отказов – это явление, когда отказ в одной части системы не приводит к отказу всей системы, т. е. отказы оказываются изолированными от системы. В монолитном приложении сбой в любой его части приведет к сбою всего приложения, потому что оно представляет собой единый процесс. В случае с микросервисами дело обстоит иначе: сбой в одном микросервисе может привести к отказу этого микросервиса, но вовсе не обязательно приведет к отказу всей программы, потому что отказавший микросервис выполняется в отдельном процессе. Например, в системе электронной коммерции, основанной на архитектуре микросервисов, в случае сбоя микросервиса «отзывы о продукте» пользователи по-прежнему смогут просматривать список товаров, имеющихся в наличии, выбирать товары для покупки, просматривать содержимое покупательской корзины и размещать заказы. Единственное, чего они не смогут, – это увидеть отзывы, пока не будет исправлен микросервис просмотра отзывов. Если бы приложение было монолитным, ошибка в компоненте, отвечающем за просмотр отзывов, могла бы прервать работу всего приложения;
- **разделение и децентрализация данных.** В отличие от монолитных приложений, где все данные обычно хранятся вместе в центральной базе данных, микросервисы дают возможность разделить данные. Каждый микросервис может владеть только своими данными и не делиться ими с другими микросервисами;
- **широта выбора.** В отличие от монолитного приложения, где все компоненты используют единую базу данных, платформу и должны быть написаны на одном языке программирования, микросервисы дают возможность использовать инструменты, лучше подходящие для каждого конкретного случая. Один микросервис может использовать Oracle и ОС Linux, а другой – NoSQL и Microsoft Windows. Больше нет необходимости связывать себя с определенными стеками технологий.

Недостатки микросервисов

Ничто не дается бесплатно, и преимущества микросервисов тоже имеют свою цену. Двигаясь в сторону микросервисов, мы должны знать, какие проблемы связаны с этой архитектурой. Не волнуйтесь! В следующей части данной книги вы узнаете, как преодолеть эти проблемы с использованием определенных систем и приложений. А теперь перечислим некоторые проблемы, характерные для микросервисов:

- **сложность поиска и устранения неисправностей.** Микросервисы предлагают свои возможности посредством механизма взаимодействий между

микросервисами, что увеличивает число потенциальных точек отказа. Это делает ответы на следующие вопросы более сложными:
  - Насколько нормально работает моя система в данный момент?
  - Если конечный пользователь сообщает о такой проблеме, как низкая производительность или длительные периоды ожидания, с чего начать устранение неполадок?
  - Отследить путь обработки запроса в монолитном приложении намного проще. Но в приложении, состоящем из микросервисов, каждый запрос может быть разбит на несколько запросов, обрабатываемых разными микросервисами. Поиск и устранение неполадок в этом случае могут стать немного сложнее;
- **увеличенные задержки.** Внутрипроцессные взаимодействия (как и в монолитных приложениях) выполняются намного быстрее межпроцессных (как в случае с микросервисами);
- **сложность сопровождения.** Когда приложение состоит из сотен или даже тысяч микросервисов, группам оперативного сопровождения приходится преодолевать сложности, связанные с настройкой инфраструктуры, развертыванием, мониторингом, резервным копированием и управлением. Можно даже сказать, что сложности монолитной архитектуры переносятся на системную сторону микросервисов. Тем не менее эти сложности можно преодолеть с помощью высокого уровня автоматизации;
- **управление версиями.** Из-за того что приложение может состоять из тысяч микросервисов, управление версиями становится немного сложнее. Для его осуществления требуется использовать более совершенные системы управления версиями.

Глава 2

Переход к микросервисам

В главе 1 «Введение в микросервисы» мы сравнили микросервисы с монолитными архитектурами. Теперь, понимая разницу между ними, вы, вероятно, пытаетесь ответить на вопрос, подходят ли микросервисы для вашей команды? Если вам уже приходилось преодолевать сложности, свойственные монолитной архитектуре, или вы планируете создать монолитную систему, тогда обязательно подумайте о возможности использования микросервисов. В противном случае нет причин переключаться на эту архитектуру, т. к. она не подходит для организации служб малого и среднего размеров. Каждый микросервис несет бремя дополнительной работы в масштабах, несоизмеримых с монолитными архитектурами. Организация API, мониторинг процессов, балансировка нагрузки для обеспечения высокой производительности/доступности и т. д. – все это необходимо предусмотреть для каждого микросервиса, а не только для всего приложения. Фактически вы меняете сложности разработки монолитного приложения на сложности сопровождения микросервисов. Если сложности разработки отсутствуют в вашей системе, то вы без необходимости добавите сложности сопровождения. Поэтому будьте очень осторожны, выбирая между этими двумя парадигмами, иначе не избежать неприятных последствий.

В этой главе изложены критерии пригодности (и непригодности) архитектуры на основе микросервисов для различных применений. Как правило, руководители и менеджеры ищут потенциальные выгоды или отдачу от инвестиций. Поэтому здесь мы кратко обсудим микросервисы с этой точки зрения и проведем простое моделирование затрат и выгод.

Предпосылки и условия

Вот основные предпосылки для переноса монолитных приложений на архитектуру микросервисов:

- сложный и продолжительный процесс развертывания;
- большой объем кода, с которым с трудом справляются интегрированные среды разработки;
- неоднородные требования к масштабированию (одни компоненты требуют большей масштабируемости, чем другие);
- высокая стоимость разработки, тестирования и развертывания;
- деградация качества кода из-за слишком большого количества внутренних зависимостей;
- ошибка в одном компоненте может вызвать отказ всего приложения.

Тщательно проанализируйте и задокументируйте эти предпосылки. Затем попробуйте определить, могут ли повысить ценность текущего приложения некоторые из следующих возможностей:

- организация служб по бизнес-функциям;
- развертывание служб по отдельности;
- асинхронные взаимодействия;
- замена компонентов платформы, языков программирования и/или баз данных для различных служб с целью увеличения производительности;
- непрерывное развертывание и непрерывная интеграция;
- каждая группа разработчиков хорошо знает и понимает свою часть бизнес-процесса, например такую, как управление заказами или поддержка покупательской корзины.

Осмысление этих условий поможет вам понять, что вы имеете и есть ли смысл переходить к парадигме микросервисов. После того как вы примете решение взять на вооружение парадигму микросервисов, пути назад не будет. Поэтому, прежде чем что-то решать, вы также должны узнать об уникальных потребностях, связанных с таким переходом:

- **изменение культуры.** Переход к парадигме микросервисов должен сопровождаться изменением ролей групп инженеров – переходом от функциональных ролей к бизнес-ориентированным ролям с общими целями и задачами. Под этим подразумевается создание групп, объединяющих менеджеров, разработчиков, тестировщиков и специалистов по сопровождению и эксплуатации, которые возьмут на себя ответственность за конкретные микросервисы. Потребуются инвестиции в воспитание новых талантов и обучение существующего персонала, а также в новые системы, инструменты и программное обеспечение. Кроме того, чтобы добиться успеха, потребуется автоматизировать весь жизненный цикл программного обеспечения;
- **организация эксплуатационного процесса.** Парадигма микросервисов требует изменения эксплуатационного процесса и структуры организации. Структура должна иметь более кросс-функциональный характер. Каждая группа должна отвечать за развертывание, поддержку, обновление и эксплуатацию своих микросервисов. Существующие процессы тестирования и развертывания монолитного приложения придется разбить на несколько обширных процессов сопровождения сотен или тысяч автономных микросервисов и поддержки взаимосвязей между ними.

Кривая обучения в организации

Для существующих инженерных и эксплуатационных групп, имеющих дело с разными аспектами разработки и поддержки монолитных приложений, должна использоваться совершенно иная кривая обучения. Она определяется такими новыми аспектами приложений на основе микросервисов, как:

- **автономные микросервисы.** Монолитные приложения существуют как один большой блок, развертываемый на нескольких серверах для масштабируемости. В случае с микросервисами таких блоков, пусть и меньших по размеру, может быть несколько сотен или даже тысяч, и все они требуют равного внимания;

- **обнаружение микросервисов.** Чем больше микросервисов, тем выше сложность. Например, вам нужно подумать о том, как будут обнаруживаться микросервисы, т. е. о том, как и где организовать реестр микросервисов. Среди других сложностей можно назвать реализацию масштабируемости по мере необходимости и управление версиями, а также удаление служб, ставших ненужными. Для решения этих задач с успехом можно использовать различные приложения, такие как Consul, Apache ZooKeeper и другие сторонние продукты. Однако эти проблемы создают необходимость найма новых или переподготовки существующих сотрудников, что может потребовать существенных финансовых вложений;
- **связь между микросервисами.** Определяя способ взаимодействий между службами и внешним миром, необходимо учесть ожидания клиента относительно времени ответа, задержки, количества повторных попыток и т. д. Также следует определить, что должно происходить, когда эти ожидания или соглашения об уровне обслуживания (Service Level Agreement, SLA) не выполняются. Возможно, потребуется реализовать стандартный интерфейс для связи;
- **тестирование микросервисов.** Методы и принципы тестирования монолитных приложений неприменимы к приложениям на основе микросервисов. Протестировать каждую отдельный микросервис достаточно легко, но задача состоит в том, чтобы протестировать все приложение, состоящее из сотен или тысяч микросервисов. Это требует работы с большим количеством компонентов, поэтому особую важность приобретает интеграционное тестирование. Некоторые сложности, связанные с тестированием, можно решить путем создания автоматизации тестирования;
- **масштабирование микросервисов.** Масштабирование микросервисов реализуется проще и эффективнее. Их можно масштабировать по отдельности и по мере необходимости. Но это удобство имеет свою цену. Во-первых, при разработке микросервисов должны учитываться потребности масштабирования, т. е. необходимо знать требования к использованию для каждого микросервиса. Во-вторых, масштабирование должно быть автоматизировано, что требует изучения и внедрения таких фреймворков, как Mesos и Marathon. Мы подробно обсудим их в последующих главах;
- **обновление микросервисов.** На первый взгляд, задача обновления микросервисов может показаться простой, потому что каждый из них самодостаточен и, следовательно, его обновление не должно вызывать нарушений в работе всего приложения. Обновление действительно может выполняться просто, если новая версия микросервиса включает лишь простые изменения, не влияющие на внешний мир. Когда изменения влияют на другие зависимые службы, при обновлении могут возникнуть некоторые сложности. Поэтому вы должны убедиться в том, что другие службы готовы использовать новые функции, или в том, что новая версия службы обратно совместима с предыдущей;
- **безопасность микросервисов.** Безопасность всегда важна, и, учитывая современные угрозы кибербезопасности, особенно внимательно нужно относиться к вопросам безопасности во время разработки, таким как: безопасность взаимодействий между микросервисами, безопасность взаимодействий между

клиентами и микросервисами, безопасная передача данных и безопасное их хранение. Некоторые проблемы безопасности помогают решать такие стандарты, как OAuth и OpenID, но другие необходимо тщательно обдумать, чтобы найти сбалансированное решение, обеспечивающее достаточно высокую безопасность и простоту использования;

- **управление микросервисами.** Независимо от архитектуры программного обеспечения или парадигмы, управление приложениями остается ключевым условием общего успеха в эксплуатации и поддержке. Управлять микросервисами сложнее, чем монолитным приложением. Существующие инструменты и методы мониторинга и управления могут оказаться малополезными. Вместо нескольких серверов и приложений нам приходится иметь дело с более сложными системами и технологиями, такими как контейнеры. Поэтому может очень пригодиться идея одного окна (т. е. единого интерфейса) для настройки, мониторинга и диагностики;
- **мониторинг в микросервисах.** При обслуживании сотен и тысяч микросервисов, разбросанных по распределенным системам, приходится иметь дело с большим количеством движущихся частей. Поэтому необходимо контролировать и саму инфраструктуру (процессор, память, производительность ввода/вывода), и приложения (файлы журналов приложений, производительность вызовов API). Данные, извлекаемые на этих уровнях мониторинга, должны быть доступными для групп сопровождения и эксплуатации, чтобы они могли обеспечить эффективное обслуживание;
- **настройка микросервисов.** Любая служба имеет различные параметры конфигурации, предусмотренные разработчиками, которые обеспечивают гибкость эксплуатации и упрощают адаптацию служб под изменяющиеся условия. К таким настройкам относятся параметры, управляющие кешированием, масштабированием, количеством потоков выполнения, флагами поддерживаемых функций, соединениями с базами данных и т. д. Управление этими аспектами для тысяч служб может быть весьма трудоемкой задачей. Существует множество инструментов для решения некоторых из этих проблем, поэтому, чтобы получить простой общий интерфейс, необходимо выбрать правильную комбинацию инструментов;
- **обработка сбоев в микросервисах.** В случае сбоя в микросервисе инструменты мониторинга, приведенные выше в этом списке, могут оказать определенную помощь, но в целом система должна предусматривать возможность сбоя. Каждый микросервис должен строиться так, чтобы сбой в зависимой службе не вызывал проблем в его собственной работе, не говоря уже о выходе из строя всей системы. Главной целью должна стать реализация возможности самовосстановления системы.

В свете всей этой информации организация должна быть полностью подготовлена к таким изменениям и иметь необходимые ресурсы для успешного перехода. Решение должно приниматься только после оценки и взвешивания всех этих пунктов. Рекомендуется также составить список недостающих ресурсов, чтобы проще было понять, какой объем инвестиций потребуется для перехода к парадигме микросервисов.

Аргументация перехода к микросервисам

Учитывая все проблемы, обсуждавшиеся выше, иногда трудно аргументировать переход на использование микросервисов. Можно было бы подумать: если создавать и сопровождать приложения на основе микросервисов сложнее, зачем тогда тратить на это силы и время? Конечно, реализация парадигмы микросервисов – сложная задача, и в первое время могут потребоваться весьма значительные усилия по подготовке персонала и изменению культуры производства, однако долгосрочные выгоды не только перевесят первоначальные инвестиции, но и помогут получить экономию и другие преимущества в перспективе. Вам нужно лишь выполнить очень простой анализ, который поможет понять или подобрать аргументы для организации.

Средний срок службы программной платформы с монолитной архитектурой обычно составляет 4–5 лет и зависит от таких факторов, как:

- изменение потребностей клиентов, приводящее к устареванию существующей функциональности;
- появление новых потребностей бизнеса;
- недостаточная гибкость для корректировки или изменения существующей архитектуры;
- недостаточная масштабируемость;
- устаревшие технологии;
- снижение производительности, обусловленное использованием устаревших систем и увеличением трафика с течением времени.

Столкнувшись с этими факторами, организации начинают искать новые технологии и, как правило, приходят к выводу о необходимости инвестировать в новую платформу следующего поколения. Это называется *циклом обновления платформы*. С точки зрения бизнеса, все необходимые изменения являются оправданными, потому что ожидания клиентов и модели доставки меняются с течением времени. Основное беспокойство в организациях вызывает необходимость больших инвестиций в каждый цикл, причем как с финансовой точки зрения, так и с точки зрения времени. Это беспокойство вполне объяснимо, потому что такие инвестиции влияют на прибыль. И в этом отношении внедрение парадигмы микросервисов выглядит очень привлекательным. Давайте проведем высокоуровневый анализ, чтобы доказать это.

Составляющие затрат

Рассмотрим на гипотетическом примере составляющие затрат, возникающих в течение жизненного цикла монолитной платформы:

- **затраты на создание** – стоимость создания программной платформы с нуля, включая все этапы цикла разработки, такие как анализ, проектирование, разработка, тестирование и выпуск. Эти затраты самые большие по объему в цикле. Обозначим их как $М_{СТВ}$;
- **затраты на сопровождение** – нормальный уход за программным обеспечением и его поддержка (например, применение исправлений на уровне операционной системы и поддержка работоспособности инфраструктуры). Обозначим их как $М_{СТМ}$;

- **затраты на изменение/обновление** – стоимость добавления новых функций, исправления ошибок, повторного тестирования, регрессионного тестирования и выпуска обновленных версий в течение жизненного цикла проекта. Обозначим их как M_{CTU};
- **затраты на масштабирование** – стоимость масштабирования платформы для поддержки короткого времени отклика и высокой производительности по мере увеличения количества пользователей. Обозначим их как M_{CTS};
- **время выхода на рынок** – время, затраченное на создание или обновление программной платформы. Время между анализом и выпуском платформы или обновления. Обозначим эту величину как M_{TTM}.

Для сравнения обозначим аналогичные затраты на то же программное обеспечение, но созданное на основе парадигмы микросервисов:

- **затраты на создание** – S_{CTB};
- **затраты на сопровождение** – S_{CTM};
- **затраты на изменение/обновление** – S_{CTU};
- **затраты на масштабирование** – S_{CTS};
- **время выхода на рынок** – S_{TTM}.

Итак, какая из архитектур более эффективна с экономической точки зрения? Сравним монолитную архитектуру с архитектурой микросервисов по каждому из предыдущих параметров:

- **затраты на создание: $M_{CTB} < S_{CTB}$.** Если у вас уже есть готовое приложение, вы должны учесть все новые инвестиции, такие как обучение персонала, изменение культуры производства, наем новых специалистов и обновление инструментов и систем. С учетом всех этих аспектов стоимость создания приложения на основе микросервисов может оказаться очень высокой, особенно по сравнению со стоимостью создания монолитного приложения. Но если вы начинаете разработку совершенно нового проекта, то затраты могут не сильно отличаться, что зависит от текущих организационных возможностей. Учитывая системные требования и необходимость приобретения инструментов, стоимость создания монолитного приложения может быть немного ниже;
- **затраты на сопровождение: $M_{CTM} > S_{CTM}$.** Стоимость профилактического обслуживания оборудования и применения исправлений при определенных условиях можно оценить стоимостью простоя приложения. Существует множество открытых технологий, автоматизирующих все этапы (от развертывания до локализации сбоев). Многие из них мы рассмотрим далее в книге. Например, развертывание микросервисов в контейнерах и переход к интеграции разработки и эксплуатации (DevOps) позволяют создавать новые контейнеры для микросервисов, экономить время и обеспечивать эффективное использование ресурсов, снижая общую стоимость обслуживания при одновременном уменьшении вероятности простоя;
- **затраты на изменение/обновление: $M_{CTU} > S_{CTU}$.** Одним из ключевых преимуществ парадигмы микросервисов является простота обновления существующей и добавления новой функциональности (микросервисов). В монолитном проекте подобное изменение может потребовать пересборки всего приложения. На обновление, сборку, тестирование и развертывание микросервиса, выполняющего только одну функцию, требуется меньше времени и усилий, чем на обновление монолитного приложения, которое может занять несколько

часов. Кроме того, на тестирование и развертывание микросервиса требуется меньше времени и сил, что в некоторых случаях не вызывает простоев;

- **затраты на масштабирование: $M_{CTS} > S_{CTS}$.** Масштабирование только по мере необходимости и только необходимых функций является ключевым преимуществом микросервиса по сравнению с монолитным приложением, требующим запуска еще одного полновесного экземпляра программы. В отличие от монолитного приложения, можно масштабировать только необходимые компоненты (микросервисы), автоматически развертывая новые контейнеры со службами и останавливая их при снижении спроса. Такой подход экономит не только усилия, но и аппаратно-программные ресурсы, как показано на рис. 2.1;

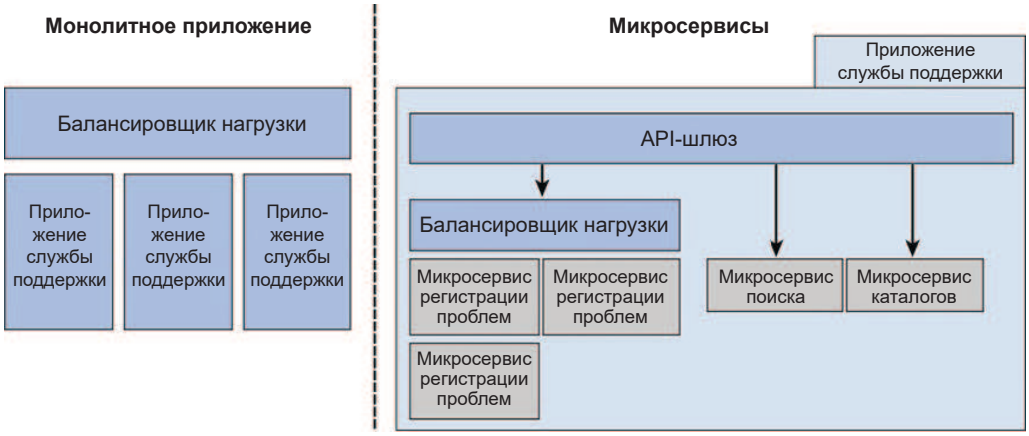


Рис. 2.1 ❖ Сравнение способности к масштабированию

- **время выхода на рынок: $M_{TTM} > S_{TTM}$.** Есть два ключевых аспекта, выгодно отличающих микросервисы от монолитных приложений по времени выхода на рынок. Во-первых, добавление новой службы и ее внедрение в большинстве случаев происходят намного быстрее, чем обновление монолитного приложения. Во-вторых, учитывая модульную организацию архитектуры микросервисов, важно помнить, что она, наряду с контейнерами, позволяет использовать другой метод доставки программного обеспечения, который называется интеграцией разработки и эксплуатации (DevOps). Фактически микросервисы и контейнеры являются ключом к успеху DevOps. Интеграция разработки и эксплуатации предлагает четыре ключевые составляющие успеха программной платформы:
  - скорость;
  - стабильность;
  - производительность;
  - сотрудничество.

  DevOps обеспечивает высокую гибкость и сокращает время выхода на рынок. Организации стремятся максимально использовать эти преимущества, чтобы сохранить конкурентное превосходство. Короткое время выхода на рынок само по себе может оказаться наиболее весомым аргументом для перехода к парадигме микросервисов;

- **будущие циклы обновления.** Как уже отмечалось, приложение с монолитной архитектурой имеет конечный средний срок службы. После того как этот срок заканчивается, организация обычно начинает новый цикл, который по стоимости сопоставим с начальными затратами $М_{СТВ}$. *Микросервисы фактически ломают всю эту концепцию циклов*, потому что они обеспечивают:
  - гибкую возможность добавления или удаления микросервисов в соответствии с бизнес-требованиями;
  - автоматическое масштабирование системы добавлением и удалением экземпляров служб за балансировщиком нагрузки;
  - возможность замены устаревших технологий для каждого микросервиса в отдельности, что способствует снижению затрат.

Благодаря своей гибкости парадигма микросервисов позволяет реализовывать новые бизнес-требования по мере необходимости и обеспечивать быстрый отклик на изменения рынка. Кроме того, отпадает необходимость менять всю платформу новым поколениям каждые несколько лет.

Суммируя вышесказанное, приходим к выводу, что чистые затраты на реализацию архитектуры микросервисов, несомненно, будут намного ниже затрат на реализацию монолитной архитектуры. Динамика затрат с течением времени выглядит примерно так, как показано на рис. 2.2, при этом чистая стоимость микросервисов значительно ниже. Место точки пересечения двух графиков затрат в действительности зависит от типа и величины проекта.

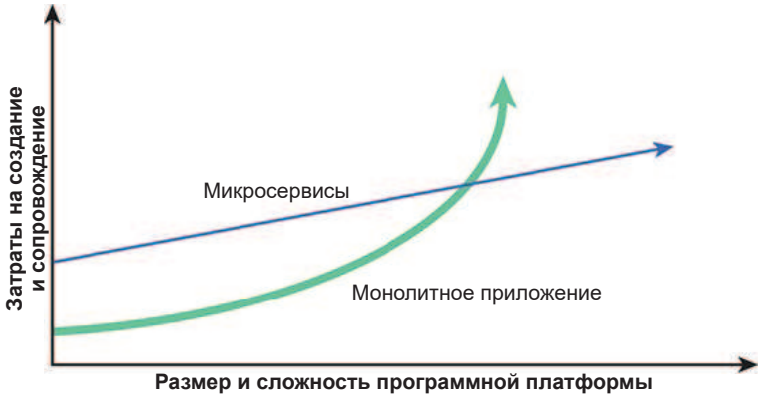


Рис. 2.2 ❖ Упрощенное сравнение затрат

Завершая аргументацию, подчеркну еще раз, что, с точки зрения экономии средств, микросервисы выглядят предпочтительнее, но эта экономия достигается только с течением времени. Парадигма микросервисов требует значительных начальных финансовых и организационных затрат. Как отмечалось выше, для большинства организаций даже простое ускорение выхода на рынок может перевесить все другие выгоды.

Принимая решение о переходе к архитектуре на основе микросервисов, организация должна учитывать конкурентные преимущества, кривую обучения, а также проанализировать затраты и выгоды.

Глава 3

Межпроцессные взаимодействия

В монолитных архитектурах взаимодействия между компонентами происходят посредством вызовов функций, методов или модулей и в большинстве случаев реализуются очень просто. В архитектурах на основе микросервисов взаимодействия имеют намного более сложную форму. Но, несмотря на то что методы управления межпроцессными взаимодействиями в архитектуре микросервисов не являются ключевой темой этой книги, в этой главе мы рассмотрим некоторые приемы, хорошо зарекомендовавшие себя на практике.

Типы взаимодействий

Обычно микросервисы предоставляют свои функциональные возможности через программные интерфейсы (API) или веб-службы. Для доступа к веб-службам в сети используются два основных шаблона организации взаимодействий:

- **синхронные взаимодействия** – взаимодействия, когда клиент ждет ответа службы, блокируя дальнейшее свое выполнение (действуя по схеме «запрос/ответ»);
- **асинхронные взаимодействия**, когда клиент продолжает выполнение, не дожидаясь ответа. В этом случае клиент посылает службе запрос, продолжает работу и обрабатывает ответ, когда он будет получен. Примерами могут служить шаблоны «публикация/подписка» и «асинхронный запрос/ответ».

Как обсуждалось в главе 1 «Введение в микросервисы», для микросервисов предпочтительнее использовать асинхронные взаимодействия. Представьте, что получится, если использовать синхронные взаимодействия. Клиент будет блокироваться до получения ответа от службы перед продолжением работы. Что произойдет, если служба аварийно завершится из-за ошибки? Этот подход плохо масштабируется, и мы потеряем большинство преимуществ микросервисов. Поэтому асинхронные взаимодействия являются лучшей альтернативой.

При использовании асинхронных взаимодействий клиент посылает запрос микросервису и продолжает выполнять другую работу, одновременно запуская дополнительный поток выполнения, ожидающий ответов и обрабатывающий их по мере поступления. Проблемы, возникающие внутри вызываемого микросервиса, в этом случае никак не скажутся на клиенте. В результате слабая зависимость от микросервисов улучшает масштабируемость.

Другой подход заключается в использовании шаблона «публикация/подписка», когда издатель публикует сообщения, например, посылая их в очередь сообщений, такую как Kafka. Подписчики регистрируются для получения интересующих их сообщений и извлекают их из очереди для обработки, игнорируя остальные. После обработки они могут опубликовать результаты, которые, в свою очередь, могут извлекаться исходным издателем в зависимости от используемых шаблонов обмена сообщениями.

Подготовка к разработке веб-служб

В общем, готовясь к созданию веб-службы, разработчик должен выбрать:

1) **протокол.** Общеизвестно, что золотым стандартом для веб-служб является протокол HTTP. Этот же протокол используется веб-браузерами, поэтому можно утверждать, что он выдержал испытание временем. Самое большое преимущество протокола HTTP заключается в том, что он легковесный и основан на простой модели «запрос/ответ», согласно которой клиент формирует и посылает HTTP-запрос, а сервер выполняет необходимые действия и возвращает HTTP-ответ;
2) **стандарт веб-служб.** На выбор есть три основных варианта:
   – *RESTful* – широко распространенный и рекомендуемый;
   – *SOAP* – довольно громоздкий и требующий поддержки на стороне клиента и сервера;
   – *Data* – открытый протокол, необходимый для реализации и использования RESTful API.

   Стандарт RESTful основан на модели «запрос/ответ». Он намного более легковесный, чем SOAP, поэтому выглядит предпочтительнее. Кроме того, RESTful-службы не имеют внутреннего состояния и могут кешироваться, что обеспечивает более высокую скорость выполнения – очень важный аспект для мобильных устройств;
3) **формат сообщения.** Существует множество широко распространенных и достаточно удобных форматов сообщений, включая XML, RSS и JSON. Однако многие разработчики предпочитают JSON, потому что это простой и удобочитаемый текстовый формат. К тому же имеется множество библиотек, позволяющих легко преобразовывать данные JSON в объекты и обратно. Поскольку JSON не перегружен избыточным синтаксисом, этот формат более компактный, чем XML. Это означает, что он обрабатывается быстрее, т. к. для отправки и получения сообщений требуется меньшая пропускная способность. Особенно хорошо формат JSON подходит для портативных и мобильных устройств с ограниченным объемом памяти, слабой вычислительной мощностью и низкой пропускной способностью, таких как мобильные телефоны и планшеты.

Разные разработчики имеют разные потребности и предпочтения, поэтому в данной главе мы рассмотрим только рекомендации. Вы должны будете сделать свой выбор, исходя из ваших потребностей, требований к производительности и удобства.

Сопровождение микросервисов

Сделав выбор способа взаимодействия между микросервисами и их реализации, вы должны будете сопровождать и поддерживать эти службы в работоспособном состоянии. Широко известные слова «ничто не вечно под луной» в равной степени применимы и к программному обеспечению. Запросы на корректировку существующих функциональных возможностей всегда будут сопровождаться новыми условиями, которые иногда будут требовать изменения реализации веб-служб. Это одна из сложностей микросервисов, о которой мы уже говорили. Чтобы удовлетворить меняющиеся потребности, вам придется позаботиться о следующих аспектах:

- **поддержка существующих реализаций клиентов.** Иногда при изменении основных функций микросервиса может потребоваться изменение его интерфейсов. Вы должны позаботиться об обратной совместимости микросервиса, т. к. есть вероятность, что один или несколько других микросервисов (потребителей) используют его публичный интерфейс для взаимодействий. Говоря иначе, вы должны обеспечить поддержку старой версии, пока разработчики микросервисов-потребителей не изменят свои реализации и не перейдут на использование нового интерфейса;
- **отказоустойчивый дизайн.** Если вызываемая веб-служба не отвечает по какой-то причине, эту проблему можно решить несколькими способами. Самый простой – добавить в клиентский код ограничение времени ожидания. Со стороны поставщика нужно предусмотреть обработку всех ошибочных ситуаций и возврат соответствующих кодов ошибок, а в некоторых случаях и значений по умолчанию. Этот подход также способствует устранению неполадок;
- **мониторинг.** Предусмотрите активный контроль микросервисов с помощью обращения к ним через регулярные промежутки времени или с помощью других методов. Реализуйте соответствующие меры для случая, если какой-то из микросервисов не отвечает. Возможно, вам придется организовать точную балансировку, потому что мониторинг будет создавать дополнительный трафик. Для достижения высокой доступности и отказоустойчивости можно использовать сторонние фреймворки, такие как Marathon. Если, например, требуется иметь два действующих экземпляра микросервиса, можно воспользоваться механизмом мониторинга в Marathon для обнаружения отказавшей службы и запуска нового экземпляра;
- **очередь.** При разработке асинхронных веб-служб используйте шаблон взаимодействий «публикация/подписка». При такой организации даже в том случае, если служба неожиданно завершится, после повторного запуска она извлечет запрос из очереди и обработает его.

В результате преобразования монолитной архитектуры в архитектуру микросервисов может получиться несколько сотен микросервисов и тысяч веб-служб или служб обмена сообщениями между этими микросервисами, поэтому соблюдение перечисленных рекомендаций имеет первостепенное значение.

Обнаружение службы

Как клиент будет взаимодействовать с приложением, насчитывающим сотни или тысячи микросервисов, особенно в том случае, если доступ к каждому микросервису может осуществляться посредством нескольких веб-служб (например, для разных версий клиентов). Это незначительная проблема в монолитной архитектуре, т. к. клиент делает только один вызов, а обо всем остальном позаботится само приложение. Но в архитектурах на основе микросервисов возникают две сложности:

1) клиенты вынуждены одновременно вызывать несколько микросервисов для достижения того же эффекта, который в монолитном приложении обеспечивался единственным вызовом;
2) клиенты должны знать адреса служб.

Проиллюстрируем эти сложности на простом примере. Представьте, что пользователь обращается к приложению книжного онлайн-магазина и хочет просмотреть страницу своей учетной записи. На странице отображаются история заказов, рекомендации, текущее состояние корзины, платежи, настройки учетной записи и т. д. В монолитном приложении, когда пользователь переходит по ссылке **Моя учетная запись**, он получает страницу **Моя учетная запись**, а все волшебство творится в серверной части приложения, где вызываются разные функции и извлекаются сведения из базы данных. В портативных и мобильных устройствах может потребоваться выполнить другой набор вызовов, учитывая ограниченность объема памяти и вычислительной мощности, что добавляет сложности.

В архитектуре на основе микросервисов клиент сам мог бы вызывать все необходимые микросервисы, возвращающие, например, состояние корзины, информацию о платежах и параметры учетной записи. Но такой подход очень неэффективен и предусматривает жестко ограниченный порядок взаимодействий. Если использовать его, мы потеряем гибкость внесения изменений, таких как дальнейшее разделение микросервисов на несколько микросервисов, когда это необходимо, или наоборот.

Кроме того, клиент должен знать адреса всех микросервисов, которые нужно вызвать, чтобы сформировать страницу **Моя учетная запись**. Поэтому нам нужна система, которая будет играть роль общей точки входа для клиентов и внешних вызовов, и еще одна система, хранящая адреса микросервисов.

API-шлюз

API-шлюз решает первую проблему и выступает в роли единой точки входа. Он отвечает за прием запросов от клиентов, вызов необходимых микросервисов и отправку результатов клиенту. Благодаря API-шлюзу клиент посылает только один запрос. Эта модель предлагает несколько преимуществ, перечисленных ниже:

1) внутренняя сложность приложения скрыта от клиента, что упрощает реализацию клиентского кода;

2) увеличение гибкости изменения, объединения, разделения или удаления микросервисов;
3) уменьшение трафика между клиентом и приложением и увеличение за счет этого эффективности.

Кроме того, API-шлюз может служить точкой входа для балансировки нагрузки, аутентификации, мониторинга и управления. Он может предоставлять разные API разным версиям клиента, а также распознавать приоритеты запросов.

Самый большой недостаток этой модели состоит в том, что API-шлюз может стать единой точкой отказа и узким местом, причем как с точки зрения производительности, так и с точки зрения разработки. API-шлюз должен проектироваться, настраиваться и обслуживаться несколькими группами разработчиков, а значит, этот процесс должен быть простым и эффективным. Например, он должен обновляться одновременно с изменением, добавлением или удалением микросервисов. С точки зрения эксплуатации, эластичная подсистема балансировки нагрузки должна гарантировать соответствие характеристик производительности и доступности.

Реестр служб

При наличии тысяч микросервисов наш API-шлюз должен знать адреса всех служб, участвующих в работе приложения. Для этой цели часто используется реестр служб – база данных микросервисов с их адресами, к которой можно обратиться при необходимости. Разработчик должен позаботиться о создании записи для своего микросервиса в этом реестре.

Суть заключается в том, чтобы при запуске микросервис регистрировал себя в реестре. Когда клиент выполняет вызов, API-шлюз определяет адреса необходимых микросервисов, вызывает их и объединяет полученные результаты для передачи обратно клиенту. На первый взгляд, идея выглядит красивой, но, как известно, дьявол кроется в деталях.

Как быть, если мы потеряем реестр с данными? Для решения этой проблемы можно использовать несколько инструментов с открытым исходным кодом, таких как Consul и SkyDNS, которые фактически обнаруживают микросервисы и удостоверяются, что они запущены и работают. Например, Consul – это зрелый инструмент, способный использовать DNS-имена для доступа к микросервисам и хранить эту информацию в реестре. Он также может выполнять периодические проверки работоспособности и поддерживать работу кластеров.

Объединяем все вместе

Рассмотрим пример простой системы, а затем расширим ее, используя вновь приобретенные знания. На рис. 3.1 показана простая модель на основе микросервисов. На этом этапе клиент сам вызывает все необходимые микросервисы для обработки запроса пользователя.

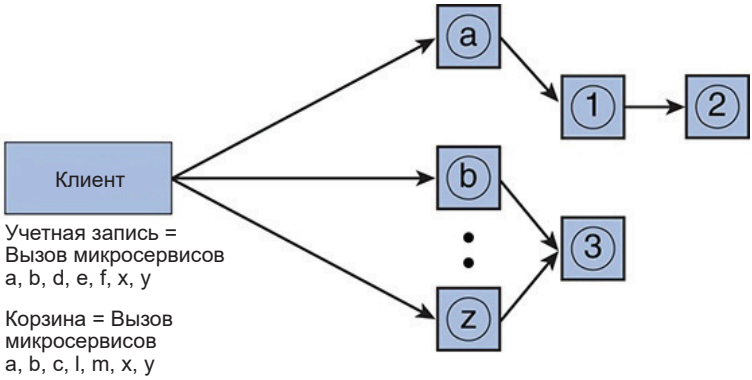


Рис. 3.1 ❖ Простая модель,
в которой клиент сам вызывает все необходимые микросервисы

Добавим API-шлюз, чтобы инкапсулировать бизнес-логику приложения и скрыть сложности от клиента. Это позволит упростить реализацию и даст клиенту возможность выполнять единственный вызов для получения такого же эффекта, что показано на рис. 3.2.

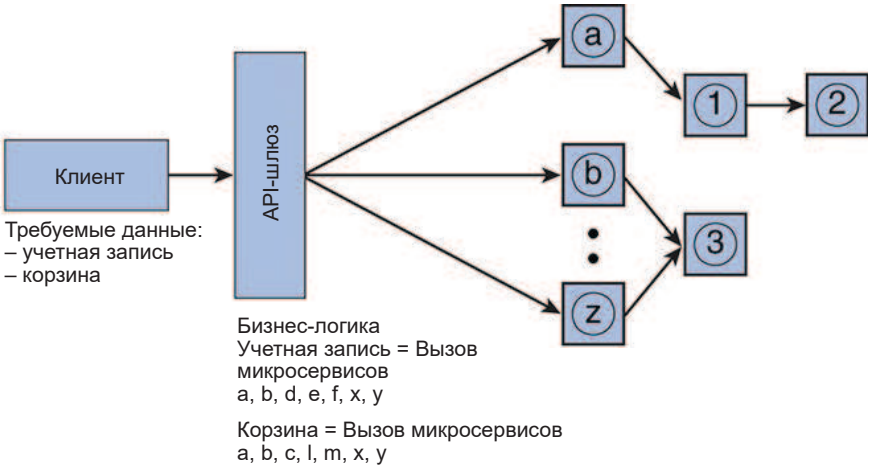


Рис. 3.2 ❖ Модель на основе микросервисов с API-шлюзом,
позволяющим клиенту получить тот же результат, выполнив всего один вызов

Теперь добавим реестр, чтобы API-шлюз мог определить каждый необходимый ему микросервис. Как уже говорилось, все микросервисы (от **a** до **z** и от **1** до **3** на рисунках) регистрируются в реестре. Когда поступает новый запрос, API-шлюз определяет, какие микросервисы он должен вызвать, затем извлекает их адреса из реестра и выполняет вызовы. Далее он объединяет результаты и возвращает их в виде HTTP-ответа клиенту, как показано на рис. 3.3.

Выглядит пугающе, но на самом деле все просто. Теперь сделаем еще шаг и попробуем масштабировать отдельные службы, воспользовавшись одним из ключевых преимуществ микросервисов – возможностью масштабирования по мере необходимости. Для этого достаточно добавить балансировщики нагрузки в нуж-

ные точки. Балансировка нагрузки должна осуществляться не только на уровне микросервисов, но и на уровне API-шлюза и реестра, потому что они тоже могут стать узким местом. Получившаяся модель изображена на рис. 3.4.

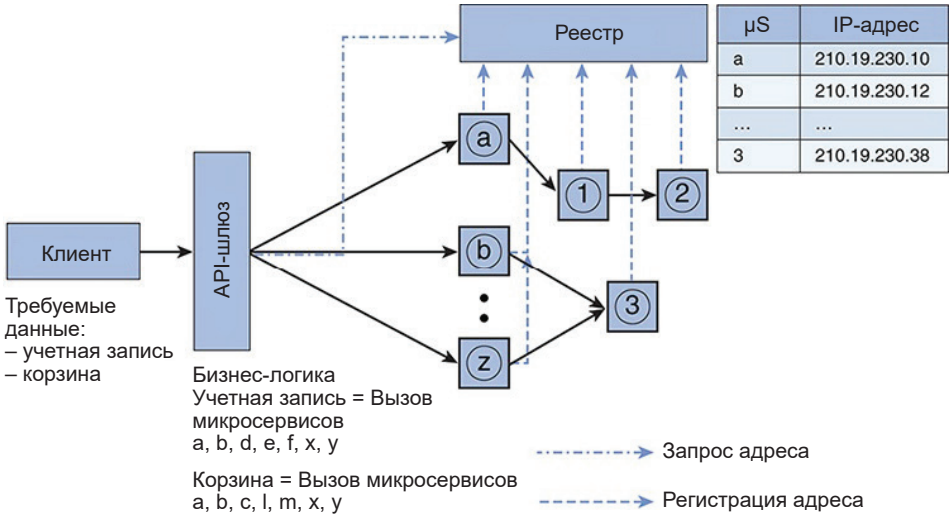


Рис. 3.3 ❖ Модель на основе микросервисов с реестром служб

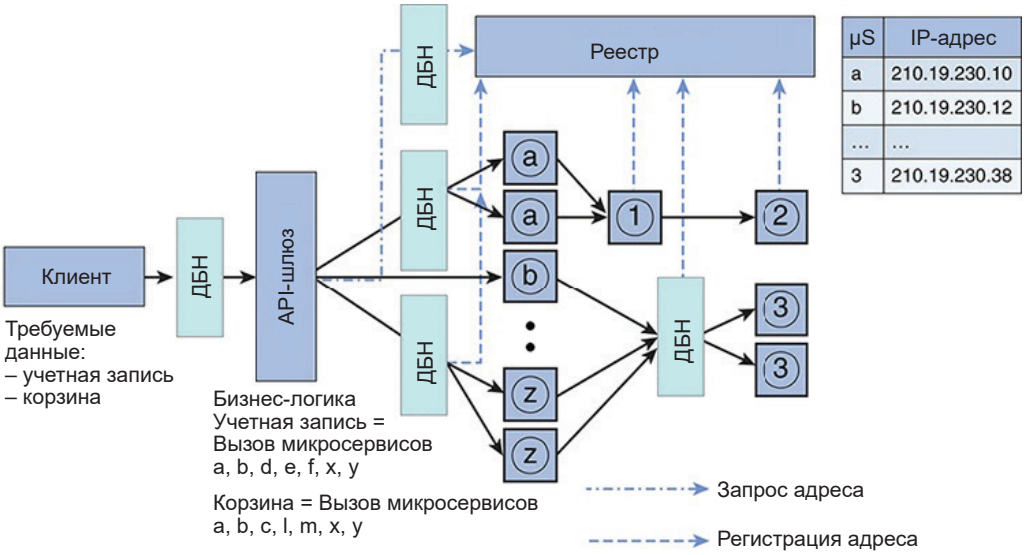


Рис. 3.4 ❖ Модель на основе микросервисов с балансировкой нагрузки (ДБН – динамический балансировщик нагрузки)

Вопросы балансировки микросервисов мы подробно обсудим в третьей части книги «Практический проект – применение теории на практике», где вашему вниманию будет представлен практический пример на основе Docker.

Глава 4

Миграция и реализация микросервисов

Теперь вы знаете, что такое микросервисы и как они работают. Если вы продолжаете читать эту книгу, значит, я достиг первой своей цели: заинтересовать вас настолько, чтобы вы стали рассматривать возможность реализации микросервисов! Следовательно, сейчас самое время перейти к сути – важнейшей теме миграции в сторону микросервисов.

Что необходимо для миграции

Как мы уже знаем, монолитное приложение является очень большим (в смысле количества строк кода) и сложным (в смысле взаимозависимости функций, данных и т. д.). Оно обслуживает сотни тысяч пользователей в разных географических регионах, и для его поддержки необходимо участие нескольких разработчиков и инженеров. На рис. 4.1 показан пример монолитного приложения.

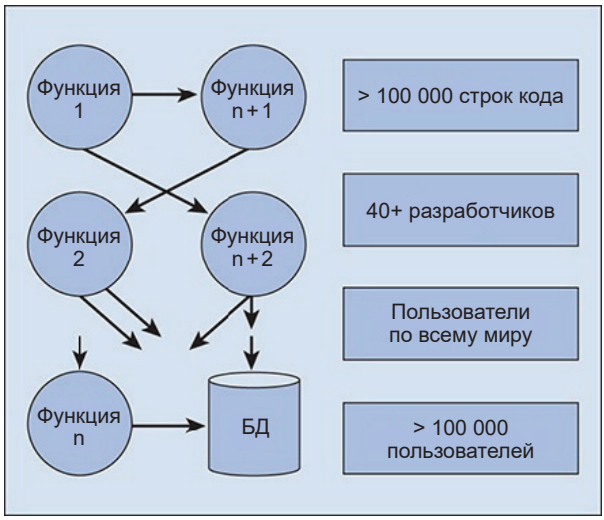


Рис. 4.1 ❖ Структура типичного монолитного приложения

Иногда даже при этих условиях приложение может первое время работать нормально, и у вас не будет проблем с масштабируемостью и производительностью. Однако по мере расширения пользовательской базы или самого приложения могут появляться проблемы, специфичные для разных приложений. Например, в облачном приложении или веб-приложении могут возникнуть проблемы масштабируемости из-за увеличения числа пользователей, а также может оказаться слишком дорого и трудно регулярно выпускать обновления из-за увеличения времени сборки и регрессионного тестирования. Как показано на рис. 4.2, пользователи или разработчики монолитных приложений могут столкнуться с одной или несколькими проблемами, перечисленными справа.

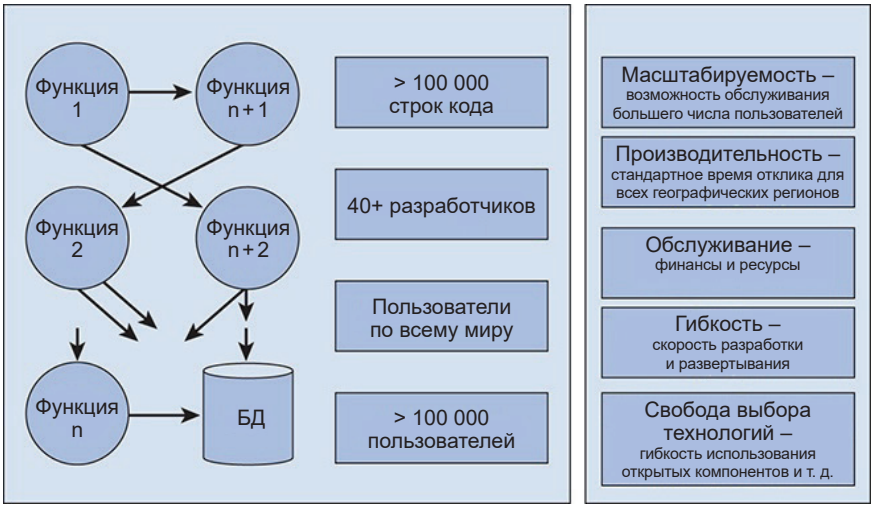


Рис. 4.2 ❖ Потенциальные проблемы, свойственные монолитным приложениям

И тогда миграция в сторону микросервисов может из новомодной идеи превратиться в насущную необходимость. В предыдущих главах мы чуть ближе познакомились с микросервисами и теперь знаем, что миграция, или переход, будет выглядеть, как показано на рис. 4.3.

Но как выполнить такой переход на практике? Есть два возможных сценария: создать новое приложение или преобразовать существующее монолитное приложение. Последний сценарий более вероятен, но независимо от текущей ситуации вы должны знать все достоинства и недостатки обоих сценариев.

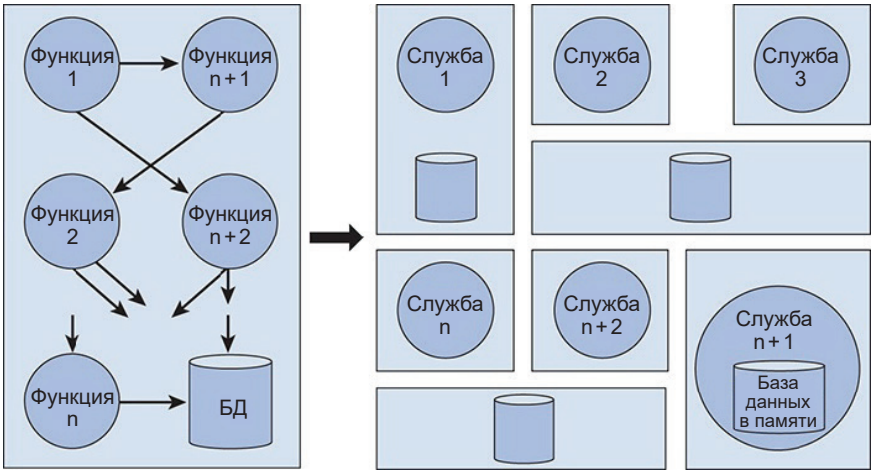


Рис. 4.3 ❖ Переход от монолитной архитектуры к архитектуре микросервисов

Создание нового приложения на основе микросервисов

Прежде чем начать, хочу оговориться, что я видел не так много случаев создания приложения на основе микросервисов с нуля. В моей практике чаще встречался сценарий преобразования существующего приложения и перехода от монолитной архитектуры к архитектуре микросервисов. В этих случаях архитекторы и разработчики всегда старались использовать хотя бы часть уже имеющегося кода. Однако по мере обретения опыта и появления на рынке новых успешных реализаций мы увидим больше примеров создания приложений на основе микросервисов с нуля, поэтому этот сценарий, безусловно, стоит обсудить.

Допустим, вы определили все требования и готовы перейти к проектированию архитектуры приложения. Существует много общих рекомендаций, которые желательно учитывать, приступая к работе, и мы рассмотрим их в следующих разделах.

Готовность организации

Как обсуждалось в главе 2 «Переход к микросервисам», первый вопрос, на который вы должны ответить: готова ли ваша организация к переходу на микросервисы? Различные отделы в вашей организации теперь должны иначе подходить к созданию и выпуску программного обеспечения:

1) **структура команды.** Команду разработчиков монолитного приложения (если она существует) необходимо разбить на несколько небольших групп специалистов, обученных передовым методам работы с микросервисами или знакомых с ними. Как показано на рис. 4.3, новая система будет состоять из множества независимых служб, каждая из которых ответственна за оказание конкретной услуги. Это одно из ключевых преимуществ парадиг-

мы микросервисов – снижение накладных расходов на согласования, в том числе на регулярное проведение собраний. Группы должны быть организованы по бизнес-задачам или областям. В этом случае останется только согласовать сроки и стандарты/протоколы, при соблюдении которых микросервисы смогут работать как единое целое;

2) **гибкость.** Каждая группа должна быть готова действовать независимо от других. Ее численность не должна превышать численности типичной scrum-команды (5–9 человек), иначе согласование снова станет проблемой. Исполнительность является ключевым показателем, и каждая группа должна быть в состоянии быстро удовлетворять меняющиеся потребности бизнеса;
3) **инструменты и обучение.** Одним из основных факторов является готовность организации инвестировать в приобретение новых инструментов и обучение людей. В большинстве случаев придется отказаться от существующих инструментов и процессов и взять на вооружение новые. Это может потребовать больших капиталовложений, а также инвестиций в наем новых специалистов, обладающих необходимыми навыками, и переподготовку существующих сотрудников. В долгосрочной перспективе, если решение о переходе к архитектуре микросервисов было правильным, организация сможет окупить эти инвестиции.

Подход на основе служб

Приложения на основе микросервисов, в отличие от монолитных приложений, должны быть организованы как набор слабо связанных служб, взаимодействующих друг с другом для обеспечения полной функциональности приложения. Каждая служба должна рассматриваться как независимая и автономная служба с собственным жизненным циклом, которая может разрабатываться и поддерживаться отдельной группой специалистов. Группы могут выбирать разные технологии разработки, включая языки программирования или базы данных, которые лучше отвечают потребностям их служб. Например, для сайта электронной коммерции одна группа могла бы написать полностью независимую службу покупательской корзины с базой данных в памяти, а другая группа – микросервис оформления заказа с реляционной базой данных. Действующее приложение может использовать микросервисы для выполнения основных функций, таких как аутентификация, управление учетной записью, регистрация нового пользователя и рассылка уведомлений, с бизнес-логикой, инкапсулированной в API-шлюз, который вызывает эти микросервисы, опираясь на запросы клиентов.

Хочу напомнить, что микросервис может быть маленькой службой, реализованной единственным разработчиком, или большой и сложной службой, развиваемой несколькими разработчиками. Размер не является определяющим фактором для микросервисов. Каждый микросервис должен реализовать одну функцию.

Также на этом этапе необходимо учесть такие аспекты, как возможность масштабирования, производительность и безопасность. Необходимость масштабирования может меняться. При этом должна быть предусмотрена возможность масштабирования каждого микросервиса в отдельности. Безопасность должна быть в центре внимания на всех уровнях, включая хранение данных, межпроцессные взаимодействия и т. д.

Межпроцессные (между службами) взаимодействия

Мы подробно обсудили тему межпроцессных взаимодействий в главе 3 «Межпроцессные взаимодействия». Ключевыми аспектами, которые обязательно следует учесть, являются безопасность и протокол взаимодействий. Особое внимание должно уделяться асинхронным взаимодействиям, потому что в этом случае важно тщательно следить за всеми текущими запросами и стараться не удерживать ресурсы дольше, чем это необходимо.

Для организации таких взаимодействий с успехом можно использовать шину сообщений, такую как RabbitMQ. Она очень проста в использовании и способна передавать до нескольких сотен тысяч сообщений в секунду. Чтобы предотвратить превращение системы передачи сообщений в единую точку отказа, если она потерпит аварию, необходимо правильно настроить шину сообщений и обеспечить ее высокую доступность. Еще одним вариантом, заслуживающим внимания, является ActiveMQ – другая легковесная платформа обмена сообщениями.

Еще одним ключевым фактором на данном этапе является безопасность. Кроме выбора правильного протокола связи, для мониторинга и оценки межпроцессных взаимодействий можно использовать стандартные инструменты, такие как AppDynamics. Информация о любых аномалиях должна автоматически передаваться в службу безопасности.

Когда приложение состоит из тысяч микросервисов, трудно уследить за ними всеми. Мы уже обсуждали в главе 3, как решать такие проблемы с помощью механизма обнаружения служб и API-шлюзов.

Выбор технологий

Самым большим преимуществом перехода к архитектуре на основе микросервисов является возможность выбора. Каждая группа может самостоятельно выбирать лучше всего подходящие для данного микросервиса язык, технологию, базу данных и т. д. В монолитном подходе у команды разработчиков, как правило, нет такой гибкости, поэтому убедитесь, что не упускаете ценных возможностей.

Даже если несколькими микросервисами занимается одна группа, все равно каждый микросервис следует рассматривать как автономный. Масштабируемость, развертывание, время сборки, интеграция, работоспособность и т. д. – все это следует учитывать при выборе технологии для каждого микросервиса. Для микросервисов, быстро оперирующих небольшими объемами данных, вполне может подойти база данных в памяти, тогда как другие микросервисы могут совместно использовать реляционные базы данных или базы данных NoSQL.

Реализация

Реализация – важнейший этап. Именно здесь пригодятся новые приемы и знания. Вот несколько важных аспектов, о которых следует помнить на этапе реализации:

1) **независимость**. Каждый микросервис должен быть максимально автономным, имеющим собственный жизненный цикл. Микросервисы должны разрабатываться и эксплуатироваться независимо от любых других микросервисов;

2) **управление исходным кодом.** Необходимо использовать надежную и стандартную систему управления версиями для хранения исходного кода микросервисов. Стандартизация гарантирует использование всеми группами одной и той же системы управления версиями. Это упростит обзор кода и обеспечит доступ ко всему коду в одном месте. В перспективе есть смысл хранить весь исходный код в одном месте;
3) **окружение.** Все разные окружения (для разработки, тестирования, сборки и эксплуатации) должны быть максимально защищенными и автоматизированными. Под автоматизацией в данном случае понимается процесс сборки. В этом случае код может интегрироваться практически на ежедневной основе. Существует множество готовых инструментов автоматизации, из которых мне хотелось бы упомянуть Jenkins – инструмент с открытым исходным кодом, позволяющий автоматизировать процесс сборки и выпуска программного обеспечения, включая непрерывную интеграцию и непрерывное развертывание;
4) **отказоустойчивость.** В мире нет ничего абсолютно надежного, и программные ошибки неизбежны. Обработка ошибок в службах – это обычная задача, решаемая в процессе разработки микросервисов. Отказы других служб должны обрабатываться так, чтобы они оставались максимально незаметными для конечного пользователя, включая управление временем ожидания ответа, обработку изменений в API других служб и ограничение количества повторных попыток;
5) **повторное использование.** Разрабатывая микросервисы, не нужно бояться повторно использовать имеющийся код, копируя и вставляя его фрагменты, но не злоупотребляйте этим. Это может привести к появлению повторяющихся фрагментов, но все же это лучше, чем использовать общий код, который может образовать тесную зависимость между службами (микросервисы должны быть максимально независимыми друг от друга). Например, представьте, что вы написали код, принимающий ответ другой службы. Нет ничего плохого в том, если вы скопируете и вставите этот код везде, где вызывается эта служба. Другой способ повторного использования кода – создание общих библиотек. Несколько клиентов может использовать одну и ту же библиотеку, но тогда для каждого клиента придется позаботиться о поддержке таких библиотек. Иногда это может вызывать сложности, особенно в том случае, если было создано большое количество библиотек и каждый клиент использует свою версию. В этом случае может потребоваться подключение нескольких версий одной и той же библиотеки. Тогда процесс сборки станет весьма затруднительным из-за необходимости сохранения обратной совместимости и других подобных проблем. Вы можете выбрать этот путь, если сумеете жестко контролировать количество библиотек и их версий. Он определенно поможет вам избежать дублирования кода;
6) **маркировка.** Учитывая огромное количество микросервисов, отладка может превратиться в очень сложную задачу, поэтому необходимо предусмотреть некоторый способ маркировки. Для этого можно снабжать каждый запрос уникальным идентификатором (меткой) и записывать информацию о нем в журнал. Этот идентификатор должен однозначно определять

отправителя запроса и передаваться всеми службами, принимающими, обрабатывающими и передающими этот запрос дальше. Столкнувшись с проблемой, вы сможете заглянуть в журнал, отследить путь запроса и выявить проблемную службу. Это решение особенно эффективно, когда используется единая система журналирования. Все службы должны фиксировать принимаемые сообщения в этой системе, используя стандартный формат, чтобы разработчики легко могли восстановить ход событий. Для поддержки централизованного журналирования предпочтительнее использовать общие библиотеки. На рынке есть несколько инструментов управления и агрегирования журналов, таких как ELK (Elasticsearch, Logstash, Kibana) и Splunk, идеально подходящих для данной цели.

Развертывание

Автоматизация является ключевым аспектом процесса развертывания. Без нее практически невозможно добиться успеха в использовании парадигмы микросервисов. Как уже говорилось, количество микросервисов может исчисляться сотнями и тысячами, поэтому наличие гибкой и автоматизированной системы развертывания является обязательным условием.

Представьте, что вы должны развертывать и сопровождать тысячи микросервисов. Что случится, если один из микросервисов потерпит аварию и остановится? Как узнать, достаточно ли ресурсов на компьютере для запуска микросервисов? Решать подобные задачи без автоматизации очень сложно. Существуют разные инструменты для автоматизации процесса развертывания, такие как Kubernetes и Docker Swarm. Учитывая важность этой темы, ей будет посвящена отдельная глава (глава 9 «Организация контейнеров»).

Эксплуатация

Процесс эксплуатации также должен быть максимально автоматизированным. Учитывая, что речь идет о сотнях и тысячах микросервисов, инструменты автоматизации должны быть достаточно зрелыми, чтобы справиться с таким уровнем сложности. Вам потребуется организовать целую систему поддержки, включающую компоненты, которые перечислены ниже:

1) **мониторинг.** Контролироваться должно все, от инфраструктуры до программных интерфейсов приложений, и должна быть предусмотрена рассылка соответствующих уведомлений. Подумайте о возможности создания дашбордов с оперативными данными и предупреждениями, появляющимися при возникновении проблем;
2) **автоматическая масштабируемость.** В случае с микросервисами задача масштабирования решается довольно просто. Достаточно просто создать еще один экземпляр требуемого микросервиса и поместить его за соответ-

ствующим балансировщиком нагрузки. Но масштабирование также должно быть автоматизировано. Как вы узнаете далее, все сводится к определению целочисленного значения, которое влияет на количество экземпляров микросервиса, действующих одновременно;

3) **экспортирование API.** Во многих случаях необходимо открыть доступ к API извне для использования внешними клиентами. Лучше всего для этого установить и настроить пограничный сервер, который будет принимать все внешние запросы. Он мог бы использовать API-шлюз и систему обнаружения служб. При таком походе можно настроить по одному пограничному серверу для разных типов устройств (например, один – для мобильных устройств, другой – для обслуживания браузеров и т. д.) или вариантов использования. С этой целью можно использовать открытое приложение Zuul, созданное в компании Netflix;
4) **выключатель.** Нет смысла посылать запросы отказавшей службе. Поэтому можно предусмотреть выключатель, который будет следить за работоспособностью служб и при обнаружении многочисленных отказов в какой-то службе блокировать отправку запросов ей (выключать ее) на некоторое время. По истечении заданного времени выполняется попытка послать запрос выключенной службе, и в случае успешного получения ответа служба опять включается. Выключение и включение должно производиться отдельно для каждого экземпляра службы. Реализацию такого выключателя можно найти в открытом приложении Hystrix компании Netflix.

Переход от монолитной архитектуры к архитектуре микросервисов

Несмотря на то что большинство рекомендаций по созданию приложений на основе микросервисов с нуля подходит также для случая перехода от монолитной архитектуры к архитектуре микросервисов, есть несколько дополнительных правил, которые помогут выполнить такой переход проще и эффективнее.

На первый взгляд, кажется правильным преобразовать сразу все монолитное приложение в набор микросервисов, однако это не самый эффективный, а иногда и довольно дорогостоящий путь. В конце концов, можно просто написать новое приложение с нуля. Более правильное решение – поэтапный переход, как показано на рис. 4.4.

Следующий вопрос: с чего начать преобразование текущего монолитного приложения? Если приложение действительно старое и разделить его на компоненты (из-за тесных связей на верхнем уровне) будет очень сложно, тогда, вероятно, лучше начать с нуля. В том случае, если есть возможность быстро отключить части кода, а технология, на которой основано приложение, не полностью устарела, лучше начать с преобразования компонентов в микросервисы и замены старого кода.

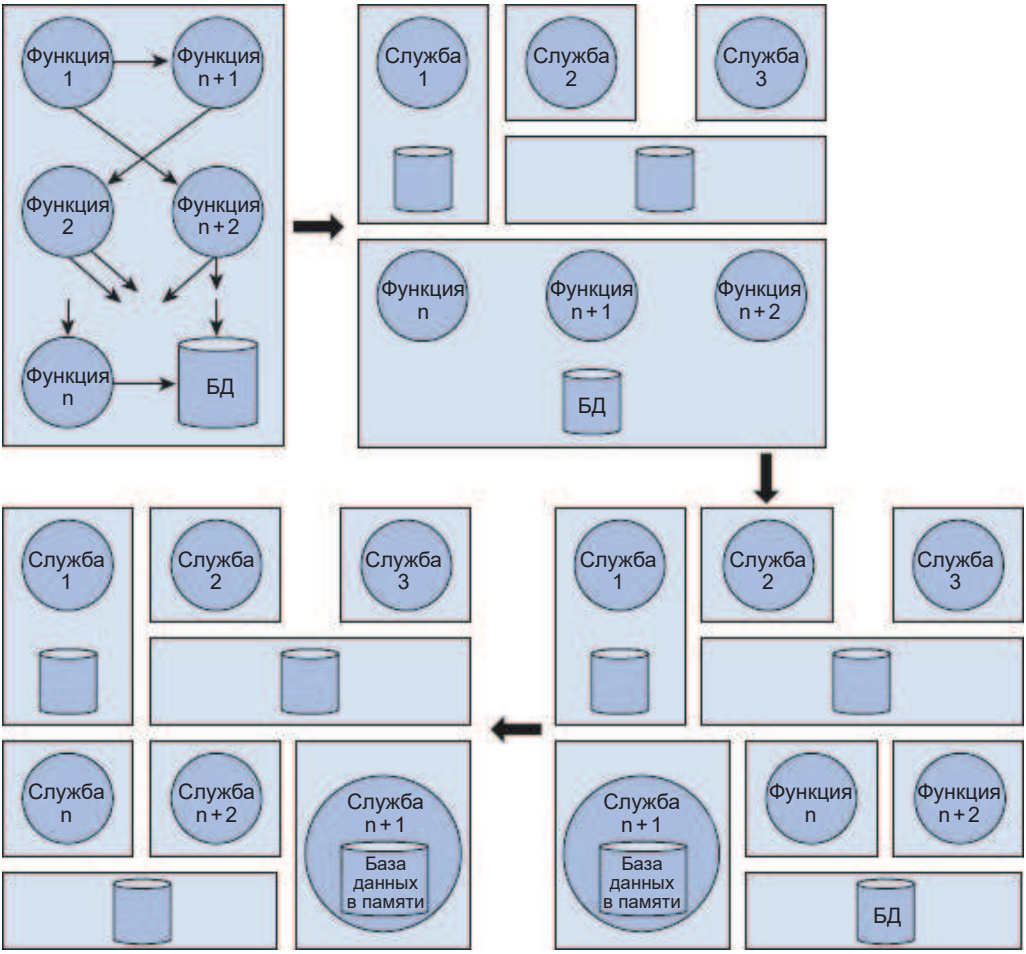


Рис. 4.4 ❖ Основные этапы миграции от монолитной архитектуры к архитектуре микросервисов

Критерии выделения микросервисов

Далее возникают вопросы: какие компоненты следует преобразовать в первую очередь и стоит ли их вообще преобразовывать? Для ответа на них необходимо установить правила, которые я называю «критериями выделения микросервисов». Они описывают способы выбора функций для преобразования в микросервисы и их очередность. Эти правила определяют, какие компоненты существующего монолитного приложения должны быть преобразованы в микросервисы с учетом потребностей организации в данный момент.

Оговорка «в данный момент» вовсе не случайна, потому что потребности организации могут изменяться со временем. Позднее вам, возможно, придется вер-

нуться и преобразовать в микросервисы еще какие-то компоненты. Иначе говоря, с изменением потребностей может понадобиться преобразование дополнительных компонентов монолитного приложения.

Ниже перечислено несколько правил, которые можно рассматривать как критерии выделения микросервисов:

- **масштабирование.** Вы должны определить, какие функции используются наиболее интенсивно. В первую очередь преобразуйте в микросервисы именно такие функции или компоненты. И помните, что один микросервис должен выполнять только одну функцию. Не забывайте об этом и, выполняя деление приложения, руководствуйтесь этим принципом;
- **производительность.** Часто в монолитных приложениях есть компоненты, работающие нелучшим образом, и есть более привлекательные альтернативы (например, плагин с открытым исходным кодом или возможность создать соответствующую службу с нуля). В таких ситуациях важно помнить о границах микросервисов. Если вы реализуете микросервис, который выполняет одну-единственную функцию, то все в порядке. Часто бывает трудно правильно определить границы микросервиса, но по мере обретения опыта это будет даваться вам все легче и легче. Определить границы микросервиса можно иначе: если вы сможете переписать микросервис за несколько недель (если это потребуется), а не за несколько месяцев, следовательно, границы определены правильно;
- **лучшие технологии, или языки программирования.** Иногда предметные задачи легче решаются с использованием предметно-ориентированных языков программирования. Это в особенной степени относится к компонентам, к которым было множество нареканий в прошлом, причем высока вероятность, что нарекания продолжатся. Если вы уверены, что можно не только усовершенствовать реализацию такого компонента с использованием нового языка или новых возможностей, но также упростить его сопровождение и обновление в будущем, значит, сейчас самое подходящее время, чтобы внести такие изменения. Также иногда вы можете обнаружить, что другой язык обеспечивает более простые и удобные абстракции параллельного выполнения. В таком случае можно написать микросервис на новом языке, а для остальной части приложения продолжать использовать прежний язык. Аналогично может появиться желание ускорить работу некоторых микросервисов, и вы решите переписать их на языке C, чтобы получить максимальный выигрыш. Так воспользуйтесь этой гибкостью;
- **лучшие механизмы хранения данных.** С увеличением объемов данных некоторые компоненты приложения могут извлечь дополнительные выгоды от замены реляционных баз данных хранилищами NoSQL. Если в вашем приложении есть такой компонент, возможно, пришло время перевести его на использование базы данных NoSQL.

Это были основные критерии, помогающие разграничить компоненты монолитного приложения и определить необходимость их преобразования в микро-

сервисы. Но, кроме этого, требуется также установить очередность преобразования, если выявится несколько таких компонентов. Для этого можно использовать правила, перечисленные ниже:

- **частота изменения.** Изменение и совершенствование функций играют важную роль в жизненном цикле любого приложения. Функции, которые изменяются чаще всего, вероятно, являются лучшими кандидатами на преобразование в микросервисы в первую очередь. Выделение таких функций в отдельные микросервисы уменьшит затраты времени на сборку и развертывание, потому что при последующих изменениях вам придется пересобрать только этот микросервис, но не все приложение целиком, что, в свою очередь, может положительно сказаться на доступности остального приложения;
- **сложность развертывания.** В приложениях всегда есть компоненты, увеличивающие сложность развертывания. В монолитном приложении, даже если такой компонент не изменялся, все равно приходится проходить сложный процесс его сборки и развертывания. Такие компоненты полезно выделить в микросервисы, чтобы сократить общее время развертывания остальной части монолитного приложения. Мы еще вернемся к этой теме, когда поближе познакомимся с контейнерами;
- **вспомогательные службы.** В большинстве приложений основная (главная) служба зависит от нескольких вспомогательных служб. Недоступность таких вспомогательных функций может повлиять на доступность основной службы. В примере нашего приложения для службы поддержки, которое обсуждается в главе 11, функция регистрации проблем зависит от функции каталога продуктов. Если функция каталога продуктов недоступна, пользователь не сможет зарегистрировать проблему. В таких случаях вспомогательные функции следует преобразовать в микросервисы в первую очередь и сделать их максимально доступными, чтобы обеспечить бесперебойную работу основных служб.

Эти критерии могут потребовать преобразовать в микросервисы большинство функций, и это нормально. Цель этих критериев состоит в том, чтобы помочь вам выбрать компоненты для преобразования, расставить приоритеты и определить план перехода к архитектуре микросервисов.

Реорганизация служб

После определения функций для преобразования в микросервисы их необходимо реорганизовать, следуя рекомендациям, описанным выше. При этом нужно иметь в виду следующие аспекты:

- **определение микросервисов.** Для каждой функции определите соответствующий микросервис, включая механизм взаимодействий (API), технологию и т. д. Посмотрите, какие данные использует существующая функция, и разработайте стратегию работы с данными в микросервисе. Если функция активно использует тяжелые базы данных, такие как Oracle, возможно, имеет смысл перейти к использованию MySQL. Определите, как вы будете управлять отношениями в данных. Наконец, запустите каждый микросервис как отдельное приложение;

- **рефакторинг кода.** В микросервисе можно использовать часть существующего кода, если вы решите оставить прежний язык программирования. Подумайте о реализации уровня хранения данных – с общей или выделенной, с внешней или внутренней (в памяти) базой данных. Цель в данном случае – не добавлять новых особенностей без необходимости, а переупаковать существующий код и организовать доступ к нему через API;
- **управление версиями.** Прежде чем приступить к программированию, выберите механизм управления версиями и неуклонно следуйте выбранным стандартам. Каждый микросервис должен оформляться как отдельный проект и развертываться как отдельное приложение;
- **миграция данных.** Если вы решите создать и использовать новую базу данных, вам также придется перенести в нее имеющиеся данные. Обычно эта задача сводится к разработке простых сценариев на SQL в зависимости от типа прежней и новой баз данных;
- **монолитный код.** На первом этапе не удаляйте код из монолитного приложения на случай отката. Вы можете изменить остальной код для использования новых микросервисов или, что еще лучше, разделить трафик приложения, чтобы иметь возможность одновременно использовать обе версии – монолитную и микросервисы. Это позволит проверить и проанализировать изменение производительности. Когда будет принято окончательное решение, вы сможете перенаправить весь трафик в микросервисы и отключить/удалить старый код;
- **независимая сборка, развертывание и управление.** Сборка и развертывание микросервисов должны производиться независимо. При развертывании новых версий микросервисов можно снова разделить трафик между старой и новой версиями на некоторое время. Таким образом, в эксплуатационной среде может быть запущено несколько версий одного и того же микросервиса. Часть пользовательского трафика можно перенаправить в новую версию микросервиса, чтобы убедиться, что она работает, причем работает правильно. Если новая версия работает не так эффективно, как ожидалось, вы легко сможете вернуть весь трафик в предыдущую версию, а новую версию отправить на доработку. Ключевыми моментами здесь являются настройка итеративного процесса автоматического развертывания и переход к непрерывному развертыванию;
- **удаление старого кода.** Удалить временный код и данные из старого хранилища можно только тогда, когда вы убедитесь, что миграция прошла успешно и все работает должным образом. Не забудьте попутно сделать резервные копии.

Гибридный подход

Создавая приложение с нуля, разработчики могут сразу следовать принципам архитектуры микросервисов, которые обсуждались выше. Но иногда разработчики выбирают своеобразный гибридный подход, разрабатывая часть приложения в виде набора микросервисов, а другую часть – следуя стандартным приемам SOA/MVC, основанным на определенных критериях. Часто это обусловлено тем, что не все компоненты приложения можно квалифицировать как микросервисы.

Как уже говорилось в главе 3, микросервисы дают большую гибкость, но она имеет свою цену. Гибридный подход позволяет сбалансировать гибкость и затраты и предполагает извлечение некоторых компонентов из монолитной части и преобразование их в микросервисы по мере необходимости. В этом случае важно иметь в виду оба подхода и помнить критерии выделения микросервисов во время перехода.

Часть II

КОНТЕЙНЕРЫ

Глава 5

Контейнеры Docker

В этой главе рассматривается еще одна популярная тема – контейнеры Docker. По мере расширения компании испытывают трудности с развертыванием и масштабированием программных компонентов. С течением времени и увеличением числа пользователей и функций программное обеспечение имеет тенденцию усложняться, а затем начинают возникать сложности с его развертыванием и масштабированием. В главе 1 «Введение в микросервисы» мы говорили, что микросервисы способны избавить от сложностей в разработке за счет упрощения архитектуры и способствуют упрощению операций развертывания и масштабирования. Однако, реализуя архитектуру на основе микросервисов, вы наверняка получите тысячи служб. Каждую из них нужно развернуть. Также необходимо организовать управление ими. В такой ситуации большинство проблем можно решить с помощью контейнеров.

Docker – это технология с открытым исходным кодом, которая решает проблемы развертывания и масштабирования путем отделения приложений от зависимостей инфраструктуры. Она решает эти проблемы благодаря применению контейнеров, позволяющих упаковать приложение со всеми его зависимостями, включая структуру каталогов, метаданные, пространство процессов, номера сетевых портов и т. д. Приложение, упакованное в контейнер, запускается одинаково на любых машинах и в любых окружениях. Именно эта особенность сделала технологию Docker особенно интересной и обеспечила ей стремительный взлет. Кто-то мог бы подумать, что то же самое делают виртуальные машины (ВМ). Чтобы понять разницу, давайте посмотрим, чем отличаются эти технологии.

Виртуальные машины

Виртуальная машина в простейшем ее виде – это автономная система, включающая в себя все, начиная от собственной операционной системы (которая называется гостевой) до окружения и самого приложения. На хост (или физическую машину) можно установить несколько виртуальных машин с помощью слоя, называемого *гипервизором* и действующего поверх ОС хост-машины. Этот гипервизор, который также называют гипервизором типа 2, действует как прокси-сервер для доступа к оборудованию, благодаря чему гостевая ОС «думает», что работает на обычном компьютере (см. рис. 5.1). Гипервизор типа 1 работает непосредственно на оборудовании (без ОС) и считается гипервизором оборудования.

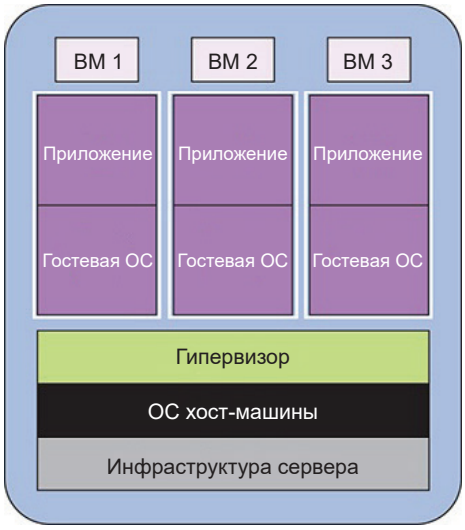


Рис. 5.1 ❖ Простая архитектура виртуальной машины с гипервизором поверх операционной системы хост-машины (гипервизор типа 2)

Идея виртуальных машин получила широкое распространение и создала многомиллиардную индустрию, потому что они позволили организациям максимально эффективно использовать имеющиеся аппаратные ресурсы. До появления технологии виртуализации компании запускали приложения на выделенных серверах. В разработке мы с успехом могли совместно использовать некоторую инфраструктуру, но в рабочей среде все ресурсы сервера выделялись единственному приложению. Это вело к напрасному расходованию ресурсов, когда приложение не могло постоянно использовать все ресурсы. Мы все знаем, насколько мощными со временем стали серверы. Виртуализация дала возможность эффективно использовать ресурсы сервера, обеспечив при этом такое разделение приложений, что каждое из них может работать в своей собственной ОС в отдельной виртуальной машине. Эта модель пользовалась большим успехом и дала начало облачным вычислениям. Виртуальные машины предлагают много преимуществ:

- **эффективность.** Виртуальная машина для приложений выглядит и действует как отдельная физическая машина. Главными преимуществами являются эффективное использование ресурсов и изоляция, обеспечивающая дополнительную безопасность;
- **гибкость.** Есть возможность перераспределять ресурсы между виртуальными машинами так, как угодно. Процессоры, память и другие аппаратные ресурсы можно распределять в соответствии с начальными условиями и фактическими потребностями. Кроме того, есть возможность настроить автоматическое распределение ресурсов, чтобы увеличить производительность. Эта идея получила название *эластичности*;
- **резервное копирование и восстановление.** Виртуальные машины могут храниться в виде одиночных файлов, резервные копии которых легко сохранять. Если потребуется, эти резервные копии легко можно вернуть на место;

- **свобода выбора ОС.** Под управлением одного и того же гипервизора может действовать несколько разных операционных систем, т. е. есть возможность поддерживать приложения, для выполнения которых требуются разные операционные системы;
- **производительность и перемещение.** Виртуальную машину можно легко перенести на другой, более производительный хост. Большинство гипервизоров поддерживает такую возможность. VMware, очень успешное решение виртуализации, предлагает, например, функцию VMotion, которая позволяет выполнять миграцию запущенной виртуальной машины с одного хоста на другой.

Есть также много других преимуществ, таких, например, как экономия затрат, но в этом обсуждении мы рассматриваем только наиболее важные.

Но зачем тогда могут понадобиться контейнеры? Чтобы ответить на этот вопрос, нужно также рассмотреть проблемы, характерные для виртуальных машин. Взглянув еще раз на рис. 5.1, можно заметить некоторые из этих проблем. У нас есть хост-машина со своей ОС. Также есть гипервизор и дополнительные ОС в каждой виртуальной машине. Мы все знаем, что операционные системы сами потребляют значительные объемы ресурсов. Во-первых, им необходимы большой объем памяти и значительная вычислительная мощность. Во-вторых, резервная копия виртуальной машины (хотя часто это один-единственный файл) имеет очень большой размер, потому что этот файл содержит ОС (Windows, Linux и т. д.), установленное приложение с зависимостями и локальные данные. Некоторые резервные копии виртуальных машин могут иметь в размерах более 20 Гбайт. В результате возникают некоторые сложности:

- **общий доступ к виртуальным машинам.** Перемещение виртуальных машин и организация общего доступа к ним из WAN занимают много времени из-за большого размера;
- **переносимость.** Когда один разработчик передает подготовленную виртуальную машину другому разработчику, он не прекращает разработку, а продолжает вносить изменения в приложение, базу данных, окружение и т. д. По истечении некоторого времени разработчик будет вынужден вновь передать виртуальную машину целиком, т. к. нет никакой возможности создать diff-файл и передать только его. Аналогичные проблемы возникают при переходе от разработки к тестированию в эксплуатационном окружении. На выбор есть два пути: или перекомпилировать весь код в каждой виртуальной машине, или переместить окружение целиком;
- **накладные расходы.** Порядок работы, когда приложение взаимодействует с гостевой ОС, взаимодействующей с гипервизором, который, в свою очередь, взаимодействует с ОС хост-машины, управляющей оборудованием и выполняющей запросы, отличается крайней неэффективностью. Из-за накладных расходов на работу промежуточных слоев страдает общая производительность. При использовании гипервизора типа 1, непосредственно управляющего оборудованием, уменьшаются накладные расходы на взаимодействие гипервизора с операционной системой хост-машины. Однако все остальные накладные расходы никуда не исчезают и продолжают оказывать отрицательное влияние на производительность;

- **эффективность использования ресурсов.** Использование виртуальных машин, безусловно, обеспечивает более эффективное использование ресурсов, чем в случае, когда приложение устанавливается на выделенный сервер с единственной ОС, а ресурсы не используются до конца в периоды ожидания приложением каких-либо событий. Но виртуализация тоже не дает максимальной эффективности из-за затрат на переключение нескольких ОС гипервизором.

Это основные недостатки, характерные для виртуализации, однако все не так плохо, потому что технология контейнеров решает не только эти, но и другие проблемы. А теперь перейдем к ним.

Контейнеры

Контейнеры тоже предлагают виртуальную среду, в которую упаковываются процесс приложения, метаданные и файловая система, т. е. все, что необходимо приложению для работы. В отличие от виртуальных машин, контейнеры не требуют собственной операционной системы – это всего лишь обертки вокруг процессов UNIX, которые непосредственно взаимодействуют с ядром. Взгляните на рис. 5.2.

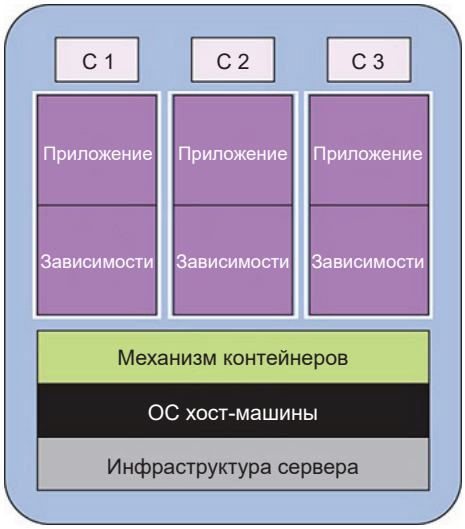


Рис. 5.2 ❖ Простая архитектура на основе контейнеров. Зависимости: дерево каталогов, библиотеки, пространство процесса и т. д.

Как можно заметить, контейнеры обеспечивают полную изоляцию приложений и процессов, когда одно приложение ничего не знает о существовании других приложений. Но все процессы используют одно и то же ядро операционной системы. Как такое возможно? Чтобы независимые процессы могли выполняться под управлением единственного экземпляра Linux, контейнеры используют средства изоляции ресурсов в ядре Linux, такие как группы управления и пространства

имен. В результате отпадает необходимость иметь отдельный экземпляр запущенной операционной системы для каждого приложения, что происходит в случае с виртуальными машинами. Это означает, что виртуальные машины обеспечивают лучшую изоляцию, чем контейнеры. Однако контейнеры оказываются более легковесными и простыми для запуска и передачи между разработчиками. Благодаря такой легковесности на данном аппаратном окружении можно запустить больше контейнеров, чем виртуальных машин. Контейнеры обеспечивают более эффективное использование аппаратных ресурсов.

Эти контейнеры также известны как *контейнеры Linux* или *LXC*. Идея контейнеров появилась достаточно давно, но лишь недавно обрела популярность благодаря Docker. Как мы уже говорили, Docker – это технология с открытым исходным кодом, внесшая несколько изменений в контейнеры Linux, чтобы сделать их более переносимыми, простыми в использовании и гибкими. С этой целью был реализован набор утилит, которые обеспечивают переносимость и гибкость контейнеров. Эти утилиты позволяют легко создавать, передавать, копировать и запускать контейнеры. Контейнеры Docker помогают преодолеть большинство недостатков виртуальных машин.

Далее перечисляются наиболее значимые различия между контейнерами Linux и Docker:

- **процессы.** В одном контейнере Linux (LXC) можно запустить несколько процессов, тогда как в контейнере Docker может выполняться только один процесс. Если приложение состоит из нескольких процессов, необходимо создать соответствующее количество контейнеров Docker. Даже при условии того, что такое решение порождает проблему управления контейнерами, оно обеспечивает огромную гибкость прикладной системы. Поскольку каждый процесс выполняется в отдельном контейнере, есть возможность управлять процессами и изменять их поведение по отдельности. Это ключевое преимущество, прекрасно согласующееся с идеей микросервисов, когда каждая служба действует автономно и в отдельном процессе;
- **постоянное хранилище.** Контейнеры Docker не имеют своего состояния, т. к. не поддерживают собственного постоянного хранилища. При необходимости вы должны сами подключить такое хранилище, смонтировав Docker;
- **переносимость.** Docker обеспечивает большую переносимость, чем LXC, что и объясняет популярность этой технологии. При использовании LXC переносимость не гарантируется, т. е. при перемещении контейнера LXC с одного хоста на другой он может работать с ошибками из-за различий между конфигурациями серверов. Docker, напротив, гарантирует беспроблемную переносимость, потому что лучше, чем LXC, изолирует приложение от таких деталей, как настройки операционной системы, сетевой подсистемы и хранилища. Закончив разработку и тестирование, программист может создать образ, который гарантированно будет работать после загрузки и запуска в эксплуатационном окружении. Это самая большая сложность, которую устраняют контейнеры Docker, упрощая жизнь инженеров.

Архитектура и компоненты Docker

В Docker используется архитектура клиент/сервер, в соответствии с которой клиент взаимодействует с демоном Docker, а тот предоставляет все необходимые клиенту услуги. Рассмотрим компоненты рабочего процесса и инструменты для управления контейнерами и их развертывания, составляющие экосистему Docker:

- **сервер, или демон Docker.** Выполняется в хост-системе и управляет всеми запущенными контейнерами;
- **контейнер Docker.** Автономная виртуальная система, содержащая выполняющийся процесс, все файлы, зависимости, адресное пространство процесса и сетевые порты, необходимые приложению. Так как каждый контейнер имеет свое пространство портов, следует организовать их отображение в фактические порты на уровне Docker. Мы еще вернемся к этому вопросу;
- **клиент Docker.** Пользовательский интерфейс, или интерфейс командной строки, для взаимодействий с демоном Docker;
- **образы Docker.** Шаблонные файлы, доступные только для чтения, с контейнером Docker. Их можно перемещать и передавать. В отличие от виртуальных машин, эти файлы можно хранить в системе управления версиями. Более того, можно воспользоваться командой `docker diff`, чтобы увидеть различия между двумя образами. Каждый образ состоит из нескольких уровней, или слоев, которые могут совместно использоваться несколькими образами. Допустим, вы обновили существующее приложение. В результате в существующий образ будет добавлен дополнительный уровень. Благодаря такой организации можно распространять и развертывать только новые уровни, упрощая и ускоряя процесс;
- **реестр Docker.** Репозиторий для хранения и распространения образов контейнеров Docker. Примером широко известного реестра может служить Docker Hub (во многом похож на GitHub), куда можно помещать и откуда можно извлекать образы. Внутри своей организации вы можете организовать свой реестр;
- **файл Dockerfile.** Это очень простой текстовый файл, содержащий команды, которые выполняют сборку образов Docker. Посредством этих команд можно устанавливать дополнительные программные компоненты, настраивать переменные окружения, рабочие каталоги и точку входа `ENTRYPOINT`, а также добавлять новый код. Подробнее о командах Docker мы поговорим в главе 7 «Интерфейс Docker»;
- **Docker Machine.** Docker Machine позволяет развертывать узлы Docker на локальной машине или внутри общедоступного либо частного облака, включая облачные системы таких поставщиков, как Amazon и Microsoft Azure. Также он обеспечивает управление узлами посредством команд `start`, `stop`, `inspect` и др. За самой свежей информацией по этой теме обращайтесь к официальной онлайн-документации Docker;
- **Docker Swarm.** По сути своей, это готовый к использованию механизм кластеризации, позволяющий объединить несколько узлов Docker в один большой хост Docker. Это отдельный инструмент, который можно установить с помощью Docker Machine или вручную, получив образ Swarm. На момент

написания этих строк он был интегрирован в Docker Engine. Процесс установки очень прост, достаточно лишь настроить диспетчер Swarm на всех узлах. Самое замечательное – это то, что мы можем просто приказать Swarm запустить наши контейнеры, а тот сам решит, на каких узлах их запустить, избавив нас от сложности выбора. Для динамической настройки и управления службами в контейнерах можно использовать службу обнаружения. Интегрированный вариант называется *режимом Swarm*. Он действует подобно инструменту Swarm, поддерживает балансировку нагрузки и обнаружение служб и может рассматриваться как полноценный механизм управления. Включается режим Swarm простой командой `init`, а добавление рабочих узлов производится командой `join`. Мы еще вернемся к Docker Swarm далее в этой книге;

- **компоновщик Docker Compose.** Приложения часто состоят из множества компонентов, и соответственно они будут выполняться в нескольких контейнерах. В состав Docker входит инструмент Compose, с помощью которого можно легко запустить приложение в нескольких контейнерах. Вы можете определить окружение для приложения в общем файле Dockerfile и определить перечень служб в файле `docker-compose.yml`, после чего Docker автоматически будет создавать и запускать необходимые контейнеры, как определено в этих файлах. Инструмент Compose так же, как Docker Machine, имеет свой набор команд для управления службами приложения.

На рис. 5.3 показана логическая организация всех этих компонентов.

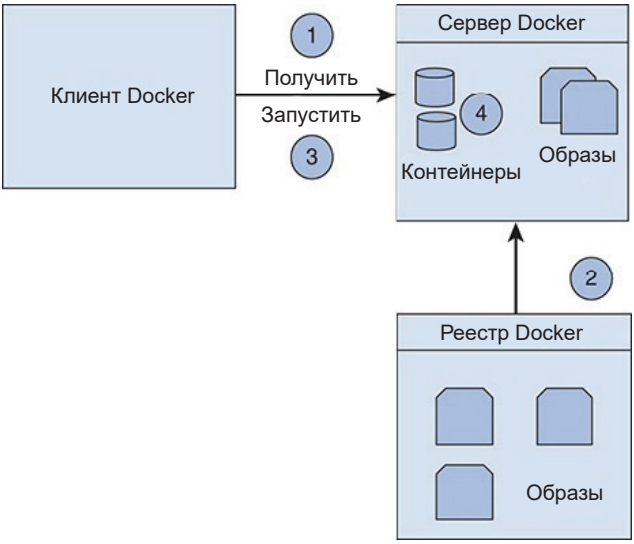


Рис. 5.3 ❖ Архитектура Docker

В последующих главах будет представлен подробный пример извлечения и запуска контейнера Docker. Но прежде рассмотрим процесс установки Docker (в главе 6) и команды (в главе 7). На данный момент важно запомнить, что контейнеры Docker представляют виртуальное окружение, а все остальные компоненты являются инструментами для управления этими контейнерами.

Итак, какие преимущества дает нам технология Docker? Она не только устраняет большинство проблем, но также сохраняет многие преимущества виртуальных машин и прекрасно укладывается в идеологию интеграции разработки и эксплуатации DevOps. Часть этих преимуществ перечисляется ниже:

- **легковесность.** У контейнеров Docker нет своей операционной системы, поэтому они имеют небольшие размеры. Кроме того, контейнеры можно хранить как образы (в виде простых файлов) и передавать их в систему управления версиями;
- **переносимость.** Контейнер Docker объединяет приложение и все его зависимости, версию ОС и т. д. Контейнер можно просто скопировать на другую машину в форме образа и запустить его без всяких проблем, т. е. образ можно собрать один раз и использовать везде, где угодно;
- **повторное использование.** Образы Docker – это простые наборы уровней и последовательности команд, объединяющие эти уровни и формирующие окончательный образ. Готовые образы Docker можно использовать как основу для создания новых образов, что позволяет ускорить сборку и уменьшить размеры образов за счет повторного использования готовых образов. Например, представьте, что у нас есть готовый образ с веб-сервером Apache, выполняющимся в Ubuntu, поверх которого добавлен слой с файлом 1. Допустим, мы решили создать другой образ – с файлом 2 поверх образа с веб-сервером Apache, выполняющимся в Ubuntu. Так как у нас уже есть первый образ, Docker повторно использует все слои из первого образа, кроме слоя с файлом 1, и на его основе создает второй образ. В результате оба образа будут совместно использовать слои с Ubuntu и Apache, и каждый будет иметь свой слой с файлом (это единственное отличие данных двух образов);
- **скорость развертывания.** Контейнеры Docker – полностью автономные пакеты, которые могут распространяться и тестироваться независимо друг от друга. Тот же самый контейнер можно развернуть в эксплуатационной среде без изменений или почти без них, что ускоряет развертывание и уменьшает вероятность ошибок из-за зависимостей в окружениях. Эта особенность очень важна для организации непрерывного развертывания;
- **эффективность использования ресурсов.** Подобно виртуальным машинам, Docker позволяет эффективно использовать аппаратные ресурсы, но и добиться еще большей эффективности благодаря легковесности контейнеров Docker. В то же время обеспечивается вполне надежная изоляция процессов друг от друга. Кроме того, учитывая малый размер, на одном хосте можно запустить большее число контейнеров.

Во многих случаях контейнеры Docker оказываются предпочтительнее виртуальных машин, но не следует заблуждаться: Docker не является заменой виртуальным машинам. На практике часто используются комбинации этих двух технологий, когда контейнеры Docker запускаются внутри виртуальных машин, что обеспечивает очень эффективное использование ресурсов.

Docker: простой пример

К настоящему моменту вы уже хотя бы теоретически понимаете широту возможностей Docker. В следующих главах вы узнаете еще больше, а пока давайте рас-

смотрим небольшой пример развертывания приложения в Docker и обсудим преимущества такого решения.

Допустим, у нас имеется простой веб-сайт на основе WordPress, состоящий из трех частей: веб-сервера со всеми приложениями WordPress, реляционной базы данных MySQL и хранилища для хранения этих данных. В мире виртуальных машин все эти части можно было бы развернуть в одной или нескольких виртуальных машинах. Для этого необходимо с помощью диспетчера создать виртуальные машины и в каждую установить операционную систему и системное программное обеспечение (MySQL, WordPress). На рис. 5.4 показана типичная организация виртуальных машин для этого случая.

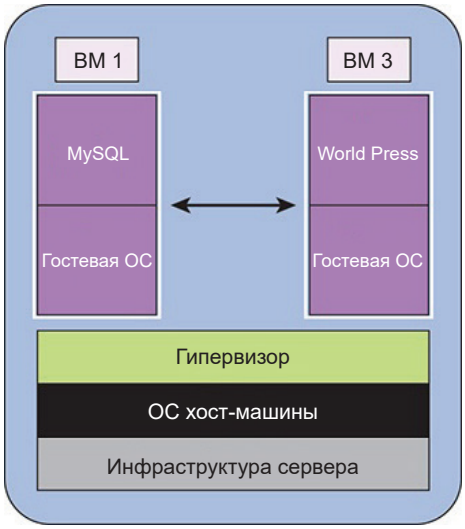


Рис. 5.4 ❖ Типичная организация виртуальных машин

Теперь попробуем представить, как те же части развернуть в контейнерах Docker с прицелом на архитектуру микросервисов. Как уже говорилось выше, это обеспечит независимость на уровне выполняемых файлов/процессов, взаимодействующих с другими службами или программами посредством стандартных механизмов межпроцессных взаимодействий.

Также обсудим, как запустить каждый процесс в отдельном контейнере Docker. При этом мы организуем совместную работу контейнеров, используя механизм связи. В этом примере мы создадим три контейнера.

Шаг 1: контейнер для данных. Самый простой вариант – получить готовый образ Ubuntu Linux из Docker Hub и запустить его. Следуя этим путем, можно создать локальное хранилище – дерево каталогов для хранения данных. Аналогично можно выделить память и ограничить объем используемых вычислительных ресурсов (CPU), выполнив следующую команду:

```
docker create --name mysql_data_container -v /var/lib/mysql ubuntu
```

Шаг 2: контейнер для MySQL. По аналогии с предыдущим шагом можно получить образ с последней версией MySQL из Docker Hub и запустить его на локальном

хосте. В команде `run` можно указать том, созданный на предыдущем шаге. Меньше чем за 2 мин мы можем установить и запустить базу данных следующей командой:

```
docker run --volumes-from mysql_data_container -v /var/lib/mysql:/var/lib/mysql -e
MYSQL_USER=mysql -e MYSQL_PASSWORD=mysql -e MYSQL_DATABASE=test -e
MYSQL_ROOT_PASSWORD=test -it -p 3306:3306 -d mysql
```

Шаг 3: контейнер для WordPress. Как и в предыдущих шагах, можно получить и запустить образ с последней версией WordPress из Docker Hub. В команде `run` можно установить связь с базой данных MySQL, созданной на предыдущем шаге:

```
docker run -d --name wordpress --link mysql:mysql wordpress
```

Меньше чем за 10 мин мы создали персональный веб-сайт на основе WordPress, структура которого изображена на рис. 5.5.

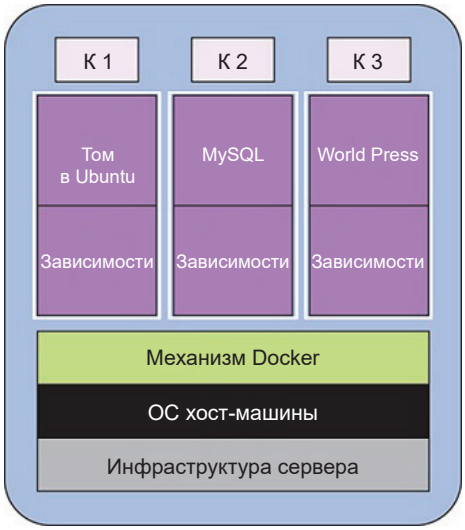


Рис. 5.5 ❖ Как создать веб-сайт на основе WordPress, действующий в трех контейнерах (К – контейнер)

Сравнив рис. 5.4 и 5.5, можно понять, почему контейнеры так легковесны. Им не требуются свои собственные операционные системы. Эта легковесность упрощает решение задач сопровождения и масштабирования:

- **простота обновления.** Представьте, что вы решили обновить образ с MySQL. Для этого вам потребуется всего лишь остановить контейнер с MySQL, запущенный на шаге 2, получить образ с последней версией MySQL и запустить его, подключив тот же самый том;
- **повторное использование.** Представьте, что вам потребовалось настроить и запустить второй экземпляр WordPress для каких-то специфических нужд. Вы можете взять другой образ WordPress и запустить еще один контейнер Docker, связав его с той же базой данных;
- **простота кластеризации.** В Docker имеется встроенный механизм кластеризации, который называется *режимом Swarm*. Выполнив всего несколько

команд, можно организовать кластер, балансировку нагрузки и обнаружение ваших служб. Более подробно этот вопрос будет рассматриваться в следующих главах.

Добро пожаловать в мир контейнеров! В этой области происходит много интересного, и сообщество Docker очень быстро движется вперед, практически ежедневно добавляя новые возможности. Кроме того, появилось множество проектов, целью которых является решение некоторых проблем и создание средств автоматизации. В следующих главах мы, конечно, больше узнаем о контейнерах Docker, но вы обязательно должны добавить в закладки ссылку на страницу сообщества Docker, https://www.docker.com/docker-community, чтобы держать руку на пульсе.

Глава 6

Установка Docker

Еще примерно год тому назад установить Docker было очень непросто, но теперь эта проблема исчезла. Docker базируется на технологиях Linux, и большинство основных дистрибутивов Linux, таких как Centos, Ubuntu и Amazon Linux, поддерживает Docker «из коробки». В этой главе мы рассмотрим установку Docker в Mac OS X, Windows и Ubuntu Linux.

Установка Docker в Mac OS X

Следующие инструкции описывают процесс установки в версию Mac OS X 10.11 или выше. Для проверки версии Mac OS выберите в меню **Apple** пункт **About This Mac** (Об этом Mac). Далее мы загрузим и установим последнюю на момент написания книги версию Docker 17.03.0:

1) введите в адресную строку браузера URL: https://docs.docker.com/docker-for-mac/install/#download-docker-for-mac. Щелкните на ссылке **Get Docker for Mac (Stable)** (Получить Docker для Mac (стабильная версия)), чтобы загрузить набор инструментов Docker в папку `Downloads` (Загрузки);
2) по завершении загрузки откройте папку `Downloads` (Загрузки) и щелкните дважды на файле пакета, чтобы открыть его. В результате должно появиться окно, изображенное на рис. 6.1;

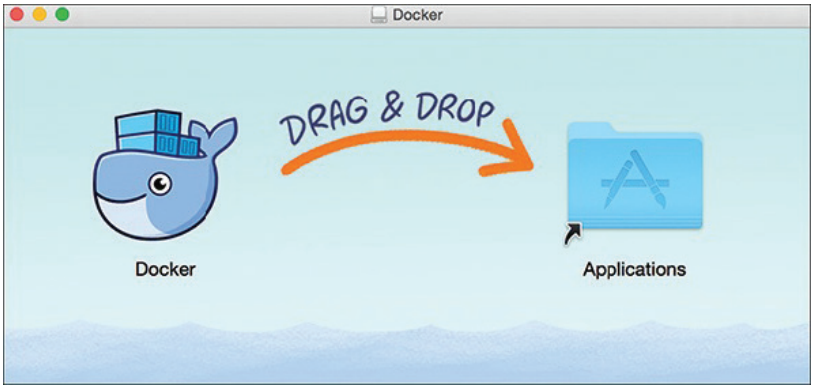


Рис. 6.1 ❖ Окно мастера установки

3) перетащите мышью значок **Docker** с изображением кита в папку `Applications`, чтобы переместить приложение Docker на свой компьютер, как показано на рис. 6.2;

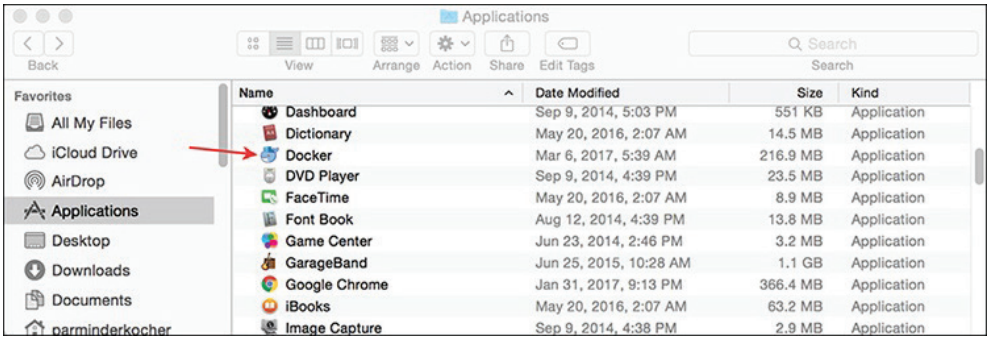


Рис. 6.2 ❖ Папка `Applications` с установленным приложением Docker

4) дважды щелкните на приложении Docker, чтобы запустить его. Вы увидите диалог, изображенный на рис. 6.3, с просьбой дать приложению привилегированный доступ. Щелкните на кнопке **OK**;

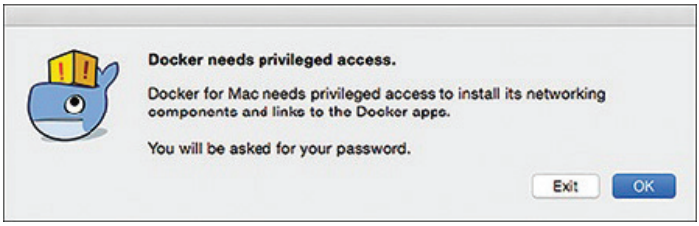


Рис. 6.3 ❖ Разрешение привилегированного доступа

5) после этого появится еще один диалог, предлагающий ввести пароль, используемый при входе в систему. Введите свой пароль;
6) если в вашей системе уже был установлен Docker, вы увидите диалог, изображенный на рис. 6.4, предлагающий скопировать имеющиеся у вас образы и контейнеры Docker. Щелкните на кнопке **Copy** (Копировать), чтобы скопировать имеющиеся образы. Если прежние образы вам не нужны, щелкните на кнопке **No** (Нет). Если Docker устанавливается впервые, этот диалог не появится;

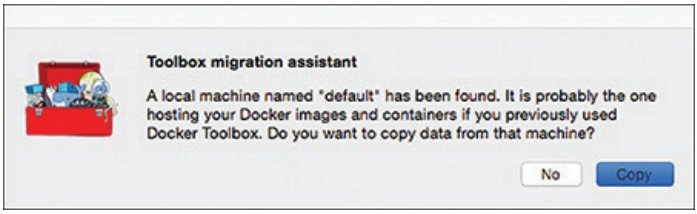


Рис. 6.4 ❖ Диалог, который появляется, только в том случае, если в системе уже имеется ранее установленная версия Docker

7) после этого начнется процесс установки, по завершении которого произойдет автоматический запуск Docker, как показано на рис. 6.5.

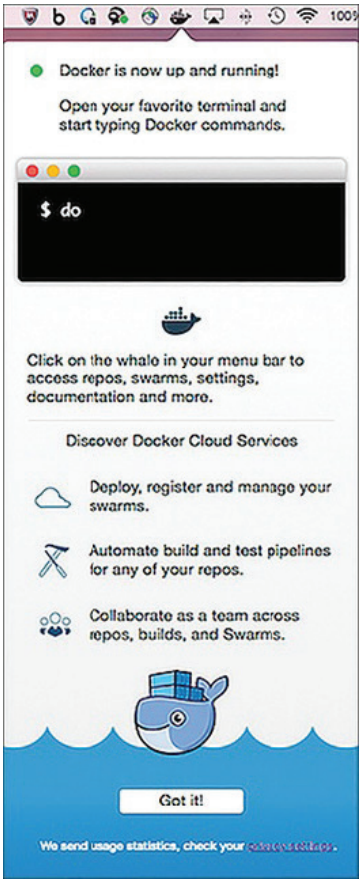


Рис. 6.5 ❖ Диалог, сообщающий об окончании установки

Вот и все – установка Docker завершена.

После установки вам дается возможность зарегистрироваться в Docker Hub. Напомню, что Docker Hub – это облачная служба реестра для хранения и распространения образов Docker с общедоступным разделом, который можно использовать для распространения своих образов Docker. Также можно приобрести платную подписку и настроить свой приватный раздел с ограниченным доступом.

Если вы уже зарегистрированы, введите имя пользователя и пароль (или просто пропустите этот шаг). После входа вы попадете в репозиторий Docker Hub, где сможете просмотреть список общедоступных образов Docker или начать создавать свои образы, как показано на рис. 6.6.

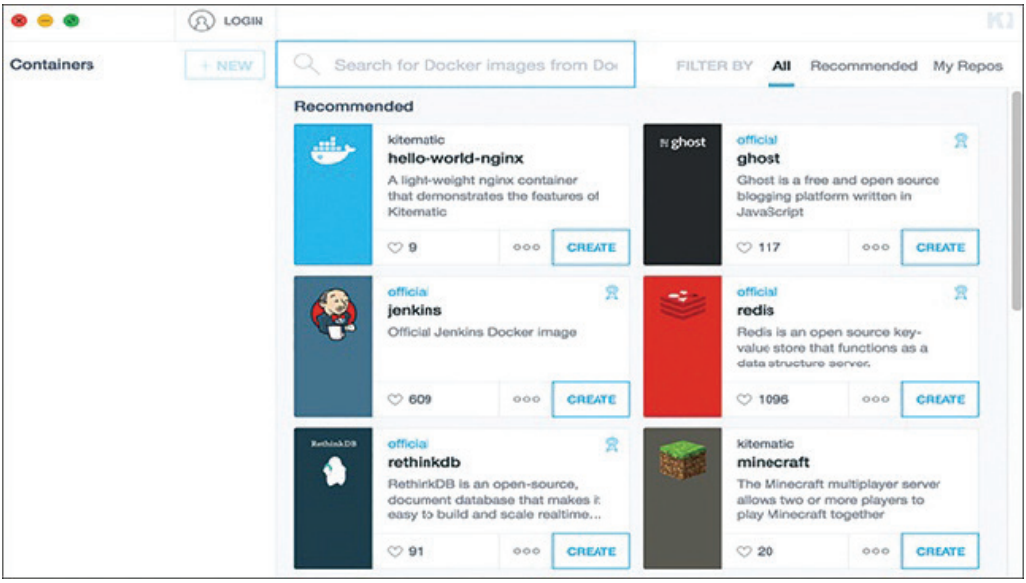


Рис. 6.6 ❖ Главная страница Docker Hub

В следующей главе мы ближе познакомимся со всеми командами Docker, а пока просто попробуем воспользоваться некоторыми из них. Сначала проверим установленную версию Docker и поэкспериментируем с интерфейсом командной строки Docker.

Откройте терминал и выполните команду `docker --version`, чтобы проверить установленную версию Docker, как показано на рис. 6.7.

```
PKOCHER-M-343X:~ parminderkocher$ docker --version
Docker version 17.03.0-ce, build 60ccb22
PKOCHER-M-343X:~ parminderkocher$
```

Рис. 6.7 ❖ Вывод версии Docker

Как показывает вывод команды, мы установили версию Docker 17.03.0.

Чтобы получить список поддерживаемых команд, выполните команду `docker --help`. Вы должны увидеть все доступные команды, как показано на рис. 6.8.

```
PKOCHER-M-343X:~ parminderkochers docker  --help

Usage:          docker COMMAND

A self-sufficient runtime for containers

Options:
      --config string          Location of client config files (default "/Users/par
                               minderkocher/.docker")
  -D, --debug                  Enable debug mode
      --help                   Print usage
  -H, --host list              Daemon socket(s) to connect to (default [])
  -l, --log-level string       Set the logging level ("debug", "info", "warn", "error",
                               "fatal") (default "info")
      --tls                    Use TLS; implied by --tlsverify
      --tlscacert string       Trust certs signed only by this CA (default "/Users/
                               parminderkocher/.docker/ca.pem")
      --tlscert string         Path to TLS certificate file (default "/Users/parmin
                               derkocher/.docker/cert.pem")
      --tlskey string          Path to TLS key file (default "/Users/parminderkocher
                               /.docker/key.pem")
      --tlsverify              Use TLS and verify the remote
  -v, --version                Print version information and quit

Management Commands:
  checkpoint   Manage checkpoints
  container    Manage containers
  image        Manage images
  network      Manage networks
  node         Manage Swarm nodes
  plugin       Manage plugins
  secret       Manage Docker secrets
  service      Manage services
  stack        Manage Docker stacks
  swarm        Manage Swarm
  system       Manage Docker
  volume       Manage volumes

Commands:
  attach       Attach to a running container
  build        Build an image from a Dockerfile
  commit       Create a new image from a container's changes
  cp           Copy files/folders between a container and the local filesystem
  create       Create a new container
  deploy       Deploy a new stack or update an existing stack
  diff         Inspect changes to files or directories on a container's filesystem
  events       Get real time events from the server
  exec         Run a command in a running container
  export       Export a container's filesystem as a tar archive
```

Рис. 6.8 ❖ Доступные команды Docker

Установка Docker в Windows

Следующие инструкции предполагают, что установка производится в 64-разрядной версии Windows 10 редакции Pro, Enterprise или Education. Для правильной установки Docker необходимо также включить поддержку виртуализации Hyper-V. За более подробной информацией обращайтесь в справочный раздел на сайте с документацией Docker: https://docs.docker.com/docker-for-windows/install/#download-docker-for-windows. Далее мы загрузим и установим версию Docker 17.03.0. Это была самая последняя версия на момент написания данных строк:

1) введите в адресную строку браузера URL: https://docs.docker.com/docker-for-windows/install/#download-docker-for-windows. Щелкните на ссылке **Get Docker for Windows (Stable)** (Получить Docker для Windows (стабильная версия)), чтобы загрузить набор инструментов Docker в папку `Downloads` (Загрузки);
2) по завершении загрузки откройте папку `Downloads` (Загрузки) и щелкните дважды на файле пакета, чтобы открыть его. В результате должно появиться окно, предлагающее подтвердить согласие с лицензионным соглашением, как показано на рис. 6.9;

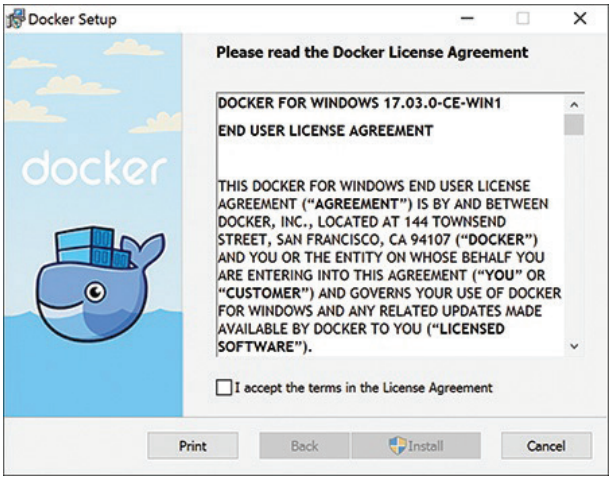


Рис. 6.9 ❖ Лицензионное соглашение Docker

3) примите условия лицензии, установив флажок внизу диалога, и затем щелкните на кнопке **Install** (Установить), чтобы установить Docker на свой компьютер;
4) после установки справа внизу на экране должно появиться всплывающее сообщение **Docker is starting** (Запуск Docker). По окончании запуска вы увидите диалог, изображенный на рис. 6.10. На этом установку можно считать законченной.

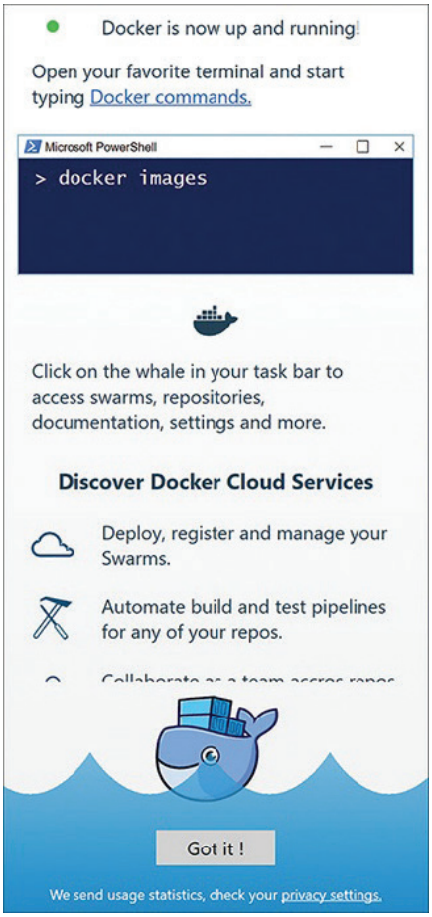


Рис. 6.10 ❖ Диалог, сообщающий о завершении установки

Вот и все – установка Docker завершена.

В следующей главе мы ближе познакомимся со всеми командами Docker, а пока просто попробуем воспользоваться некоторыми из них. Сначала проверим установленную версию Docker и поэкспериментируем с интерфейсом командной строки Docker.

Откройте терминал и выполните команду `docker --version`, чтобы проверить установленную версию Docker, как показано на рис. 6.11.

Рис. 6.11 ❖ Вывод версии Docker

Как показывает вывод команды, мы установили версию Docker 17.03.0.

Чтобы получить список поддерживаемых команд, выполните команду `docker --help`. Вы должны увидеть все доступные команды, как было показано выше (на рис. 6.8).

Установка Docker в Ubuntu Linux

Далее мы загрузим и установим версию Docker 17.03.0. Это была самая последняя версия на момент написания данных строк. Самую свежую информацию о последней доступной версии вы найдете по адресу: https://docs.docker.com/engine/installation/linux/ubuntu/#install-using-the-repository. Если вы пользуетесь другим дистрибутивом Linux, все равно обращайтесь по этому адресу.

Следующие инструкции описывают процесс установки в одну из следующих 64-разрядных версий Ubuntu:

- Trusty 14.04;
- Yakkety 16.10;
- Xenial 16.04.

Для проверки версии операционной системы выполните команду `lsb_release -a`, как показано на рис. 6.12.

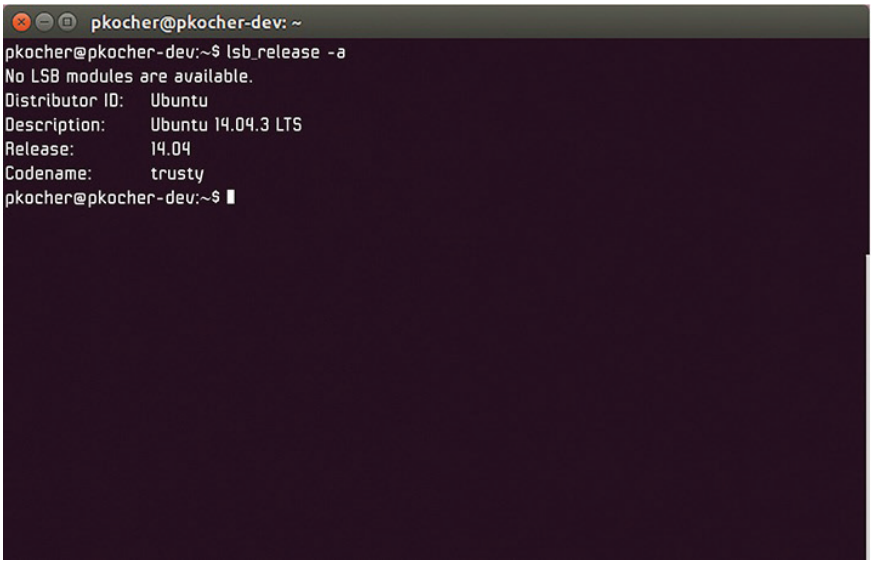


Рис. 6.12 ❖ Проверка версии Ubuntu

Ниже описываются шаги по установке Docker в Linux впервые. Далее описываются шаги по установке в Trusty 14.04, но они подходят также для других двух версий.

Если вы используете Trusty 14.04, рекомендуется установить пакеты `linux-image-extra -*`, если они еще не установлены. Эти пакеты позволят Docker использовать драйверы AUFS устройств хранения, которые по умолчанию использует Docker

в Ubuntu. (В других дистрибутивах по умолчанию используется Device Mapper.) Для установки пакетов выполните следующую команду:

```
$ sudo apt-get update
```

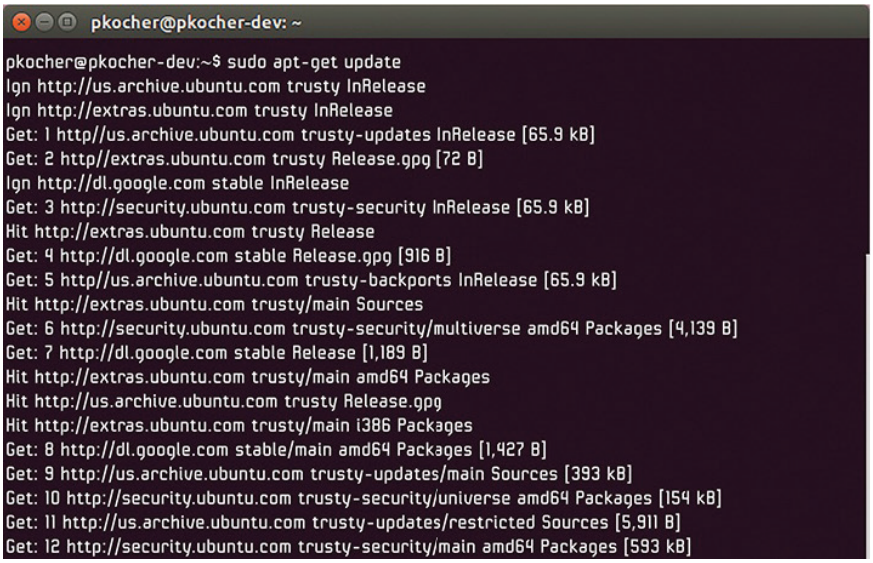


Рис. 6.13 ❖ Установка дополнительных пакетов

Предыдущая команда загрузит последние версии пакетов, как показано на рис. 6.13, и вы будете готовы установить их. Для этого выполните следующую команду:

```
$ sudo apt-get install \
    linux-image-extra-$(uname -r) \
    linux-image-extra-virtual
```

Эта команда установит загруженные обновления, после чего можно начинать установку Docker. Для установки доступны две редакции: Docker CE (Community Edition) и Docker EE (Enterprise Edition). Мы установим редакцию CE:

1) сначала нужно установить репозиторий Docker, откуда потом можно выполнить установку. Для этого следующей командой установите пакет, необходимый для доступа к репозиторию через HTTPS:

   ```
   $ sudo apt-get install \
     apt-transport-https \
     ca-certificates \
     curl \ software-properties-common
   ```

2) добавьте ключ GPG для доступа к официальному репозиторию Docker:

   ```
   $ curl -fsSL https://download.Docker.com/linux/ubuntu/gpg | sudo apt-key add
   ```

3) проверьте контрольную сумму ключа, она должна быть следующей: 9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88 (см. рис. 6.14):

```
$ sudo apt-key fingerprint 0EBFCD88
```

```
pub     4096R/0EBFCD88 2017-02-22
        Key fingerprint = 9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88
uid                     Docker Release (CE deb) <docker@docker.com>
sub     4096R/F273FCD8 2017-02-22
pkocher@pkocher-dev:~$
```

Рис. 6.14 ❖ Проверка контрольной суммы ключа

4) добавьте репозиторий Docker в источники APT (Advanced Packaging Tool):

```
$ sudo add-apt-repository "deb [arch=amd64] <-DOCKER-EE-URL> \
  $(lsb_release -cs) \ stable-"
```

5) обновите индекс пакетов после добавления репозитория:

```
$ sudo apt-get update
```

6) установите последнюю версию Docker (см. рис. 6.15):

```
$ sudo apt-get install Docker-ce
```

```
pkocher@pkocher-dev: ~
Unpacking  liberror-perl (0.17-1.1) ...
Selecting  previously unselected package git-man.
Preparing  to unpack ... /git-man_1%3a1.9.1-1ubuntu0.3_all.deb ...
Unpacking  git-man (1:1.9.1-1ubuntu0.3) ...
Selecting  previously unselected package git.
Preparing  to unpack ... /git_1%3a1.9.1-1ubuntu0.3_amd64.deb...
Unpacking  git (1:1.9.1-1ubuntu0.3) ...
Selecting  previously unselected package cgroup-lite.
Preparing  to unpack .../cgroup-lite_1.9_all.deb ...
Unpacking  cgroup-lite (1.9) ...
Processing  triggers for man-db (2.6.7.1-1ubuntu1) ...
Processing  triggers for ureadahead (0.100.0-16) ...
ureadahead will be reprofiled on next reboot
Setting  up  aufs-tools (1:3.2+20130722-1.1) ...
setting  up  docker-ce (17.03.0~ce-0~ubuntu-trusty) ...
docker start/running, process 31184
Setting  up  liberror-perl (0.17-1.1) ...
Setting  up  git-man (1:1.9.1-1ubuntu0.3) ...
Setting  up  git (1:1.9.1-1ubuntu0.3) ...
Setting  up  cgroup-lite (1.9) ...
cgroup-lite  start/running
Processing  triggers for libc-bin (2.19-0ubuntu6.6) ...
Processing  triggers for ureadahead (0.100.0-16) ...
pkocher@pkocher-dev:~$
```

Рис. 6.15 ❖ Установка последней версии Docker

7) проверьте установленную версию, как показано на рис. 6.16:

```
$ docker --version
```

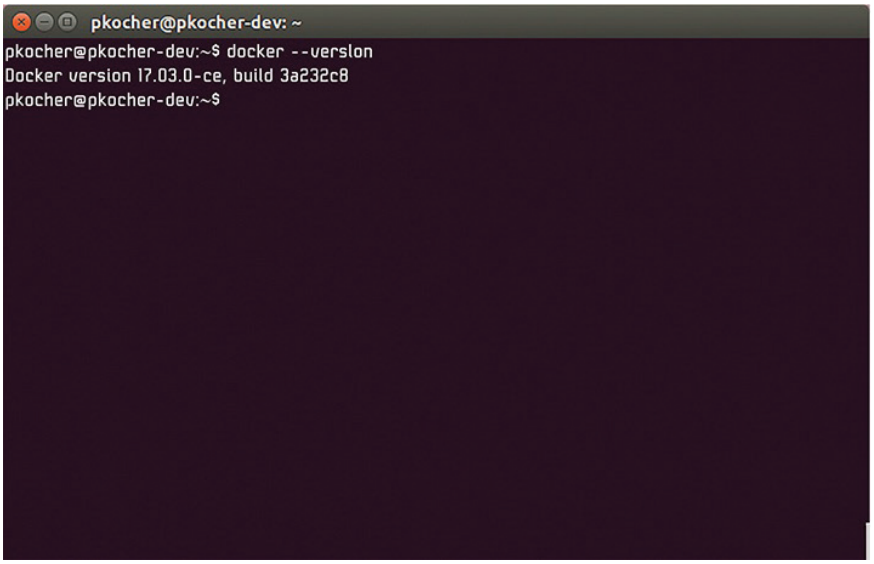


Рис. 6.16 ❖ Вывод версии Docker

Вот и все – установка Docker в Ubuntu Linux завершена.

Глава 7

Интерфейс Docker

В главе 5 «Контейнеры Docker» упоминался файл Dockerfile, содержащий последовательность команд, выполняемых демоном Docker. В этой главе мы познакомимся с некоторыми из этих команд, которые используются наиболее часто. Затем мы создадим файл Dockerfile, выполним его и обсудим получившиеся результаты.

Основные команды Docker

Следующий сборник команд можно рассматривать как своеобразную Библию, знать которую совершенно необходимо для успешного применения Docker (от поиска и сборки образов до создания своих файлов Dockerfile). Сначала мы рассмотрим наиболее простые команды, а затем, опираясь на вновь обретенные знания, перейдем к изучению более сложных.

docker search

Команду `docker search` можно использовать для поиска образов, доступных в реестре Docker:

```
docker search [options] term
```

Графический интерфейс Docker также поддерживает возможность поиска.

В примере, изображенном на рис. 7.1, показано действие команды `docker search mysql`, которая вернула все образы, содержащие «mysql» в названиях. Как видите, она вернула первые 25 результатов. Поиск в графическом интерфейсе дает те же результаты, как показано на рис. 7.2.

Некоторые результаты, такие как `dockerizedrupal`, встречаются один раз, но некоторые повторяются, потому что образы с повторяющимися названиями были созданы и выгружены разными пользователями для каких-то своих специфических нужд. Если выполнить команду `search` с ключом `-s`, она вернет только наиболее часто используемые образы, опираясь на отзывы других пользователей:

```
docker search -s 50 mysql
```

Эта команда вернет все образы, имеющие слово «mysql» в названии и получившие не менее 50 звезд по отзывам пользователей, как показано на рис. 7.3.

```
Parminders-MacBook-Pro:~ parminderkochers docker Search mysql
NAME                      DESCRIPTION                                     STARS    OFFICIAL    AUTOMATED
mysql                     MySQL is a widely used, open-source relati...   1064     [OK]
mysql/mysql-server        Optimized MySQL Server Docker images. Crea...   41                   [OK]
orchardup/mysql                                                           41                   [OK]
centurylink/mysql         Image containing mysql. Optimized to be li...   27                   [OK]
wnameless/mysql-phpmyadmin MySQL + phpMyAdmin https://index.docker.io...  23                   [OK]
sameersbn/mysql                                                           20                   [OK]
google/mysql              MySQL server for Google Compute Engine          13                   [OK]
loggstream/mysql          MySQL Image with Master-Slave replication       5                    [OK]
appcontainers/mysql       CentOS 6.7 based Customizible MySQL 5.5 Co...   5                    [OK]
marvambass/mysql          MySQL Server based on Ubuntu 14.04              3                    [OK]
jdeathe/centos-ssh-mysql  CentOS-6 6.6 x86_64 / MySQL.                    2                    [OK]
azukiapp/mysql            Docker image to run MySQL by Azuki - http:...   2                    [OK]
frodenas/mysql            A Docker Image for MySQL                        1                    [OK]
ibourgeois/mysql          MySQL image from ibourgeois/base                1                    [OK]
bahmni/mysql              Mysql container for bahmni. Contains the...     1                    [OK]
phpmentors/mysql          MySQL server image                              1                    [OK]
jmoati/mysql                                                              0                    [OK]
guihatano/mysql           MySQL Server on Ubuntu 14.04                    0                    [OK]
lancehudson/docker-mysql  MySQL is a widely used, open-source relati...   0                    [OK]
tetraweb/mysql                                                            0                    [OK]
vkyii/mysql               mysql base on alpine                            0                    [OK]
wenzizone/mysql           mysql                                           0                    [OK]
dockerizedrupal/mysql     docker-mysql                                    0                    [OK]
javl3r/mysql              mysql                                           0                    [OK]
ahmet2mir/mysql           This is a Debian based image with MySQL se...   0                    [OK]
Parminders-MacBook-Pro:~ paraminderkochers
```

Рис. 7.1 ❖ Результат выполнения команды docker search mysql

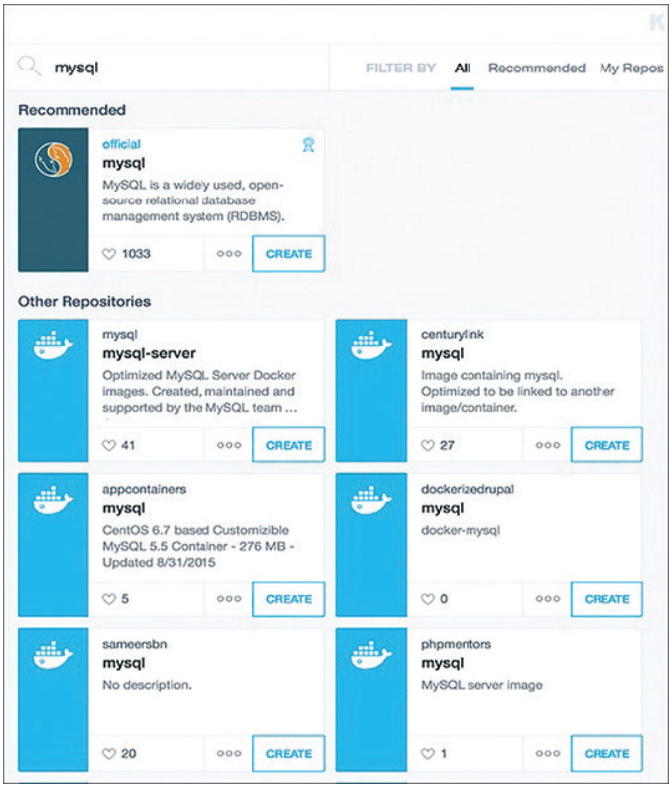


Рис. 7.2 ❖ Результаты поиска по слову «mysql» в графическом интерфейсе Docker

```
Parminders-MacBook-Pro:~ parminderkochers docker search -s 50 mysql
NAME        DESCRIPTION                                     STARS    OFFICIAL    AUTOMATED
mysql       MySQL is a widely used, open-source relati...   1044     [OK]
mariadb     MariaDB is a community-developed fork of M...   214      [OK]
Parminders-MacBook-Pro:~ paraminderkochers
```

Рис. 7.3 ❖ Результаты поиска по слову «mysql» образов, имеющих не менее 50 звезд

На этот раз команда вернула только два результата, потому что только два образа получили рейтинг выше 50.

> **Примечание.** Развитие Docker продолжается высокими темпами, поэтому набор команд, их ключи и даже особенности работы могут изменяться с каждой новой версией. Например, когда я писал эту книгу, ключ `-s` команды `search` был объявлен устаревшим, а вместо него рекомендовалось использовать ключ `--filter`. Таким образом, команда поиска всех образов MySQL с рейтингом выше 50, использующая ключ `--filter`, будет выглядеть так:
>
> ```
> docker search --filter stars=50 mysql
> ```

docker pull

Команда `docker pull` загружает указанный образ из реестра Docker на локальный компьютер:

```
docker pull image:tag
```

Например, как показано на рис. 7.4, команда `docker pull MySQL` загрузит образ MySQL. Если тег (`tag`, обозначающий версию) не указан, команда подставит тег `"latest"` и загрузит только последнюю версию образа MySQL. В этом случае она будет действовать подобно команде

```
docker pull MySQL:latest
```

```
Parminders-MacBook-Pro:~ parminderkochers docker pull mysql
Using default tag: latest
latest: Pulling from library/mysql

ba249489d0b6: Pull complete
19de96c112fc: Pull complete
2e32b26a94ed: Pull complete
637386aea7a0: Pull complete
f40aa7fe5d68: Pull complete
ca21348f3728: Pull complete
b783bc3b44b9: Pull complete
f94304dc94e3: Pull complete
efb904a945ff: Pull complete
64ef882b700f: Pull complete
291b704c92b1: Pull complete
adfeb78ac4de: Pull complete
f27e5410cda3: Pull complete
ca4b92f905b9: Pull complete
065018fec3d7: Pull complete
6762f304c834: Pull complete
library/mysql: latest: The image you are pulling has been verified. Important: image verification is a
tech preview feature and should not be relied on to provide security

Digest: sha256:842ee1ad1b0f19561d9fee65bb7c6197b2a2b4093f069e7969acefb6355e8c1b
Status: Downloaded newer image for mysql: latest
Parminders-MacBook-Pro:~ parminderkochers
```

Рис. 7.4 ❖ Команда `docker pull` загрузит из реестра последнюю версию образа MySQL

docker images

Команда `docker images` возвращает список образов верхнего уровня, доступных на локальном компьютере:

```
docker images [options]
```

Например, команда `docker images -a` выведет список всех образов верхнего уровня, включающий название репозитория, тег, дату создания и виртуальный размер, как показано на рис. 7.5. Она не отображает образы промежуточных уровней.

```
Parminders-MacBook-Pro:~ parminderkochers$ docker images -a
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
ubuntu              latest              91e54cfb1179        4 weeks ago         188.4 MB
<none>              <none>              d74508fb6632        4 weeks ago         188.4 MB
<none>              <none>              c22013c84729        4 weeks ago         188.4 MB
<none>              <none>              d3a1f33e8a5a        4 weeks ago         188.2 MB
Parminders-MacBook-Pro:~ parminderkochers$
```

Рис. 7.5 ❖ Команда `docker images` возвращает список всех образов верхнего уровня с сопутствующей информацией

Важно помнить, что при создании или сборке образов Docker на локальном компьютере дополнительно создаются различные промежуточные уровни. Например, если используемый нами файл Dockerfile имеет несколько команд, выполняющих сборку образов, каждая из них обеспечивает дополнительный уровень и образ для него. Это один из ключевых аспектов Docker, который делает контейнеры идеально подходящими для повторного использования.

docker rmi

Команда `docker rmi` удаляет указанный образ или образы на локальном компьютере:

```
docker rmi [options] image [image, image...]
```

Например, команда `docker rmi MySQL`, представленная на рис. 7.6, удалит образ MySQL, а также все уровни, установленные с хоста.

```
Parminders-MacBook-Fro:~ parminderkochers$ docker rmi mysql
Untagged: mysql:latest
Deleted: 6762f304c83428bf1945e9ab0aa05119a8a758d33d93eca50ba03665a89b5d97
Deleted: 065018fec3d7c28754f0d40a3cid56f103996a49f2995fde8c79edlbd524a9d0
Deleted: ca4b92f905b922ee6d5faf8f21592a4e8fb16a56fce47447c58c0c9356243384
Deleted: f27e5410cda3728deb33a884fda066d826c0b9bd0268ea9990ab6754f979ac3a
Deleted: adfeb78ac4de9f11124e4585a62bb9a5bfbb7e1686b4f2977106dff8626806c9
Deleted: 291b704c92b15a350ac3be00279a251b7038826cf9253047b594bfc1c50bd82b
Deleted: 64ef882b700fb8ad04e843e28ea56552265519925f3ceafb1a187c49cf27e2df
Deleted: efb904a945ff1ab48b1a03f5052a0d0ef3365e38436f0f3dd581d4c77854e1a3
Deleted: f94304dc94e325bb13db37589878 0bec04fc83362381d6b8476ab288287e5d9a
Deleted: b783bc3b44b9b8cd7b781bc86133ad490e3b7b1dca740a4df3e365843cbe5a5a
Deleted: ca21348f372879b0b48ccc5a7e7ce8c97da42f1339b86ec893223 1c15bd548be
Deleted: f40aa7fe5d68f46e6ae72ffla2808c95411f773d140d986506f352b90e412171
Deleted: 637386aea7a0d378aef7c4213300cab50d0ccbbe8ddb0bad18620f5ce73d0c53
Deleted: 2e32b26a94eda87d141712d27037a22abc0fa0cbc5b924e4f6870d5dc207f0d3
Deleted: 19de96c112fcca5b6de16611dc0a359b0b977c551921ca79ac5cf4a8bfff9351
Deleted: ba249489d0b6512128b60a4910e78fa2000c785d59e0599188a6802bd01155f2
Parminders-MacBook-Fro:~ parminderkochers$
Parminders-MacBook-Fro:~ parminderkochers$
Parminders-MacBook-Fro:~ parminderkochers$ docker images
REPOSITORY              TAG             IMAGE ID        CREATED         VIRTUAL SIZE
Parminders-MacBook-Fro:~ parminderkochers$ ▮
```

Рис. 7.6 ❖ Удаление образа MySQL командой `docker rmi`

docker run

Следующий логичный шаг после загрузки образа (командой `pull`) – его запуск. Именно это делает команда `docker run`:

```
docker run [options] image: tag [command, args]
```

Данная команда разворачивает контейнер в его собственной файловой системе, имеющей свой набор портов и IP-адрес. Кроме названия образа, команде `run` можно также передать дополнительные ключи и аргументы. Вот наиболее часто используемые из них:

- `i` переключает команду в интерактивный режим и открывает STDIN;
- `t` создает псевдотерминал tty.

Команда `docker run` поддерживает множество других ключей, например для запуска процесса в фоновом режиме (`-d`), когда контейнер запускается без поддержки командной строки. Также можно переопределить являющиеся частью запускаемого образа команды по умолчанию. Дополнительно можно задать ограничения на объем памяти и количество доступных процессоров.

Для примера давайте загрузим образ Ubuntu и выполним команду `run` (см. рис. 7.7):

```
docker pull ubuntu:latest
```

```
Parminders-MacBook-Pro:~ parminderkocher$ docker pull ubuntu: latest
latest: Pulling from library/ubuntu

d3alf33e8a5a : Pull complete
c22013c84729 : Pull complete
d74508fb6632: Pull complete
91e54dfb1179  : Pull complete
library/ubuntu: latest: The image you are pulling has been verified. Important: image verification is a tech preview feature and
should not be relied on to provide security.

Digest: sha256:73fbe2308f5f5cb6e343425831b8ab44f10bbd7707ecdfbe4081daa4cbe3ed1
Status: Downloaded newer image for ubuntu: latest
Parminders-MacBook-Pro:~ parminderkocher$ ▮
```

Рис. 7.7 ❖ Загрузка образа Ubuntu из репозитория Docker Hub

Эта команда загрузит образ Ubuntu на локальный компьютер, в чем можно убедиться, выполнив команду `docker images -a`, как показано на рис. 7.8.

```
Parminders-MacBook-Pro:~ parminderkocher$ docker images -a
REPOSITORY        TAG             IMAGE ID        CREATED         VIRTUAL SIZE
ubuntu            latest          91e54dfb1179    4 weeks ago     188.4 MB
<none>            <none>          d74508fb6632    4 weeks ago     188.4 MB
<none>            <none>          c22013c84729    4 weeks ago     188.4 MB
<none>            <none>          d3alf33e8a5a    4 weeks ago     188.2 MB
Parminders-MacBook-Pro:~ parminderkocher$ ▮
```

Рис. 7.8 ❖ Была загружена последняя версия образа Ubuntu

Теперь запустим этот образ на локальном компьютере с ключами `i` и `t`. Дополнительно потребуем запустить процесс командной оболочки:

```
docker run -it ubuntu sh
```

Сейчас на нашем локальном компьютере выполняется контейнер Ubuntu, и в окне терминала отображается приглашение к вводу. Вы можете выполнить любые команды в этом окне, какие пожелаете. В качестве примера на рис. 7.9 показаны результаты выполнения простых команд:

- `echo 'Learning Docker'`;
- `ls`;
- `cd bin` (чтобы вывести содержимое каталога `bin`).

```
Parminders-MacBook-Pro:~ parminderkochers docker run -it ubuntu sh
# echo 'Learning Docker';
Learning Docker
# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
# cd bin
# ls
bash        chgrp       dumpkeys  kill      mknod       openvt                  sed         true        zfgrep
bunzip2     chmod       echo      kmod      mktemp      pidof                   setfont     udevadm     zforce
bzcat       chown       egrep     less      more        ping                    setupcon    umount      zgrep
bzcmp       chvt        false     lessecho  mount       ping6                   sh          uname       zless
bzdiff      cp          fgconsole lessfile  mountpoint  plymouth                sh.distrib  uncompress  zmore
bzegrep     cpio        fgrep     lesskey   mt          plymouth-upstart-bridge sleep       unicode_start znew
bzexe       dash        findmnt   lesspipe  mt-gnu      ps                      ss          vdir
bzfgrep     date        grep      ln        mv          pwd                     stty        which
bzgrep      dd          gunzip    loadkeys  nc          rbash                   su          whiptail
bzip2       df          gzexe     login     nc.openbsd  readlink                sync        ypdomainname
bzip2recover dir        gzip      ls        netcat      rm                      tailf       zcat
bzless      dmesg       hostname  lsblk     netstat     rmdir                   tar         zcmp
bzmore      dnsdomainname ip      lsmod     nisdomainname run-parts             tempfile    zdiff
cat         domainname  kbd_mode  mkdir     open        running-in-container    touch       zegrep
#
```

Рис. 7.9 ❖ Выполнение команд в интерактивной командной оболочке

Как видите, каталог `bin` содержит все основные программы, необходимые системе для работы.

docker ps

Команда `docker ps` выводит список всех контейнеров, запущенных к настоящему моменту, как показано на рис. 7.10:

```
docker ps [options]
```

Напомню, что в каждом контейнере выполняется только один процесс. В данном примере нет ни одного запущенного контейнера, поэтому список пуст.

```
Parminders-MacBook-Pro:~ parminderkochers docker ps
CONTAINER  ID        IMAGE        COMMAND        CREATED        STATUS        PORTS        NAMES
Parminders-MacBook-Pro :~ parminderkochers
```

Рис. 7.10 ❖ Команда `docker ps` выводит список всех запущенных контейнеров

Попробуем выполнить команду `ps` с ключом `-a`, как показано на рис. 7.11, чтобы увидеть список всех контейнеров, даже не запущенных.

```
Parminders-MacBook-Pro:~ parminderkochers docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                      PORTS
 NAMES
c8b977Oc88e9        ubuntu              "sh"                5 minutes ago       Exited (0) 3 minutes ago
 admiring_albattani
Parminders-MacBook-Pro:~ parminderkochers docker restart c8b977Oc88e9
c8b977Oc88e9
```

Рис. 7.11 ❖ Ключ `-a` добавляет в список неактивные контейнеры

Как видите, после выхода из командной оболочки контейнер Ubuntu был остановлен и стал неактивным, но при этом он не был удален. При необходимости его можно повторно запустить, как показано ниже.

docker logs

Команда `docker logs` выводит из указанного контейнера файлы журналов, содержащие стандартный вывод (`stdout` и `stderr`) контейнера:

```
docker logs [options] container
```

Эта команда доступна только для контейнеров с драйвером журналирования JSON File.

Например, выполните следующую команду, чтобы запустить процесс командной оболочки:

```
docker run -it ubuntu sh
```

Выполните пару команд оболочки (например, `ls -a` и `cd bin`), как показано на рис. 7.12.

```
PKOCHER-M-343X:~ parminderkochers docker run -it ubuntu sh
# ls
bin   dev  home  lib64  mnt  proc run  srv  tmp var
boot  etc  lib   media  opt  root sbin sys  usr
# cd bin
# ls -a
.                false      more           stty                                uname
..               fgrep      mount          su                                  uncompress
bash             findmnt    mountpoint     sync                                vdir
cat              grep       mv             systemctl                           wdctl
chgrp            gunzip     networkctl     systemd                             which
chmod            gzexe      nisdomainname  systemd-ask-password                ypdomainname
chown            gzip       pidof          systemd-escape                      zcat
cp               hostname   ps             systemd-inhibit                     zcmp
dash             journalctl pwd            systemd-machine-id-setup            zdiff
date             kill       rbash          systemd-notify                      zegrep
dd               ln         readlink       systemd-tmpfiles                    zfgrep
df               login      rm             systemd-tty-ask-password-agent      zforce
dir              loginctl   rmdir          tailf                               zgrep
dmesg            ls         run-parts      tar                                 zless
dnsdomainname    lsblk      sed            tempfile                            zmore
domainname       mkdir      sh             touch                               znew
echo             mknod      sh.destrib     true
egrep            mktemp     sleep          umount
```

Рис. 7.12 ❖ Пример выполнения нескольких команд оболочки

Откройте еще одно окно терминала и найдите идентификатор запущенного контейнера Ubuntu, выполнив следующую команду (см. рис. 7.13):

```
docker ps -a
```

```
PKOCHER-M-343X:~ parminderkochers docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
        PORTS               NAMES
eded3539719c        ubuntu              "sh"                6 minutes ago       Up 6 minutes
                                        flamboyant_edison
6a3f4a2d3694        ubuntu              "sh"                7 minutes ago       Exited (0) 7 mirutes ago
                                        friendly_wilson
PKOCHER-M-343X:~ parminderkochers
```

Рис. 7.13 ❖ Поиск идентификатора контейнера Ubuntu

Скопируйте идентификатор в буфер обмена. Теперь передайте его команде `logs`, чтобы увидеть содержимое журнала конкретного контейнера (см. рис. 7.14):

```
docker logs eded3539719c
```

```
PKOCHER-M-343X:~ parminderkochers docker logs eded3539719c
= ls
bin   dev  home  lib64  mnt  proc  run  srv  tmp  var
boot  etc  lib   media  opt  root  sbin sys  usr
= cd bin
= ls -a
.              false        more          stty                          uname
..             fgrep        mount         su                            uncompress
bash           findmnt      mountpoint    sync                          vdir
cat            grep         mv            systemctl                     wdctl
chgrp          gunzip       networkctl    systemd                       which
chmod          gzexe        nisdomainname systemd-ask-password          ypdomainname
chown          gzip         pidof         systemd-escape                zcat
cp             hostname     ps            systemd-inhibit               zcmp
dash           journalctl   pwd           systemd-machine-id-setup      zdiff
date           kill         rbash         systemd-notify                zegrep
dd             ln           readlink      systemd-tmpfiles              zfgrep
df             login        rm            systemd-tty-ask-password-agent zforce
dir            loginctl     rmdir         tailf                         zgrep
dmesg          ls           run-parts     tar                           zless
dnsdomainname  lsblk        sed           tempfile                      zmore
domainname     mkdir        sh            touch                         znew
echo           mknod        sh.distrib    true
egrep          mktemp       steep         umount
=
=
=
=
=
=
=
=
PKOCHER-H-343X:~ parminderkochers
```

Рис. 7.14 ❖ Выполнение команды `docker logs` для просмотра журнала этого контейнера

В данном случае журнал отражает историю выполнявшихся в контейнере команд.

Рассмотрим еще один, более сложный пример. Загрузим и создадим контейнер MySQL.

Сначала загрузим последний образ MySQL (см. рис. 7.15):

```
docker pull MySQL:latest
```

```
Parminders-MacBook-Pro:~ parminderkochers$ docker pull mysql
Using default tag: latest
latest: Pulling from library/mysql

ba2949489d0b6: Pull complete
19de96c112fc  : Pull complete
2e32b26a94ed: Pull complete
637386aea7a0: Pull complete
f40aa7fe5d68 : Pull complete
ca21348f3728 : Pull complete
b783bc3b44b9: Pull complete
f94304dc94e3 : Pull complete
efb904a945ff : Pull complete
64ef882b700f : Pull complete
291b704c92b1 : Pull complete
adfeb78ac4de : Pull complete
f27e5410cda3 : Pull complete
ca4b92f905b9 : Pull complete
065018fec3d7 : Pull complete
6762f304c834 : Pull complete
library/mysql: latest: The image you are pulling has been verified. Important : image verification is a tech preview feature and
should not be relied on to provide security.

Digest: sha256:842eeladlb0f19561d9fee65bb7c6197b2a2b4093f069e7969acefb6355e8c1b
Status: Downloaded newer image for mysql:latest
Parminders-MacBook-Pro:~ parminderkochers$
```

Рис. 7.15 ❖ Загрузка последнего образа MySQL

Затем выполним команду `run`, чтобы собрать контейнер MySQL (см. рис. 7.16) и получить его идентификатор:

```
docker run --name myDatabase \
> -e MySQL_ROOT_PASSWORD=myPassword \
> -d MySQL:latest
```

Здесь `name` – это имя базы данных, `-e` – ключ для определения переменной окружения с паролем доступа к базе данных, `-d` – ключ, требующий, чтобы команда `docker run` запустила процесс в фоновом режиме.

```
Parminders-MacBook-Pro:~ parminderkochers$ docker run --name myDatabase -e MYSQL_ROOT_PASSWORD=myPassword -d mysql:latest
fcb85434597bc8abf5e97acdf985a3315027aa9836eeef4af9b66669493d2c39
Parminders-MacBook-Pro:~ parminderkochers$
```

Рис. 7.16 ❖ Запуск контейнера MySQL

Теперь проверим, запущен ли процесс в контейнере:

```
docker ps
```

Как видите, процесс запущен и выполняется (см. рис. 7.17).

```
Parminders-MacBook-Pro:~ parminderkochers docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
fcb85434597b        mysql:latest        "/entrypoint.sh mysql"   25 seconds ago      Up 24 seconds       3306/tcp            myDatabase
Parminders-MacBook-Pro:~ parminderkochers
```

Рис. 7.17 ❖ Проверка процесса в контейнере

Теперь, когда контейнер запущен и работает, подключимся к нему. Для начала узнаем порт. Конечно, все мы знаем номер порта, используемый MySQL по умолчанию, но давайте все же проверим журнал командой `logs`:

```
docker logs fcb85434597b
```

Здесь `fcb85434597b` – это идентификатор прежде запущенного контейнера (см. рис. 7.18).

```
Parminders-MacBook-Pro:~ parminderkochers docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
fcb85434597b        mysql:latest        "/entrypoint.sh mysql"   25 seconds ago      Up 24 seconds       3306/tcp            myDatabase
Parminders-MacBook-Pro:~ parminderkochers
Parminders-MacBook-Pro:~ parminderkochers
Parminders-MacBook-Pro:~ parminderkochers docker logs fcb85434597b
Running mysql_install_db
2015-10-14 03:32:52 0 [Note] /usr/sbin/mysqld [mysqld 5.6.27] starting as process 15 ...
2015-10-14 03:32:52 15 [Note] InnoDB: Using atomics to ref count buffer pool pages
2015-10-14 03:32:52 15 [Note] InnoDB: The InnoDB memory heap is disabled
2015-10-14 03:32:52 15 [Note] InnoDB: Mutexes and rw_locks use GCC atomic builtins
2015-10-14 03:32:52 15 [Note] InnoDB: Memory barrier is not used
2015-10-14 03:32:52 15 [Note] InnoDB: Compressed tables use zlib 1.2.8
2015-10-14 03:32:52 15 [Note] InnoDB: Using Linux native AIO
2015-10-14 03:32:52 15 [Note] InnoDB: Using CPU crc32 instructions
2015-10-14 03:32:52 15 [Note] InnoDB: Initializing buffer pool, size= 128.0M
2015-10-14 03:32:52 15 [Note] InnoDB: Completed initialization of buffer pool
2015-10-14 03:32:52 15 [Note] InnoDB: The first specified data file ./ibdata1 did not exist: a new database to be created!
2015-10-14 03:32:52 15 [Note] InnoDB: Setting file ./ibdata1 size to 12 MB
2015-10-14 03:32:52 15 [Note] InnoDB: Database physically writes the file full: wait ...
2015-10-14 03:32:52 15 [Note] InnoDB: Setting log file ./ib_logfile101 size to 48 MB
2015-10-14 03:32:52 15 [Note] InnoDB: Setting log file ./ib_logfile1 size to 48 MB
2015-10-14 03:32:52 15 [Note] InnoDB: Renaming log file ./ib_logfile101 to ./ib_logfile0
2015-10-14 03:32:52 15 [Warning] InnoDB: New log files created, LSN=45781
2015-10-14 03:32:52 15 [Note] InnoDB: Doublewrite buffer not found: creating new
2015-10-14 03:32:52 15 [Note] InnoDB: Doublewrite buffer created
2015-10-14 03:32:52 15 [Note] InnoDB: 128 rollback segment(s) are active.
2015-10-14 03:32:52 15 [Warning] InnoDB: Creating foreign key constraint system tables.
2015-10-14 03:32:52 15 [Note] InnoDB: Foreign key constraint system tables created
2015-10-14 03:32:52 15 [Note] InnoDB: Creating tablespace and datafile system tables.
2015-10-14 03:32:52 15 [Note] InnoDB: Tablespace and datafile system tables created.
2015-10-14 03:32:52 15 [Note] InnoDB: Waiting for purge to start
2015-10-14 03:32:52 15 [Note] InnoDB: 5.6.27 started; log sequence number 0
2015-10-14 03:32:53 15 [Note] Binlog end
2015-10-14 03:32:53 15 [Note] InnoDB: FTS optimize thread exiting.
2015-10-14 03:32:53 15 [Note] InnoDB: Starting shutdown ...
2015-10-14 03:32:54 15 [Note] InnoDB: Shutdown completed; log sequence number 1625977
```

Рис. 7.18 ❖ Результат выполнения команды `logs`

На рис. 7.19 можно увидеть версию и номер порта, на котором MySQL принимает запросы на соединение.

```
MySQL init process done. Ready for start up.

2015-10-14 03:33:00 0 [Note] mysqld (mysqld 5.6.27) starting as process 1 ...
2015-10-14 03:33:00 1 [Note] Plugin 'FEDERATED' is disabled.
2015-10-14 03:33:00 1 [Note] InnoDB: Using atomics to ref count buffer pool pages
2015-10-14 03:33:00 1 [Note] InnoDB: The InnoDB memory heap is disabled
2015-10-14 03:33:00 1 [Note] InnoDB: Mutexes and rw_locks use GCC atomic builtins
2015-10-14 03:33:00 1 [Note] InnoDB: Memory barrier is not used
2015-10-14 03:33:00 1 [Note] InnoDB: Compressed tables use zlib 1.2.8
2015-10-14 03:33:00 1 [Note] InnoDB: Using Linux native AIO
2015-10-14 03:33:00 1 [Note] InnoDB: Using CPU crc32 instructions
2015-10-14 03:33:00 1 [Note] InnoDB: Initializing buffer pool, size= 128.0M
2015-10-14 03:33:00 1 [Note] InnoDB: Completed initialization of buffer pool
2015-10-14 03:33:00 1 [Note] InnoDB: Highest supported file format is Barracuda.
2015-10-14 03:33:00 1 [Note] InnoDB: 128 rollback segment(s) are active.
2015-10-14 03:33:00 1 [Note] InnoDB: Waiting for purge to start
2015-10-14 03:33:00 1 [Note] InnoDB: 5.6.27 started; log sequence number 1625997
2015-10-14 03:33:00 1 [Note] Server hostname (bind-address): '*'; port: 3306
2015-10-14 03:33:00 1 [Note] IPv6 is available.
2015-10-14 03:33:00 1 [Note]   - '::' resolves to '::';
2015-10-14 03:33:00 1 [Note] Server socket created on IP: '::'.
2015-10-14 03:33:00 1 [Warning] 'proxies_priv' entry '@ root@fcb85434597b' ignored in --skip-name-resolve mode.
2015-10-14 03:33:00 1 [Note] Event Scheduler: Loaded 0 events
2015-10-14 03:33:00 1 [Note] mysqld: ready for connections.
Version: '5.6.27' socket: '/var/run/mysqld/mysqld.sock' port: 3306 MySQL Community Server (GPL)
Parminders-MacBook-Pro:~ parminderkochers$
```

Рис. 7.19 ❖ Версия и номер порта, на котором MySQL принимает запросы

Напомню еще раз, что `docker logs` выводит содержимое `stdout` и `stderr` для указанного контейнера. Не путайте это со стандартными файлами журналов MySQL.

> **Примечание.** Узнать номер порта, прослушиваемого контейнером, можно также с помощью команды docker ps. Если вы заметили, на рис. 7.18 есть столбец `PORTS` со значением `3306/tcp`, которое показывает, что сервер MySQL готов принимать запросы через порт `3306`.

docker restart

Команда `docker restart` перезапускает указанный контейнер:

```
docker restart [options] container
```

Перезапустим контейнер Ubuntu, указав его идентификатор `c8b9770c88e9`, который мы видели в примерах выше (см. рис. 7.20).

```
Parminders-MacBook-Pro:~ parminderkochers$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                      PORTS
 NAMES
c8b9770c88e9        ubuntu              "sh"                5 minutes ago       Exited (0) 3 minutes ago
 admiring_albattani
Parminders-MacBook-Pro:~ parminderkochers$ docker restart c8b9770c88e9
C8b9770c88e9
```

Рис. 7.20 ❖ Повторный запуск контейнера Ubuntu

Если снова выполнить команду `docker ps`, мы увидим, что контейнер активен, как показано на рис. 7.21.

```
Parminders-MacBook-Pro:~ parminderkochers$ docker restart c8b9770c88e9
C8b9770c88e9
Parminders-MacBook-Pro:~ parminderkochers$ docker ps
CONTAINER ID      IMAGE       COMMAND      CREATED        STATUS          PORTS       NAMES
c8b9770c88e9      ubuntu      "sh"         2 weeks ago    Up 6 seconds                  admiring_albattani
Parminders-MacBook-Pro:~ parminderkochers
```

Рис. 7.21 ❖ Команда `docker ps` вывела список активных контейнеров

Обратите внимание, что на этот раз мы не получили приглашения к вводу. Это можно исправить, выполнив команду `docker attach`, которая будет обсуждаться далее.

docker attach

Команда `docker attach` позволяет передать указанный активный контейнер под интерактивное управление или увидеть его стандартный вывод:

```
docker attach [options] container
```

Подключим контейнер Ubuntu с идентификатором `c8b9770c88e9`, чтобы получить возможность взаимодействовать с командной оболочкой в нем (см. рис. 7.22).

```
Parminders-MacBook-Pro:~ parminderkochers$ docker ps -a
CONTAINER ID      IMAGE       COMMAND      CREATED        STATUS                     PORTS       NAMES
c8b9770c88e9      ubuntu      "sh"         8 minutes ago  Exited (0) 24 seconds ago                  admiring_albattani
Parminders-MacBook-Pro:~ parminderkochers$ docker restart c8b9770c88e9
c8b9770c88e9
Parminders-MacBook-Pro:~ parminderkochers$ docker ps
CONTAINER ID      IMAGE       COMMAND      CREATED        STATUS          PORTS       NAMES
c8b9770c88e9      ubuntu      "sh"         8 minutes ago  Up 5 seconds                  admiring_albattani
Parminders-MacBook-Pro:~ parminderkochers$ docker attach c8b9770c88e

=
```

Рис. 7.22 ❖ Получение интерактивного доступа к контейнеру с помощью команды `docker attach`

Как видите, приглашение к вводу появилось вновь, и мы можем продолжить эксперименты. Важно отметить, что мы всегда будем получать приглашение к вводу после перезапуска контейнера, несмотря ни на что. Это поведение по умолчанию нельзя изменить, потому что мы определили его, собрав контейнер с ключом `-it` в команде `run`. Конечно, мы можем запустить тот же образ Ubuntu с другими ключами и параметрами. В этом заключается особая прелесть Docker.

docker rm

Команда `docker rm` удаляет указанные контейнеры:

```
docker rm [options] containers
```

Попробуем для примера удалить контейнер Ubuntu. Перед удалением контейнер нужно остановить или передать команде `rm` ключ `-f` (force – принудительно), действие которого заключается в отправке сигнала `SIGKILL` процессу, выполняющемуся в контейнере:

```
docker stop [options] containers
```

На рис. 7.23 показано состояние контейнера Ubuntu. Как видите, этот контейнер находится в активном состоянии и выполняется уже 38 ч (см. столбец `STATUS`).

```
Parminders-MacBook-Pro:~ parminderkochers$ docker ps -a
CONTAINER ID    IMAGE     COMMAND    CREATED       STATUS          PORTS    NAMES
c8b9770c88e9    ubuntu    "sh"       2 weeks ago   Up 38 hours              admiring_albattani
Parminders-MacBook-Pro:~ parminderkochers$
```

Рис. 7.23 ❖ Состояние контейнера Ubuntu

Теперь выполним команду `stop` и снова проверим состояние (см. рис. 7.24).

```
Parminders-MacBook-Pro:~ parminderkochers$ docker stop c8b9770c88e9

c8b9770c88e9
Parminders-MacBook-Pro:~ parminderkochers
Parminders-MacBook-Pro:~ parminderkochers$ docker ps -a
CONTAINER ID    IMAGE     COMMAND    CREATED       STATUS                    PORTS    NAMES
c8b9770c88e9    ubuntu    "sh"       2 weeks ago   Exited (137) 18 seconds ago        admiring_albattani
```

Рис. 7.24 ❖ Результат выполнения команд `stop` и `ps -a`

Как видите, контейнер больше не выполняется. Он получил статус «завершился» с кодом 137, который свидетельствует о том, что контейнер получил сигнал `SIGKILL`. Команда `stop` сначала посылает сигнал `SIGTERM`, а затем, выждав некоторое время, сигнал `SIGKILL`. Период ожидания можно изменить, передав ключ `-t` с числом секунд. Эта возможность может очень пригодиться в случаях, когда требуется, чтобы процесс в контейнере успел завершить обработку запроса (например, запроса HTTP).

Также можно воспользоваться командой `docker kill`, которая немедленно посылает сигнал `SIGKILL`, но в этом случае процесс в контейнере не получит возможности завершиться обычным путем. Однако эта команда имеет ключ, позволяющий послать процессу в контейнере другой сигнал, отличный от `SIGKILL`.

Остановив контейнер, удалим его и вновь выполним команду `ps -a`, как показано на рис. 7.25.

```
Parminders-MacBook-Pro:~ parminderkochers$ docker rm c8b9770c88e9
c8b9770c88e9
Parminders-MacBook-Pro:~ parminderkochers
Parminders-MacBook-Pro:~ parminderkochers$ docker ps -a
CONTAINER ID   IMAGE        COMMAND      CREATED      STATUS       PORTS       NAMES
Parminders-Mac Book-Pro:~ parminderkochers$
```

Рис. 7.25 ❖ Контейнер удален

Обратите внимание, что контейнер удален бесследно, а команда `ps -a` не обнаружила никаких напоминаний о нем.

docker inspect

Команда `docker inspect` выводит подробную информацию о контейнере или образе:

```
docker inspect [options] container/image
```

Попробуем с ее помощью получить информацию о контейнере MySQL (см. рис. 7.26). Напомню, что `fcb85434597b` – это идентификатор нашего контейнера:

```
docker inspect fcb85434597b
```


```
Parminders-MacBook-Pro:~ parminderkochers docker inspect fcb85434597b
[
{
    "Id": "fcb85434597bc8abf5e97acdf985a3315027aa9836eeef4af9b66669493d2c39",
    "Created": "2015-10-14T03:32:52.567316916Z",
    "Path": "/entrypoint.sh",
    "Args": [
        "mysqld"
    ],
    "State": {
        "Running": true,
        "Paused": false,
        "Restarting": false,
        "OOMKilled": false,
        "Dead": false,
        "Pid": 501,
        "ExitCode": 0,
        "Error": "",
        "StartedAt": "2015-10-14T03:32:52.640686653Z",
        "FinishedAt": "0001-01-01T00:00:00Z"
    },
    "Image": "9726f738a97ab24fab227043c6d0f64a409b952fa41ba4dd7d28fc3d0149f718",
    "NetworkSettings": {
        "Bridge": "",
        "EndpointID": "8bell7ced4e0716b522fd0b16d2d52ef8cd587ced4d277ab98430299ff8e7eb7",
        "Gateway": "172.17.42.1",
        "GlobalIPv6Address": "",
        "GlobalIPv6PrefixLen": 0,
        "HairpinMode": false,
        "IPAddress": "172.17.0.6",
        "IPPrefixLen": 16,
        "IPv6Gateway": "",
        "LinkLocalIPv6Address": "",
        "LinkLocalIPv6PrefixLen": 0,
        "MacAddress": "02:42:ac:11:00:06",
        "NetworkID": "476d36a59ec3c998ff0e8b8c87e1b64ad09571919069 5a8cdd55b99263c556ff",
        "PortMapping": null,
        "Ports": {
            "3306/tcp": null
        },
        "SandboxKey": "/var/run/docker/netns/fcb85434597b",
        "SecondaryIPAddresses": null,
        "SecondaryIPv6Addresses": null
    },
    "ResolvConfPath": "/mnt/sda1/var/lib/docker/containers/fcb85434597bc8abf5e97acdf985a3315027aa9836eeef4af9b66669493d2c39/resolv.conf",
    "HostnamePath": "/mnt/sda1/var/lib/docker/containers/fcb85434597bc8abf5e97acdf985a3315027aa9836eeef4af9b66669493d2c39/hostname",
    "HostsPath": "/mnt/sda1/var/lib/docker/containers/fcb85434597bc8abf5e97acdf985a3315027aa9836eeef4af9b66669493d2c39/hosts",
    "LogPath": "/mnt/sda1/var/lib/docker/containers/fcb85434597bc8abf5e97acdf985a3315027aa9836eeef4af9b66669493d2c39/fcb85434597bc8abf5e97acdf985a33
            15027aa9836eeef4af9b66669493d2c39-json.log",
    "Name": "/myDatabase",
    "RestartCount": 0,
    "Driver": "aufs",
```


Рис. 7.26 ❖ Результаты выполнения команды `docker inspect`

Как видите, команда вернула массив JSON со всей информацией о контейнере. Есть возможность задать другой формат вывода или запросить другую информацию, например имя базы данных, IP-адрес и порт.

Следующая команда вернет имя базы данных:

```
docker inspect -format='{{.Name}}' fcb85434597b
```

А эта команда вернет IP-адрес контейнера MySQL:

```
docker inspect \
> -format='{{.NetworkSettings.IPAddress}}' fcb85434597b
```

docker exec

Команда `docker exec` позволяет удаленно выполнить другую команду в уже запущенном контейнере:

```
docker exec [options] container command [arg...]
```

Давайте попробуем выполнить эту команду в контейнере Ubuntu (см. рис. 7.27). Напомню, что `c8b9770c88e9` – это идентификатор контейнера:

```
docker exec c8b9770c88e9 ls -a
```

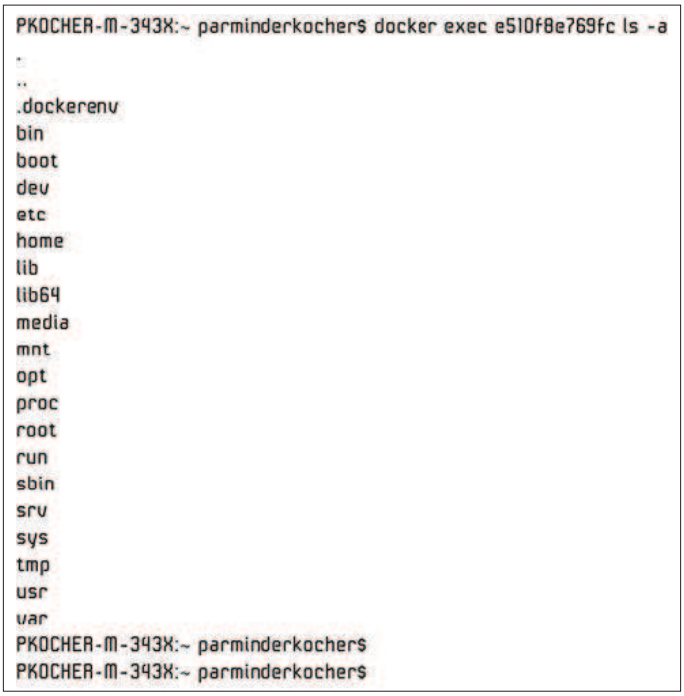


Рис. 7.27 ❖ Команда `docker exec` позволяет выполнить другую команду в запущенном контейнере

docker rename

Вы еще не устали, копируя и вставляя идентификаторы контейнеров? Мы можем дать контейнерам осмысленные имена, которые проще запомнить и классифицировать. Команда `docker rename` позволяет переименовать уже запущенный контейнер:

```
docker rename container new_name
```

Попробуем переименовать контейнер Ubuntu. Сначала определим его текущее имя.

```
docker ps -a
```

Обратите внимание на рис. 7.28. В данный момент этот контейнер имеет имя `jolly_gates`.

```
PKOCHER-M-343X:~ parminderkochers docker ps -a
CONTAINER ID    IMAGE     COMMAND    CREATED          STATUS                      PORTS    NAMES
e510f8e769fc    ubuntu    "sh"       8 minutes ago    Up 8 minutes                         jolly_gates
eded3539719c    ubuntu    "sh"       26 minutes ago   Exited (0) 8 minutes ago             flamboyant_edison
6a3f4a2d3694    ubuntu    "sh"       28 minutes ago   Exited (0) 27 minutes ago            friendly_wilson
PKOCHER-M-343X:~ parminderkochers docker
```

Рис. 7.28 ❖ Определение имени контейнера с помощью команды `docker ps -a`

А затем выполним команду `rename`:

```
docker rename e510f8e769fc Parminder
```

Обратите внимание на рис. 7.29. Команда `rename` изменила имя контейнера.

```
PKOCHER-M-343X:~ parminderkochers docker ps -a
CONTAINER ID    IMAGE     COMMAND    CREATED          STATUS                      PORTS    NAMES
e510f8e769fc    ubuntu    "sh"       8 minutes ago    Up 8 minutes                         jolly_gates
eded3539719c    ubuntu    "sh"       26 minutes ago   Exited (0) 8 minutes ago             flamboyant_edison
6a3f4a2d3694    ubuntu    "sh"       28 minutes ago   Exited (0) 27 minutes ago            friendty_wilson
PKOCHER-M-343X:~ parminderkochers docker rename e510f8e769fc Parminder
PKOCHER-M-343X:~ parminderkochers
PKOCHER-M-343X:~ parminderkochers
PKOCHER-M-343X:~ parminderkochers
PKOCHER-M-343X:~ parminderkochers docker ps -a
CONTAINER ID    IMAGE     COMMAND    CREATED          STATUS                      PORTS    NAMES
e510f8e769fc    ubuntu    "sh"       10 minutes ago   Up 10 minutes                        Parminder
eded3539719c    ubuntu    "sh"       28 minutes ago   Exited (0) 10 minutes ago            flamboyant_edison
6a3f4a2d3694    ubuntu    "sh"       29 minutes ago   Exited (0) 28 minutes ago            friendty_wilson
PKOCHER-M-343X:~ parminderkochers
```

Рис. 7.29 ❖ Контейнер успешно переименован

Теперь новое имя можно использовать в командах `docker` вместо шестнадцатеричного идентификатора (см. рис. 7.30).

```
PKOCHER-M-343X:~ parminderkochers docker logs Parminder
= ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
PKOCHER-M-343X:~ parminderkochers
```

Рис. 7.30 ❖ Новое имя можно использовать в командах `docker`

docker cp

Команда `docker cp` позволяет копировать файлы между контейнером и машиной, на которой тот выполняется. Так выглядит команда копирования файла из контейнера на локальную машину:

```
docker cp [options] container:src_path dest_path
```

А это команда копирования файла с локальной машины в контейнер:

```
docker cp [options] src_path|- container:dest_path
```

Сначала попробуем скопировать файл из контейнера Ubuntu на локальную машину. На рис. 7.31 показан файл `sample.txt`, который будет использоваться в этом примере.

```
# pwd
/var
# ls -a
. . .  backups  cache  lib  local  lock  log  mail  opt  run  sample.txt  spool  tmp
#
```

Рис. 7.31 ❖ Файл `sample.txt`, используемый в примере

В предыдущем примере мы уже присвоили своему контейнеру имя `Parminder`, поэтому используем его в команде копирования файла (см. рис. 7.32):

```
docker cp Parminder:/var/sample.txt .
```

```
PKOCHER-M-343X:~ parminderkochers$ docker cp Parminder:/var/sample.txt .
PKOCHER-M-343X:~ parminderkochers$ ls
Applications                                    Downloads                       Public
Box Sync                                        IdeaProjects                    Root
Cloudera-Admin-test-VM                          Learning Scala                  VirtualBox VMs
Cloudera-Admin-test-VM.zip                      Library                         Whiteboard.ucf
Cloudera-Training-Get2EC2-VM-1.1-vmware-1.1     Movies                          eclipse
Cloudera-Training-Get2EC2-VM-1.1-vmware-1.1.zip Music                           myGitProject
Desktop                                         MyDocker                        sample.txt
Dockerfile                                      MyJabberFiles                   target
Documents                                       Pictures
PKOCHER-M-343X:~ parminderkochers$
```

Рис. 7.32 ❖ Копирование файла из контейнера Parminder на локальную машину

Теперь попытаемся скопировать файл с локальной машины в контейнер. Используем в этом примере файл `Myfile.txt`, находящийся на локальной машине (см. рис. 7.33).

```
PKOCHER-M-343X:~ parminderkochers$ touch MyFile.txt
PKOCHER-M-343X:~ parminderkochers$ ls
Applications                                    Downloads                       Pictures
Box Sync                                        IdeaProjects                    Public
Cloudera-Admin-test-VM                          Learning Scala                  Root
Cloudera-Admin-test-VM.zip                      Library                         VirtualBox VMs
Cloudera-Training-Get2EC2-VM-1.1-vmware-1.1     Movies                          Whiteboard.ucf
Cloudera-Training-Get2EC2-VM-1.1-vmware-1.1.zip Music                           eclipse
Desktop                                         MyDocker                        myGitProject
Dockerfile                                      MyFile.txt                      sample.txt
Documents                                       MyJabberFiles                   target
PKOCHER-M-343X:~ parminderkochers$
```

Рис. 7.33 ❖ Файл `Myfile.txt` на локальной машине, который будет использован в примере

Следующая команда скопирует этот файл в каталог /var в контейнере с именем Parminder (см. рис. 7.34). Напомню, что Parminder – это имя контейнера Ubuntu:

```
docker cp MyFile.txt Parminder:/var
```

```
# pwd
/var
# ls -a
.  ..  MyFile.txt  backups  cache  lib  local  lock  log  mail  opt  run  sample.txt  spool  tmp
#
```

Рис. 7.34 ❖ Файл Myfile.txt скопирован с локальной машины в каталог /var внутри контейнера с именем Parminder

docker pause/unpause

Команда docker pause приостанавливает все процессы в указанных контейнерах:

```
docker pause container [container...]
```

В Linux она использует механизм управления группами процессов.

Команда docker unpause возобновляет выполнение контейнеров:

```
docker unpause container [container...]
```

Попробуем приостановить контейнер Ubuntu, как показано на рис. 7.35. Напомню, что Parminder – это имя данного контейнера:

```
docker pause Parminder
```

```
PKOCHER-M-343X:~ parminderkochers docker pause Parminder
Parminder
PKOCHER-M-343X:~ parminderkochers docker ps -a
CONTAINER ID   IMAGE   COMMAND   CREATED              STATUS                        PORTS   NAMES
e02085c7ba70   ubuntu  "sh"      19 minutes ago       Exited (0) 18 minutes ago             vibrant_saha
e510f8e769fc   ubuntu  "sh"      23 hours ago         Up 16 minutes (Paused)                Parminder
eded3539719c   ubuntu  "sh"      23 hours ago         Exited (0) 23 hours ago               flamboyant_edison
6a3f4a2d3694   ubuntu  "sh"      23 hours ago         Exited (0) 23 hours ago               friendly_wilson
PKOCHER-M-343X:~ parminderkochers
```

Рис. 7.35 ❖ Результат выполнения команды docker pause для контейнера Ubuntu

Мы только что приостановили контейнер, фактически сделав то же самое со всеми процессами внутри него. Если теперь попробовать выполнить какую-нибудь команду внутри контейнера, то можно получить результат, изображенный на рис. 7.36.

```
# ls
```

Рис. 7.36 ❖ Попытка выполнить команду внутри приостановленного контейнера

Теперь возобновим выполнение контейнера (см. рис. 7.37):

```
docker unpause Parminder
```

```
PKOCHER-M-343X:~ parminderkochers docker unpause Parminder
Parminder
PKOCHER-M-343X:~ parminderkochers docker ps -a
CONTAINER ID    IMAGE     COMMAND    CREATED              STATUS                      PORTS    NAMES
e02085c7ba70    ubuntu    "sh"       22 minutes ago       Exited (0) 21 minutes ago           vibrant_saha
e510f8e769fc    ubuntu    "sh"       23 hours ago         Up 19 minutes                       Parminder
eded3539719c    ubuntu    "sh"       23 hours ago         Exited (0) 23 hours ago             flamboyant_edison
6a3f4a2d3694    ubuntu    "sh"       23 hours ago         Exited (0) 23 hours ago             friendly_wilson
PKOCHER-M-343X:~ parminderkochers
```

Рис. 7.37 ❖ Выполнение контейнера возобновлено

Так как мы возобновили выполнение контейнера, все процессы в нем тут же продолжили работу. Наша «подвисшая» команда `ls` тоже благополучно выполнилась, как показано на рис. 7.38.

```
# ls
MyFile.txt  backups  cache  lib  local  lock  log  mail  opt  run  sample.txt  spool  tmp
#
```

Рис. 7.38 ❖ Подвисшая команда `ls`, запущенная в предыдущем примере, благополучно выполнилась

docker create

Команда `docker create` создает новый уровень контейнера, доступный для записи, поверх указанного образа и подготавливает его для выполнения указанной команды:

```
docker create [options] image [command] [arg...]
```

После выполнения она выводит идентификатор контейнера. Эта команда отличается от `docker run -d` тем, что не запускает контейнер. Чтобы запустить его, нужно выполнить команду `docker start`. Она может пригодиться в случаях, когда нужно заранее создать и настроить контейнер, чтобы он был готов к запуску, когда наступит подходящий момент.

Создадим новый контейнер (см. рис. 7.39):

```
docker create -t -i fedora bash
```

```
PKOCHER-M-343X:~ parminderkochers docker create -t -i fedora bash
Unable to find image 'fedora:latest' locally
latest: Pulling from library/fedora
1b3997eabd9: Pull complete
Digest: sha256:8d3f642aa4d3fa8f9dc52ab0e3bbbe8bc2494643dc6ebb26c4a6958db888e5a2
Status: Downloaded newer image for fedora:latest
239cae10b3cf6d35d3f8621f8eab5a7bcdf9bcd363e4e21971de7e7b2365654f
PKOCHER-M-343X:~ parminderkochers docker ps -a
CONTAINER ID    IMAGE     COMMAND    CREATED              STATUS                      PORTS    NAMES
239cae10b3cf    fedora    "bash"     18 seconds ago       Created                             sleepy_euclid
e02085c7ba70    ubuntu    "sh"       31 minutes ago       Exited (0) 30 minutes ago           vibrant_saha
e510f8e769fc    ubuntu    "sh"       23 hours ago         Exited (0) 42 seconds ago           Parminder
eded3539719c    ubuntu    "sh"       23 hours ago         Exited (0) 23 hours ago             flamboyant_edison
6a3f4a2d3694    ubuntu    "sh"       23 hours ago         Exited (0) 23 hours ago             friendly_wilson
PKOCHER-M-343X:~ parminderkochers
```

Рис. 7.39 ❖ Новый контейнер создан

Обратите внимание, что контейнер создан, но не запущен.

docker commit

Команда `docker commit` является простой, но очень важной. Она позволяет создать новый образ со всеми изменениями, выполненными в контейнере:

```
docker commit [options] container [repository:tag]
```

Если, выполнив какие-то изменения в контейнере, вы решите передать его кому-то, например группе разработки или тестирования, в виде образа, то сможете создать такой образ из запущенного контейнера.

docker diff

Команда `docker diff` говорит сама за себя. Это еще одна очень важная команда, выводящая изменения в файлах и каталогах в файловой системе контейнера:

```
docker diff container
```

По прошествии времени, когда вы внесете какие-либо изменения в свой контейнер, эта команда поможет вам увидеть отличия в файловой системе в сравнении с базовым образом.

Dockerfile

Давайте соберем тот же контейнер MySQL, что использовался в предыдущих примерах, поверх контейнера Ubuntu, применив Dockerfile. Как уже говорилось, файл Dockerfile содержит последовательность инструкций, которые Docker должен выполнить, чтобы собрать образ. Это простой текстовый файл, который можно создать, даже не зная никакого языка программирования. В нем используются команды с простым синтаксисом.

Далее дается краткое описание формата файла:

- файл Dockerfile всегда должен начинаться с инструкции `FROM`, определяющей базовый образ. Строки комментариев должны начинаться с символа #. Инструкция `FROM` поддерживает переменные, объявляемые в инструкции `ARG` (это единственная инструкция, которая может предшествовать инструкции `FROM`). Например:

  ```
  ARG OS_VERSION=14.04
  FROM Ubuntu:${OS_VERSION}
  ```

- инструкции имеют синтаксис: `instruction arguments`;
- инструкции выполняются последовательно, сверху вниз;
- вместе с файлом Dockerfile демону Docker передаются все файлы, находящиеся в этом же каталоге. Поэтому, чтобы не загромождать образ, не храните лишних файлов в одном каталоге с файлом Dockerfile.

Вот несколько простых инструкций, которые можно использовать в Dockerfile:

- `ADD` копирует файлы из указанного каталога в локальной системе или URL удаленной системы в указанный каталог внутри контейнера;

- CMD выполняет указанную команду после создания контейнера. В файле Dockerfile может быть только одна инструкция CMD. Если указать несколько инструкций CMD, выполнена будет самая последняя;
- ENTRYPOINT определяет выполняемый файл в контейнере, который должен запускаться по умолчанию в момент запуска контейнера. Эта инструкция обязательно должна быть, если вы хотите, чтобы процесс в контейнере запускался автоматически, иначе вам придется использовать инструкцию CMD;
- ENV определяет переменные окружения, которые затем могут использоваться в других инструкциях, например ENV MySQL_ROOT_PASSWORD mypassword;
- EXPOSE определяет номер порта, который будет прослушивать контейнер;
- FROM определяет образ, который будет служить основой для создаваемого образа. Эта инструкция должна быть самой первой в Dockerfile. Она является обязательной;
- MAINTAINER определяет автора информации в сгенерированных образах, например MAINTAINER pkocher@domain.com;
- RUN выполняет указанную команду (или команды). Для каждой инструкции RUN будет создан свой уровень. Каждый следующий уровень накладывается поверх предыдущего подтвержденного;
- USER определяет имя или идентификатор пользователя для использования при запуске образа или в таких инструкциях, как RUN, CMD и ENTRYPOINT;
- VOLUME определяет один или несколько общих томов в файловой системе хост-машины, которые будут доступны из контейнера;
- WORKDIR определяет рабочий каталог для инструкций RUN, CMD, ENTRYPOINT, COPY и ADD.

Dockerfile для MySQL

Теперь, разобравшись с форматом Dockerfile, создадим файл, который генерирует контейнер для MySQL поверх контейнера Ubuntu. Запустите любой текстовый редактор (vi, pico или другой, который выберете) и создайте новый файл с именем Dockerfile. Добавьте в него следующие инструкции:

```
FROM ubuntu:14.04
MAINTAINER pkocher@domain.com
RUN apt -get update
RUN apt -get -y install MySQL-server
EXPOSE 3306
CMD ["/usr/bin/MySQLd_safe"]
```

Сохраните файл и закройте редактор. Обратите внимание, что в качестве начального использован образ Ubuntu версии 14.04. Инструкция RUN выполнит команду apt -get -y install, которая загрузит и установит пакет MySQL с его зависимостями. Инструкция EXPOSE откроет в контейнере порт 3306 для приема запросов.

Наконец, инструкция CMD запустит MySQL в безопасном режиме.

Теперь можно выполнить команду:

```
docker build [options] path/URL,
```

которая соберет образ, следуя инструкциям в указанном файле Dockerfile, а также с учетом контекста. Под контекстом в данном случае понимаются файлы ресурсов, находящиеся в одном каталоге с Dockerfile. Контекст определяется в виде пути к каталогу или URL репозитория GitHub.

Всегда передавайте ключ `-t` команде `docker build`, чтобы снабдить образ меткой, которую затем можно использовать для идентификации. Простые и осмысленные метки помогут вам управлять образами.

На рис. 7.40 показано, как протекает процесс сборки образа MySQL на основе только что созданного файла Dockerfile. Имейте в виду, что файл с инструкциями должен иметь имя Dockerfile и в одном каталоге с ним не должно быть никаких других файлов:

```
docker build -t pkocher/MySQL
```

```
Parminders-MacBook-Pro:MyDocker parminderkocher$ docker build -t pkocher/mysql .
Sending build context to Docker daemon 3.072 kB
Step 0 : FROM ubuntu
 ---> 91e54dfb1179
Step 1 : MAINTAINER pkocher@gmail.com
 ---> Running in ff60156730fd
 ---> 0dd9db6f7989
Removing intermediate container ff60156730fd
Step 2 : RUN apt-get -y install mysql-server
 ---> Running in fe6f48d526af
Reading package lists ...
Building dependency tree ...
Reading state information ...
The following extra packages will be installed:
  libaio1 libdbd-mysql-perl libdbi-perl libhtml-template-perl libmysqlclient18
  libterm-readkey-perl libwrap0 mysql-client-5.5 mysql-client-core-5.5
  mysql-common mysql-server-5.5 mysql-server-core-5.5 psmisc tcpd
Suggested packages:
  libclone-perl libmldbm-perl libnet-daemon-perl libplrpc-perl
  libsql-statement-perl libipc-sharedcache-perl tinyca mailx
The following NEW packages will be installed:
  libaio1 libdbd-mysql-perl libdbi-perl libhtml-template-perl libmysqlclient18
  libterm-readkey-perl libwrap0 mysql-client-5.5 mysql-client-core-5.5
  mysql-common mysql-server mysql-server-5.5 mysql-server-core-5.5 psmisc tcpd
0 upgraded, 15 newly installed, 0 to remove and 0 not upgraded.
Need to get 9159 kB of archives.
After this operation, 97.0 MB of additional disk space will be used.
Get: 1 http://archive.ubuntu.com/ubuntu/ trusty/main libaio1 amd64 0.3.109-4 [6364 B]
Get: 2 http://archive.ubuntu.com/ubuntu/ trusty/main mysql-common all 5.5.35+dfsg-1ubuntu1 [14.1 kB]
```

Рис. 7.40 ❖ Сборка образа MySQL

Как можно видеть на рис. 7.41, Docker начинает сборку с первой инструкции и последовательно выполняет все последующие, кешируя результаты.

Если снова запустить сборку с тем же самым файлом Dockerfile, то ничего не произойдет, потому что ничего не изменилось. Попробуйте и убедитесь сами.

По окончании сборки вы получите файл образа, который можно добавить в репозиторий. Давайте проверим:

```
docker images
```

Как показано на рис. 7.42, образ pkocher/MySQL готов к использованию.

```
Setting up libhtml-template-perl (2.95-1) ...
Setting up tcpd (7.6.q-25) ...
Processing triggers for ureadahead (0.100.0-16) ...
Setting up mysql-server (5.5.35+dfsg-1ubuntu1) ...
Processing triggers for libc-bin (2.19-0ubuntu6.6) ...
 ---> 08e9a7c04c4f
Removing intermediate container fe6f48d526af
Step 3 : EXPOSE 3306
 ---> Running in 1ae5e57c81ce
 ---> 2e9d44165b70
Removing intermediate container 1ae5e57c81ce
Step 4 : CMD /usr/bin/mysqld_safe
 ---> Running in a09f3a5bc93e
 ---> ae267abf008c
Removing intermediate container a09f3a5bc93e
Successfully built ae267abf008c
Parminders-MacBook-Pro:MyDocker parminderkochers$
```

Рис. 7.41 ❖ Docker последовательно выполняет инструкции сборки и кеширует их результаты

```
Parminders-MacBook-Pro MyDocker parmirderkochers$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
pkocher/mysql       latest              8ef5ceb3439e        About a minute ago  318.1 MB
mysql               latest              9726f738a97a        2 weeks ago         324.3 MB
ubuntu              latest              91e54dfb1179        8 weeks ago         188.4 MB
Parminders-MacBook-Pro MyDocker parmirderkochers$
```

Рис. 7.42 ❖ Образ pkocher/MySQL готов к использованию

Теперь запустим этот образ и убедимся, что он работает (см. рис. 7.43):

```
docker run -d -p 3306:3306 pkocher/MySQL
```

```
Parminders-MacBook-Pro:MyDocker parminderkochers$ docker run -d -p 3306:3306 pkocher/mysql
5063c4bed669ef217b65b870c126c908e522e122e992b65447ed1ae22898b419
Parminders-MacBook-Pro:MyDocker parminderkochers$
```

Рис. 7.43 ❖ Запуск образа pkocher/MySQL

Как вы помните, в нашем файле Dockerfile только одна инструкция `CMD`, которая должна запустить сервер MySQL. Проверим это:

```
docker ps
```

Как показано на рис. 7.44, наш образ действительно запущен.

```
Parminders-MacBook-Pro:MyDocker parminderkochers$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                    NAMES
5063c4bed669        pkocher/mysql       "/usr/bin/mysqld_safe"   36 seconds ago      Up 35 seconds       0.0.0.0:3306->3306/tcp   modest_euclid
Parminders-MacBook-Pro:MyDocker parminderkochers$
```

Рис. 7.44 ❖ Образ запущен

Теперь сделаем еще шаг вперед и попробуем выполнить несколько запросов. Сначала воспользуемся командой exec, чтобы запустить командную оболочку bash в этом контейнере, как показано на рис. 7.45. Обратите внимание, что 5063c4bed669 – это идентификатор контейнера:

```
docker exec -it 5063c4bed669 bash
```

```
Parmincers-MacBook-Pro:MyDocker parminderkochers docker exec -it 5063c4bed669 bash
root@5063c4bed669:/#
```

Рис. 7.45 ❖ Запуск командной оболочки bash командой docker exec

А теперь подключимся к MySQL и выполним несколько запросов, чтобы убедиться, что все действительно работает (см. рис. 7.46):

```
mysql
show databases;
connect information_schema
show tables
```

```
root@5063c4bed669:/# mysql
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.5.35-1ubuntu1 (Ubuntu)

Copyright (c) 2000, 2013, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
+--------------------+
3 rows in set (0 .00 sec)

mysql> connect information_schema
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Connection id:      3
Current database: information_schema

mysql> show tables
    -->;
+---------------------------------------+
| Tables_in_information_schema          |
+---------------------------------------+
| CHARACTER_SETS                        |
| COLLATIONS                            |
| COLLATION_CHARACTER_SET_APPLICABILITY |
| COLUMNS                               |
| COLUMN_PRIVILEGES                     |
| ENGINES                               |
| EVENTS                                |
| FILES                                 |
| GLOBAL_STATUS                         |
| GLOBAL_VARIABLES                      |
| KEY_COLUMN_USAGE                      |
| PARAMETERS                            |
| PARTITIONS                            |
| PLUGINS                               |
```

Рис. 7.46 ❖ Проверка работоспособности сервера баз данных

Компоновщик Docker Compose

Приложения, использующие Docker, обычно состоят из нескольких контейнеров, т. е. они имеют компоненты (например, приложение, веб-сервер, базу данных), развернутые в нескольких контейнерах Docker. Чтобы упростить запуск многоконтейнерных приложений, в рамках проекта Docker был создан инструмент Docker Compose.

Допустим, что вам нужно развернуть приложение, включающее сервер Tomcat и базу данных MySQL. Вот как можно определить эти два компонента в файле *docker-compose.yml*:

```
version: ‘2'
services:
  tomcat:
    image: ‘tomcat:7'
    container_name: appserver
    ports:
      - ‘8080:80'
    depends_on:
      - db
  db:
    image: ‘mysql:5.7'
    container_name: dbserver
    ports:
      - ‘3306:3306'
    environment:
      - MYSQL_ROOT_PASSWORD=sample
      - MYSQL_DATABASE=helpdesk
      - MYSQL_USER=helpdesk
      - MYSQL_PASSWORD=helpdesk
```

Настройки для Docker Compose оформляются в виде файла в формате YAML (YAML Ain’t Markup Language – YAML, это не язык разметки), познакомиться с которым поближе можно на сайте http://www.yaml.org. Однако файлы YAML можно использовать не только в Docker Compose, но и во многих других приложениях.

В этом файле *docker-compose.yml* определены две службы: Tomcat и MySQL. Настройки служб выглядят достаточно очевидными. Единственное, что следует отметить: служба Tomcat имеет в своей конфигурации параметр `depends_on`, описывающий зависимость от `db`. Это означает, что Docker должен первой запустить службу базы данных, а только потом Tomcat. Компоновщик Docker Compose поддерживает множество других параметров, описание которых можно найти в онлайн-документации Docker.

Запустить службы после создания файла *docker-compose.yml* можно командой

```
docker-compose up -d
```

Эта команда проверит файл *docker-compose.yml*, затем отыщет объявленные там службы, создаст граф зависимостей, определяющий порядок запуска служб, и запустит их в этом порядке. Если указанный в файле образ отсутствует на локальной машине, он будет загружен из реестра Docker обычным образом. На рис. 7.47 показан вывод команды.

```
lelakshm (remove) $ docker-compose up -d
Creating network "remove_default" with the default driver
Pulling db (mysql:5.7) ...
5.7: Pulling from library/mysql
85b1f47fba49 : Pull complete
5671503d4f93 : Pull complete
3b43b3b913cb: Pull complete
4fbb803665d0: Pull complete
05808866e6f9: Pull complete
1d8c65d48cfa : Pull complete
e189e187b2b5 : Pull complete
02d3e6011ee8 : Pull complete
d43b32d5ce04: Pull complete
2a809168ab45: Pull complete
Digest: sha256:1a2f3361228e9b10b4c77a651b46082851484 5dc7ac51735b919c2c4aec864b7
Status: Downloaded newer image for mysql:5.7
Pulling tomcat (tomcat:7) ...
7: Pulling from library/tomcat
85b1f47fba49  : Already exists
ba6bd283713a : Pull complete
b7aa4dbe97e5 : Pull complete
9e61d008c81f  : Pull complete
c29ddaee3569: Pull complete
134c34ceaaa5 : Pull complete
ce255e8bcfe2 : Pull complete
9b9cfdb3562c : Pull complete
00c3060b4e32: Pull complete
fd27456f3ba2 : Pull complete
9d04c86dfa35 : Pull complete
8300cf32c1b1  : Pull complete
Digest: sha256:9ca301c5c37cdb858332d18ba98e7097d5749a5e14e077026adfa1db4c354d4e
Status: Downloaded newer image for tomcat:7
Creating dbserver ...
Creating dbserver ... done
Creating appserver ...
Creating appserver ... done
lelakshm (remove) $
lelakshm (remove) $
lelakshm (remove) $ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                    NAMES
cfe293439c4f        tomcat:7            "catalina.sh run"        About a minute ago  Up About a minute   8080/tcp, 0.0.0.0:8080->80/tcp   appserver
2da28febf786        mysql:5.7           "docker-entrypoint..."   About a minute ago  Up About a minute   0.0.0.0:3306->3306/tcp   dbserver
lelakshm (remove) $
```

Рис. 7.47 ❖ Запуск служб с помощью Docker Compose

Как видите, из-за того что образы Tomcat и MySQL отсутствуют на локальной машине, они получаются из репозитория перед запуском. Нужно учитывать еще одно важное замечание. Так как MySQL является зависимостью для Tomcat, образ MySQL был загружен и запущен перед запуском Tomcat.

На этом мы завершаем обсуждение команд Docker. Они продолжают развиваться, поэтому старайтесь следить за изменениями в онлайн-документации Docker.

Глава 8

Поддержка сети в контейнерах

В трех предыдущих главах мы познакомились с основами контейнеров и узнали, как Docker объединяет контейнеры в уровни. Но контейнеры редко бывают абсолютно автономными и должны как-то взаимодействовать друг с другом и с внешним миром. В этой главе мы узнаем о поддержке сети в мире контейнеров и обсудим ее. Но прежде освежим наши знания о некоторых ключевых понятиях в Linux, которые пригодятся нам в обсуждении вопросов поддержки сети в контейнерах.

Ключевые понятия Linux

Как мы уже знаем, контейнеры – это изолированные виртуальные окружения. В них может выполняться приложение целиком или только его часть. В любом случае, одной из важнейших потребностей является возможность подключения к контейнерам по сети.

Мы уже использовали клиентов для подключения к нашим контейнерам, но нам нужна всеобъемлющая возможность взаимодействий по сети. Нужно иметь связь между контейнерами в пределах хоста и между несколькими вычислительными центрами, т. е. нам необходимо научиться создавать свою сеть. Для реализации таких возможностей Docker использует сетевую подсистему Linux и механизмы ядра.

Мы не будем слишком углубляться в основы Linux, но некоторые идеи знать необходимо, чтобы понимать, как организована поддержка сетей в Docker:

- **сетевое пространство имен в Linux.** Обычно Linux предоставляет стандартный набор сетевых интерфейсов и записей в таблице маршрутизации. Этот набор используется всеми компонентами операционной системы для осуществления сетевых взаимодействий. Сетевое пространство имен можно представить как стек с собственными сетевыми интерфейсами и соответствующими записями в таблице маршрутизации, действующими изолированно. Docker использует эту особенность для изоляции контейнеров и обеспечения безопасности. Вы можете иметь несколько сетевых пространств имен, что позволяет запустить каждый контейнер изолированно, но делает невозможным их взаимодействие с другими контейнерами на одном хосте, пока администратор не определит необходимые настройки. Хост

имеет собственное пространство имен, содержащее интерфейсы и таблицы маршрутизации хоста;

- **мост Linux.** Эта часть реализована как модуль ядра Linux и обеспечивает работу сетевой подсистемы Linux. Ее можно рассматривать как сетевой коммутатор второго уровня, который дополнительно производит фильтрацию. Здесь принимается решение о перенаправлении пакетов на основе таблицы MAC-адресов, которая заполняется динамически в процессе анализа трафика;
- **виртуальные Linux-устройства Ethernet.** Они также известны как устройства *veth* (от англ. *virtual Ethernet*). Эти интерфейсы связывают сетевые пространства имен. Можно создать несколько записей в стеке сетевого пространства имен и настроить *veth* для подключения. Эти устройства можно рассматривать как каналы, связывающие сетевые пространства имен друг с другом и с внешним миром;
- **Linux iptables.** *iptables* – это часть ядра Linux, реализующая фильтрацию пакетов и другие функции брандмауэра для операционной системы. С помощью этого механизма можно определять правила и цепочки правил, пропускающих или блокирующих трафик. Docker использует его для разделения трафика между контейнерами, реализации отображения портов, позволяя связать порт контейнера с портом хоста, и др.

Теперь, кратко обозначив сетевые механизмы Linux, обсудим способы соединения контейнеров и начнем с самого простого – прямого соединения (linking).

Прямое соединение

До появления в Docker поддержки продвинутых сетевых возможностей (которые мы обсудим чуть позже) самым простым способом организации соединений между контейнерами было «прямое соединение». Флаг `--link`, ныне считающийся устаревшим и не рекомендуемым к использованию, позволяет контейнерам обнаруживать и создавать защищенное соединение друг с другом и передавать информацию через это соединение. Данный метод является более универсальным способом соединения контейнеров, чем метод на основе отображения портов. Он основан на использовании переменных окружения и файла */etc/hosts*, который автоматически создается механизмом Docker.

Для примера организуем связь между контейнерами с сервером приложений Tomcat и базой данных MySQL. Эти два контейнера должны иметь возможность взаимодействовать друг с другом. Получим последнюю версию образа Tomcat (см. рис. 8.1), выполнив команду

```
docker pull tomcat
```

```
[ANUJSIN-M-T2H9:pkocher anujsin$ docker pull tomcat
Using default tag: latest
latest: Pulling from library/tomcat
9f0706ba7422 : Pull complete
d3942a742d22: Pull complete
2b95a7bc6bf9 : Pull complete
7bd307c6c6e7 : Pull complete
ba7da8b01135 : Pull complete
74169d04cf0d : Pull complete
08cc0e294332: Pull complete
d2f5746bc4d3 : Pull complete
eb109ae04806 : Pull complete
99ac3ea73cee: Pull complete
24772bc65b49 : Pull complete
03774cef060c : Pull complete
8673b4967afd : Pull complete
3a49ad4798f1 : Pull complete
Digest: sha256:c55c84d34b82d794293bb7ee8c70f52f9dfcd1bd34106394b2bc99ed60216f16
Status: Downloaded newer image for tomcat:latest
ANUJSIN-M-T2H9:pkocher anujsin$ ▮
```

Рис. 8.1 ❖ Получение последней версии образа Tomcat

Далее запустим контейнер Tomcat и назовем его `tomcatContainer`:

```
docker run -d -- name tomcatContainer tomcat
```

Проверим, запустился ли контейнер, выполнив команду (см. рис. 8.2):

```
docker ps
```

```
[ANUJSIN-M-T2H9:pkocher anujsin$ docker ps
CONTAINER ID   IMAGE          COMMAND            CREATED        STATUS         PORTS        NAMES
90d4a06e190e   tomcat:latest  "catalina.sh run"  4 minutes ago  Up 4 minutes   8080/tcp     tomcatContainer
ANUJSIN-M-T2H9:pkocher anujsin$ ▮
```

Рис. 8.2 ❖ Контейнер Tomcat запущен

Теперь запустим контейнер MySQL и установим прямое соединение между ним и контейнером Tomcat, используя флаг `--link`:

```
docker run --link tomcatContainer:tomcat --name sqlcontainer \
> -e MYSQL_ROOT_PASSWORD=password -d mysql
```

Эта команда получит образ MySQL, если он отсутствует на локальной машине, как показано на рис. 8.3, и запустит его.

```
ANUJSIN-M-T2H9:prometheus anujsin$ docker run --link tomcatContainer:tomcat --name sqlcontainer -e MYSQL
_ROOT_PASSWORD=password -d mysql
Unable to find image 'mysql: latest' locally
latest: Pulling from library/mysql
9f0706ba7422 : Already exists
2290e155d2d0 : Pull complete
547981b8269f : Pull complete
2c9d42ed2f48 : Pull complete
55e3122f1297 : Pull complete
abc10bd84060 : Pull complete
c0a5ce64f2b0 : Pull complete
c4595eab8e90: Pull complete
098988cead35: Pull complete
300ca5fa5eea : Pull complete
43fdc4e3e690 : Pull complete
Digest: sha256:d178dffba8d81afedc251498e227607934636e06228ac63d58b72f9e9ec271a6
```

Рис. 8.3 ❖ Получение MySQL

Давайте проверим связь между контейнерами. Сначала запустим командную оболочку в контейнере MySQL, выполнив следующую команду:

```
docker exec -it sqlcontainer /bin/bash
```

Далее проверим содержимое файла */etc/hosts*:

```
cat /etc/hosts
```

Как можно видеть на рис. 8.4, в файле присутствует запись, соответствующая контейнеру сервера приложений Tomcat, с IP-адресом 172.17.0.2.

```
[root@f864f6e4150f:/#
[root@f864f6e4150f:/#
[root@f864f6e4150f:/# cat /etc/hosts
127.0.0.1                    localhost
::1             localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.2     tomcat 90d4a06e190e tomcatContainer
172.17.0.3     f864f6e4150f
root@f864f6e4150f:/#
```

Рис. 8.4 ❖ Запись, соответствующая контейнеру сервера приложений Tomcat

Проверим действительность IP-адреса контейнера Tomcat, для чего откроем еще одно окно терминала, как показано на рис. 8.5, и выполним команду

```
docker inspect TomcatContainer | grep IP
```

```
ANUJSIN-M-T2H9:~ anujsin$
ANUJSIN-M-T2H9:~ anujsin$ docker inspect tomcatContainer |grep IP
            "LinkLocalIPv6Address": "",
            "LinkLocalIPv6PrefixLen": 0,
            "SecondaryIPAddresses": null,
            "SecondaryIPv6Addresses": null,
            "GlobalIPv6Address":"",
            "GlobalIPv6Prefixlen": 0,
            "IPAddress": "172.17.0.2"
            "IPPrefixlen": 16,
            "IPv6Gateway": "",
                    "IPAMConfig": null,
                    "IPAddress": "172.17.0.2",
                    "IPPrefixLen": 16,
                    "IPv6Gateway": "",
                    "GlobalIPv6Address": "",
                    "GlobalIPv6PrefixLen": 0,
ANUJSIN-M-T2H9:~ anujsin$
```

Рис. 8.5 ❖ Проверка IP-адреса контейнера Tomcat

Обратите внимание, что IP-адрес 172.17.0.2 соответствует указанному в файле *hosts*, а это означает, что все готово к открытию соединения. Проверим доступность контейнера Tomcat, выполнив команду `ping` из контейнера MySQL. Вернемся в предыдущий терминал и выполним такую команду:

```
ping 172.17.0.2
```

Как показано на рис. 8.6, соединение успешно установлено.

```
[root@f864f6e4150f:/#
[root@f864f6e4150f:/# ping 172.17.0.2
PING 172.17.0.2 (172.17.0.2): 56 data bytes
64 bytes from 172.17.0.2: icmp_seq=0 ttl=64 time=0.190 ms
64 bytes from 172.17.0.2: icmp_seq=1 ttl=64 time=0.109 ms
64 bytes from 172.17.0.2: icmp_seq=2 ttl=64 time=0.114 ms
64 bytes from 172.17.0.2: icmp_seq=3 ttl=64 time=0.089 ms
64 bytes from 172.17.0.2: icmp_seq=4 ttl=64 time=0.125 ms
64 bytes from 172.17.0.2: icmp_seq=5 ttl=64 time=0.103 ms
64 bytes from 172.17.0.2: icmp_seq=6 ttl=64 time=0.105 ms
^C--- 172.17.0.2 ping statistics ---
7 packets transmitted, 7 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.089/0.119/0.190/0.031 ms
```

Рис. 8.6 ❖ Соединение успешно установлено

Варианты подключения к сети по умолчанию

Флаг `--link` был объявлен устаревшим, и его поддержка рано или поздно будет прекращена, поэтому его не следует использовать в новых проектах. Вместо флага `--link` Docker предлагает три варианта подключения для соединения контейнеров, используемых по умолчанию. Все они создаются автоматически в процессе установки: `none`, `host` и `bridge`. Выполните следующую команду, чтобы перечислить их:

```
docker network ls
```

Вы должны увидеть вывод, как показано на рис. 8.7.

```
[ANUJSIN-M-T2H9:~ anujsin$ docker network ls
NETWORK ID          NAME                DRIVER          SCOPE
fe3118460998        bridge              bridge          local
4a8e216f9a47        host                host            local
1bb6d94233c0        none                null            local
ANUJSIN-M-T2H9:~ anujsin$
```

Рис. 8.7 ❖ Список сетей

Рассмотрим поближе каждый из этих вариантов.

none

Это самый простой вариант, который, по сути, означает отсутствие поддержки сетевых взаимодействий. Он получает стек и пространство имен для контейнера без сетевого интерфейса. Для этого контейнера не настроен IP-адрес, и он не может соединяться с другими контейнерами и внешней сетью. Но поддержка петлевого интерфейса с IP-адресом 127.0.0.1 имеется.

Например, запустим контейнер Tomcat еще раз, но теперь укажем значение `none` в параметре `network`:

```
docker run -it --network=none tomcat /bin/bash
```

Проверим IP-адрес контейнера:

```
docker inspect 43c10fe289b3| grep IP
```

Как и следовало ожидать, контейнер не получил IP-адреса (см. рис. 8.8).

```
ANUJSIN-M-T2H9:~ anujsin$
ANUJSIN-M-T2H9:~ anujsin$
ANUJSIN-M-T2H9:~ anujsin$ docker inspect 43c10fe289b3 |grep IP
            "LinkLocalIPv6Address": "",
            "LinkLocalIPv6PrefixLen": 0,
            "SecondaryIPAddresses": null,
            "SecondaryIPv6Addresses": null,
            "GlobalIPv6Address":"",
            "GlobalIPv6PrefixLen": 0,
            "IPAddress": "",
            "IPPrefixLen": 0,
            "IPv6Gateway": "",
                    "IPAMConfig": null,
                    "IPAddress": "",
                    "IPPrefixLen": 0,
                    "IPv6Gateway": "",
                    "GlobalIPv6Address": "",
                    "GlobalIPv6PrefixLen": 0,
ANUJSIN-M-T2H9:~ anujsin$
```

Рис. 8.8 ❖ Контейнер не получил IP-адреса

Как видите, этот конкретный контейнер оказался полностью изолированным от других контейнеров и от сети. Данный вариант настройки сети обычно используется для тестирования в изолированном окружении, когда подключение извне не предусматривается.

host

Как следует из названия, вариант `host` добавляет контейнер в сетевое пространство имен хоста, т. е. хост и контейнер используют общее пространство имен. Это второй самый простой вариант организации сетевых взаимодействий. Контейнер может использовать все интерфейсы в стеке хоста. В этом случае сетевые порты контейнера и хоста отображаются один в один, т. е. если запустить контейнер с сервером приложений, прослушивающим порт 8080, то этот сервер приложений будет доступен через порт 8080 хоста.

Здесь следует сделать два важных замечания: в этом режиме все еще необходимо определить настройки сети, а также здесь отсутствует возможность использования отображения портов. Причина в том, что контейнер и хост используют общее пространство имен. Если порт 8080 понадобится использовать для какой-то другой службы, вы столкнетесь с неразрешимой проблемой. Эта проблема отсутствует при использовании варианта `bridge`, который мы обсудим в следующем разделе.

Давайте запустим новый образ CentOS и укажем вариант поддержки сети `host`:

```
docker run --network=host -d centOS
```

Теперь убедимся, что контейнер запустился, и войдем в него, как показано на рис. 8.9.

```
docker ps
docker exec -it kickass_minsky /bin/bash
```

```
[[root@cml ~]# docker ps
CONTAINER ID   IMAGE     COMMAND       CREATED          STATUS          PORTS     NAMES
9fa5e216d856   centos    "/bin/bash"   50 seconds ago   Up 49 seconds             kickass_minsky
[[root@cml ~]# docker exec -it kickass_minsky /bin/bash
[root@cml /]#
```

Рис. 8.9 ❖ Выполнен вход в контейнер CentOS

Пока все хорошо. Отыщем IP-адрес контейнера CentOS:

```
ifconfig | grep inet
```

Обратите внимание, что, как показано на рис. 8.10, наш контейнер получил IP-адрес 10.88.30.156.

```
[[root@cml ~]# ifconfig |grep inet
        inet 10.88.30.156 netmask 255.255.255.128  broadcast 10.88.30.255
        inet6 2001:420:1402:2033:21d:9ff:fe6d:5aea  prefixlen 64  scopeid 0x0<global>
        inet6 fe80::21d:9ff:fe6d:5aea  prefixlen 64  scopeid 0x20<link>
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 0.0.0.0
        inet6 fe80::42:bcff:fe24:eelb  prefixlen 64  scopeid 0x20<link>
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
[root@cml ~]#
```

Рис. 8.10 ❖ Проверка IP-адреса контейнера

Теперь откроем еще один терминал и определим IP-адрес хост-машины:

```
ifconfig | grep inet
```

Результат, изображенный на рис. 8.11, вполне ожидаем: контейнер получил тот же IP-адрес, что и хост: 10.88.30.156.

```
[root@cml /]#
[root@cml /]# ifconfig |grep inet
        inet 10.88.30.156  netmask 255.255.255.128  broadcast 10.88.30.255
        inet6 2001:420:1402:2033:21d:9ff:fe6d:5aea  prefixlen 64  scopeid 0x0<global>
        inet6 fe80::21d:9ff:fe6d:5aea  prefixlen 64  scopeid 0x20<link>
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 0.0.0.0
        inet6 fe80::42:bcff:fe24:eelb  prefixlen 64  scopeid 0x20<link>
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
[root@cml /]#
```

Рис. 8.11 ❖ Проверка IP-адреса хоста

По сути, этот конкретный контейнер действует в сети как физический сервер, что дает нам важное преимущество: его быстродействие очень близко к быстродействию, обеспечиваемому аппаратурой. На рис. 8.12 показано, как выглядит организация сети в этом случае.

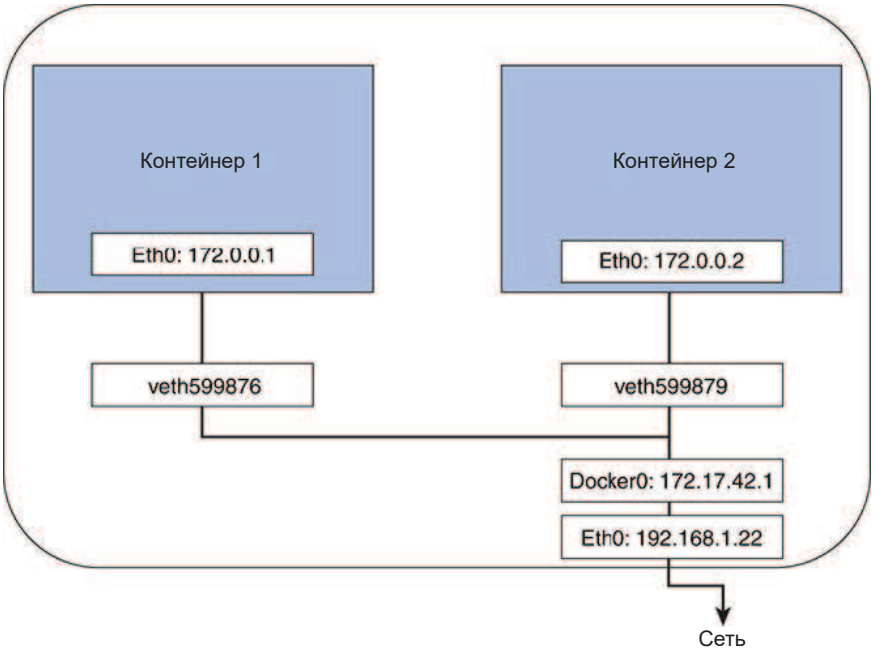


Рис. 8.12 ❖ Организация сети при выборе варианта `host`

bridge

Вариант `bridge`, также известный как *`docker0`*, используется по умолчанию, если явно не указать никаких параметров (`none` или `host`) в команде `run`. Не путайте его с мостом Linux, который мы обсудили выше, даже учитывая, что Docker использует его для поддержки варианта `bridge`.

Как вы уже, наверное, догадались по названию, `bridge` создает внутреннюю скрытую сеть для взаимодействия между контейнерами. Обратите внимание, что IP-адрес, присваиваемый контейнеру в этом случае, недоступен из-за пределов хоста. Зато есть возможность организовать отображение портов для доступа к контейнерам извне. Чтобы лучше понять, что все это значит, выполним следующую команду:

```
docker network inspect bridge
```

Как показано на рис. 8.13, раздел `Containers` пуст, потому что у нас пока не запущено ни одного контейнера.

```
ANUJSIN-M-T2H9:~ anujsin$ docker network inspect bridge
[
        {
                "Name": "bridge",
                "ID": "fe31184609981ba9602670fe4de2f48458fb057b6ff786be92550c5ad79f5bbb",
                "Created": "2017-07-08T19:08:03.0165057062",
                "Scope": "local",
                "Driver": "bridge",
                "EnableIPv6": false,
                "IPAM": {
                    "Driver": "default",
                    "Options": null,
                    "Config": [
                        {
                            "Subnet": "172.17.0.0/16",
                            "Gateway": "172.17.0.1",
                        }
                    ]
                },
                "Internal": false,
                "Attachable": false,
                "Ingress": false,
                "ConfigFrom": {
                    "Network": ""
                },
                "ConfigOnly": false,
                "Containers": {},
                "Options": {
                    "com.docker.network.bridge.default_bridge": "true",
                    "com.docker.network.bridge.enable_icc": "true",
                    "com.docker.network.bridge.enable_ip_masquerade": "true",
                    "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
                    "com.docker.network.bridge.name": "docker0",
                    "com.docker.network.driver.mtu": "1500"
                },
                "Labels": {}
        }
]
```

Рис. 8.13 ❖ Раздел `Containers` пуст

Давайте запустим пару контейнеров: один будет иметь параметр `bridge`, а для другого оставим выбор варианта по умолчанию:

```
docker run -d --network=bridge mysql
docker run -d --network=default tomcat
```

Теперь выполним команду `inspect` еще раз и отметим отличия:

```
docker network inspect bridge
```

Как можно видеть на рис. 8.14, оба контейнера оказались подключены через один и тот же мост и могут взаимодействовать друг с другом посредством своих IP-адресов.

```
[
    {
        "Name": "bridge",
        "Id": "fe311846093981ba9602670fe4de2f48458fb057b6ff786be92590c5ad79f5bbb",
        "Created": "2017-07-08T19:08:03.0165057062",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16",
                    "Gateway": "172.17.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "04c7ae5a7e73b249b729b2927244122b82e114d45e63291a260847ca8634ad48": {
                "Name": "wonderful_kalam",
                "EndpointID":
"30ad3accb403119dc09ae38207fdb1aa44e9f13647df980ce349181e2d2b01f7",
                "MacAddress": "02:42:ac:11:00:03",
                "IPv4Address": "172.17.0.3/16",
                "IPv6Address": ""
            },
            "3c056870b9a7db53de33914a12edb65415e5e14fd26c8916d629d8439e47d928": {
                "Name": "blissful_babbage",
                "EndpointID":
"1d365a32d1aea32e3e00d35e4d074f1cf8d293006dd88b3320010272920c5ae0",
                "MacAddress": "02:42:ac:11:00:02" ,
                "IPv4Address": "172.17.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true" ,
            "com.docker.network.bridge.enable_ip_masquerade": "true",
            "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
            "com.docker.network.bridge.name": "docker0",
            "com.docker.network.driver.mtu": "1500"
        },
        "Labels": {}
    }
]
```


Рис. 8.14 ❖ Контейнеры подключены через один и тот же мост и могут взаимодействовать друг с другом посредством своих IP-адресов

Вы можете подключиться к любому контейнеру и посмотреть, как выглядит сеть изнутри, выполнив команду `attach` и затем `ifconfig`, как это делалось выше

(см. рис. 8.11). Для проверки можно воспользоваться командой `ping` в контейнере 1 и указать ей IP-адрес контейнера 2. Итак, что же происходит за кулисами, когда мы выбираем вариант `bridge`?

В действительности Docker использует сетевую подсистему Linux. Все контейнеры, запущенные с параметром `bridge` или вообще без параметра, подключаются к единственному мосту (`docker0`) и благодаря этому получают возможность взаимодействовать друг с другом. При этом Docker помещает все необходимые записи в файл */etc/hosts* (*iptables* и пр.). На рис. 8.15 показано, как выглядит организация сети в этом случае.

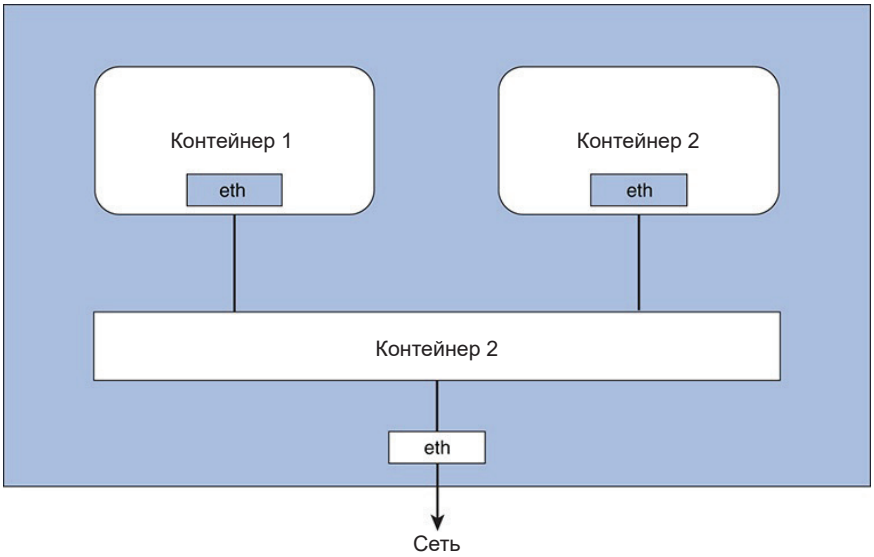


Рис. 8.15 ❖ Организация сети при использовании параметра `bridge`

Нестандартная организация сети

В дополнение к трем стандартным вариантам Docker также дает возможность определить нестандартную организацию сети для более точного управления соединениями. Для этого в состав Docker входят сетевые драйверы, которые вы можете использовать. Организация нестандартной сети дает полный контроль над соединениями и особую гибкость, о которых вы узнаете в этом разделе. Далее мы обсудим три основных нестандартных способа организации сети: с использованием нестандартного драйвера моста, драйвера оверлейной сети и базового сетевого драйвера MACVLAN.

Нестандартный драйвер сетевого моста

Этот драйвер очень похож на `docker0`, обсуждавшийся выше, но обладает дополнительными возможностями, такими как IPAM (IP Address Management – управление IP-адресами) и служба обнаружения. Кроме того, он обладает более высокой гибкостью.

Создать нестандартный сетевой мост можно командой

```
docker network create [OPTIONS] NETWORK
```

При необходимости в команде можно указать IP-адрес и подсеть. Если этого не сделать, Docker выберет следующую подсеть, доступную в пространстве приватных IP-адресов. Выполним эту команду:

```
docker network create --driver bridge pkNetwork
```

Проверим результат командой `ls`:

```
docker network ls
```

Только что созданную сеть pkNetwork можно видеть в списке сетей на рис. 8.16.

```
ANUJSIN-M-T2H9:~ anujsin$
ANUJSIN-M-T2H9:~ anujsin$
ANUJSIN-M-T2H9:~ anujsin$ docker network ls
NETWORK ID          NAME                  DRIVER    SCOPE
fe3118460998        bridge                bridge    local
133b7c28f2be        docker_gwbridge       bridge    local
d163ca8f2882        dockerservices_default bridge   local
4a8e216f9a47        host                  host      local
vf5dr217stoa        ingress               overlay   swarm
1bb6d94233c0        none                  null      local
33edf6b8d0de        pkNetwork             bridge    local
ANUJSIN-M-T2H9:~ anujsin$
```

Рис. 8.16 ❖ Список сетей

Проверим эту новую сеть, как делалось выше с `docker0`:

```
docker network inspect pkNetwork
```

Взгляните на рис. 8.17 и обратите внимание, что используется драйвер `bridge`. Это наш нестандартный сетевой мост.

```
ANUJSIN-M-T2H9:~ anujsin$ docker network inspect pkNetwork
[
        {
                "Name": "pkNetwork",
                "Id": "33edf6b8d0de1493a2d7dfde6762c1664ce59ae8b38aa7cae0a5726552d8edd9",
                "Created": "2017-07-08T21:59:15.8214098562",
                "Scope": "local",
                "Driver": "bridge",
                "EnableIPv6": false,
                "IPAM": {
                        "Driver": "default",
                        "Options": { },
                        "Config": [
                                {
                                        "Subnet": "172.20.0.0/16",
                                        "Gateway": "172.20.0.1"
                                }
                        ]
                },
                "Internal": false,
                "Attachable": false,
                "Ingress": false,
                "ConfigFrom": {
                        "Network": ""
                },
                "ConfigOnly": false,
                "Containers": { },
                "Options": { },
                "Labels": { }
        }
]
```

Рис. 8.17 ❖ Нестандартный сетевой мост

В настоящий момент у нас нет ни одного контейнера в этой сети. Так же, как в примере с мостом `docker0`, можете попробовать запустить пару контейнеров, указав сеть `pkBridge`, а затем проверить работу этой сети. За кулисами Docker автоматически определит все необходимые настройки.

Отображение портов

Как отмечалось выше, контейнеры Docker находятся в одной сети, посредством которой взаимодействуют друг с другом. Именно с этой целью все контейнеры включаются в одну общую сеть. Но доступ извне регулируется брандмауэром, т. е. контейнеры недоступны внешнему миру, если явно не разрешить внешние соединения. Такое разрешение реализуется отображением внутренних портов во внешние порты хоста в команде `run`. Также можно использовать комбинацию команд экспортирования и публикации, чтобы сначала экспортировать нужные порты в интерфейсы хоста, а потом опубликовать их.

Рассмотрим следующий пример:

```
docker run -d --network pkBridge -p 8000:80 --name tomcatPK -d tomcat
```

Скриншот на рис. 8.18 доказывает возможность доступа к контейнеру Tomcat извне с помощью браузера.

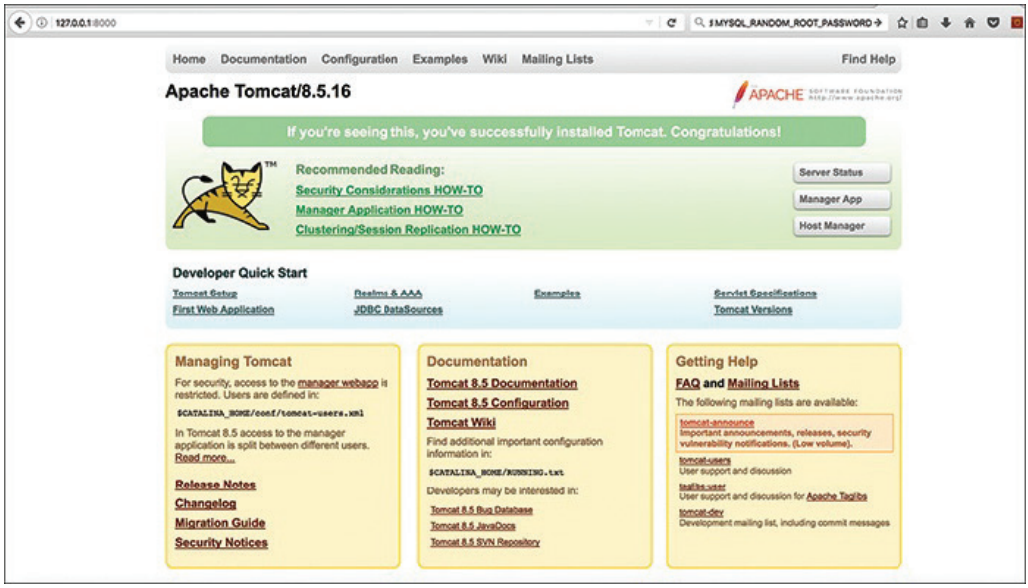


Рис. 8.18 ❖ Контейнер Tomcat доступен из внешнего браузера

А теперь разберем происходящее здесь. За кулисами механизм Docker добавляет правило iptables для трансляции сетевых адресов (Network Address Translation, NAT). Попробуйте вывести список правил iptables. В нем вы должны увидеть правило, определяющее отображение.

Возможно, вы обратили внимание, что драйвер `bridge` находится в локальной области видимости, т. е. его действие ограничивается единственным хостом. Другие два драйвера (драйвер оверлейной сети и базовый сетевой драйвер MACVLAN) доступны извне.

Драйвер оверлейной сети

Драйвер оверлейной сети позволяет организовать взаимодействие контейнеров, выполняющихся на разных хостах. Это достигается путем отделения сети контейнеров от лежащего под ним физического уровня и создания туннеля между хостами. Оверлейную сеть можно рассматривать как единую сеть, охватывающую несколько хостов. Все контейнеры, находящиеся в этой сети, могут взаимодействовать друг с другом, как если бы они находились на одном хосте. Организация такой сети показана на рис. 8.19.

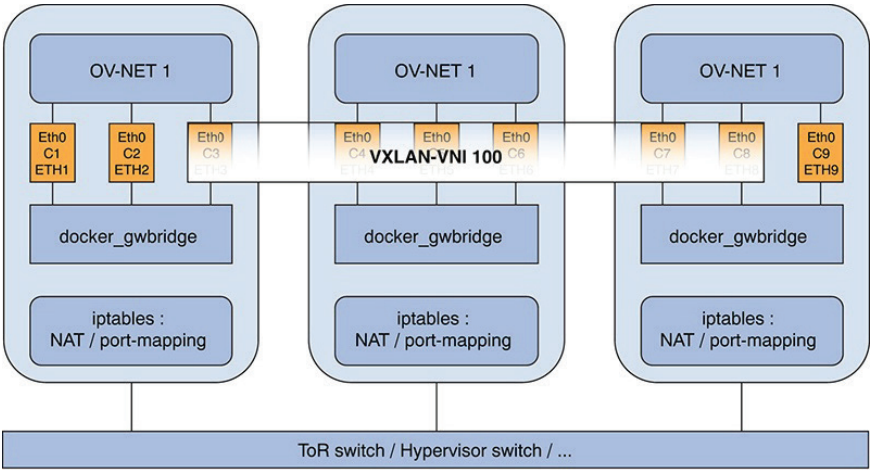


Рис. 8.19 ❖ Оверлейная сеть

Обратите внимание, что контейнер, находящийся в оверлейной сети, не сможет взаимодействовать с другими контейнерами, даже выполняющимися на том же хосте, если не включить их в эту сеть.

Для туннелирования Docker использует технологию VXLAN (Virtual eXtensible LAN – виртуальную расширяемую локальную сеть). Как и сеть `bridge`, оверлейную сеть можно создать, указав подсеть, а Docker автоматически определит необходимые настройки (мост Linux с соответствующими интерфейсами VXLAN) для подключения к хостам.

При этом настройки будут определены только на хостах, где требуется организовать подключение к контейнеру. Благодаря этому исключается распространение оверлейной сети на все машины, что отвечает требованиям распределенного развертывания микросервисов и организации соединений между ними.

Docker Swarm

На практике для запуска служб приложения часто создается кластер Docker. Управление таким кластером осуществляется с помощью инструмента Docker Swarm. На каждом узле такого кластера действует свой экземпляр механизма Docker, выполняющийся в режиме Swarm. Одной из ключевых особенностей Docker Swarm является организация сети с несколькими хостами посредством драйвера оверлейной сети, который мы только что обсудили. Когда запускается служба, использующая оверлейную сеть, управляющий узел Swarm автоматически включает в оверлейную сеть другие узлы, являющиеся частью этой службы.

Но Docker Swarm – это не единственный механизм управления кластерами. Существует еще целый ряд технологий с открытым исходным кодом, таких как Kubernetes и Mesos. При их использовании для организации оверлейной сети необходимо иметь хранилище пар ключ/значение для хранения необходимой информации, такой как результаты обнаружения служб, конечные точки, IP-адреса и т. д. С этой целью, например, можно использовать такие хранилища, как Consul, Zookeeper и т. д.

Базовый сетевой драйвер MACVLAN

Драйвер управления доступом к среде виртуальной локальной сети (Media Access Control Virtual Local Area Network, MACVLAN) – это еще один встроенный сетевой драйвер, более легковесный и простой, чем другие драйверы. Он не использует мост Linux и отображение портов, а подключает интерфейсы контейнеров непосредственно к интерфейсам хоста (eth) или производным интерфейсам.

По сути, все эти виртуальные интерфейсы находятся позади единственного физического интерфейса хоста. При такой организации каждый виртуальный интерфейс имеет уникальные MAC- и IP-адреса. Это позволяет контейнерам взаимодействовать с внешними ресурсами напрямую, минуя механизмы NAT и отображения портов, что делает этот драйвер более эффективным в сравнении с другими альтернативами.

Подобно оверлейным сетям, сети MACVLAN сегментированы. Контейнеры, находящиеся на одном хосте, но в разных сетях, не могут взаимодействовать друг с другом. На рис. 8.20 показано, как выглядит такая сеть.

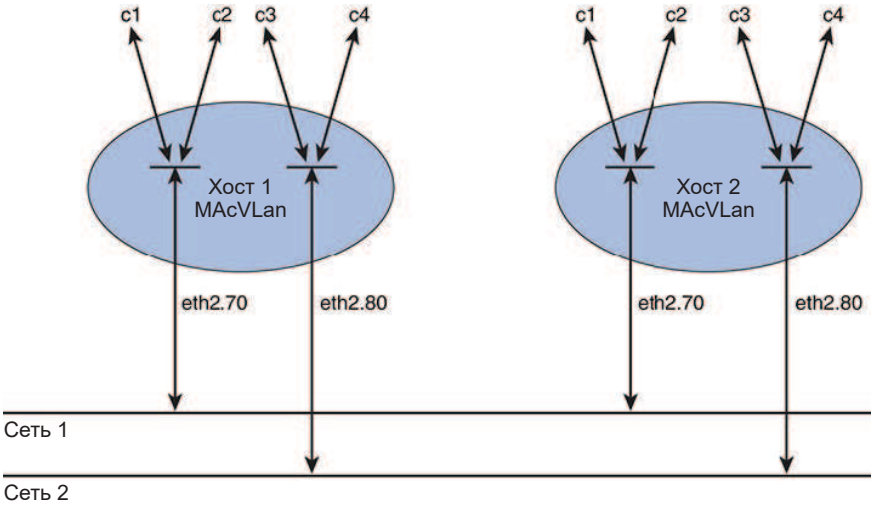


Рис. 8.20 ❖ Сеть MACVLAN

Как видите, Docker обладает удивительной гибкостью в организации сетевых взаимодействий. Если в вашем случае выдвигаются сложные требования, которые нельзя удовлетворить с помощью механизмов, обсуждавшихся выше, напишите свой плагин сетевого драйвера или используйте один из имеющихся плагинов, таких как Weave Net или Flannel.

Глава 9

Организация контейнеров

Управление горсткой контейнеров в корне отличается от управления набором контейнеров в промышленном масштабе, когда число контейнеров может достигать нескольких сотен и даже тысяч. Для поддержки управления контейнерами нам нужен простой способ развертывания и обслуживания контейнеров в промышленном масштабе. В этой главе мы познакомимся с некоторыми доступными решениями и особенностями работы каждого из них. Организация контейнеров – быстро меняющаяся область, поэтому, разобравшись с рассматриваемыми здесь технологиями и поняв различия между ними, обязательно посетите предлагаемые ссылки, чтобы познакомиться с последними разработками.

В области организации контейнеров создано множество решений, поэтому, как вы понимаете, порой непросто определить, какое из них лучше подходит в той или иной ситуации. Вот несколько популярных решений, широко используемых в этой сфере:

- Kubernetes;
- Mesos + Marathon;
- Docker Swarm.

Мы рассмотрим их в следующих нескольких разделах.

Kubernetes

Kubernetes – это проект с открытым исходным кодом, возглавляемый компанией Google. Google имеет богатый опыт развертывания контейнеров в большом масштабе и управления ими. Kubernetes – один из механизмов, помогающих запускать приложения в контейнерах в нужном месте и в нужное время, выделяя необходимые ресурсы и возможности, как показано на рис. 9.1.

Рассмотрим основные компоненты этого механизма.

Kubectl

Kubernetes имеет интерфейс командной строки `kubectl`. Он используется для запуска команд и взаимодействия с кластерами Kubernetes.

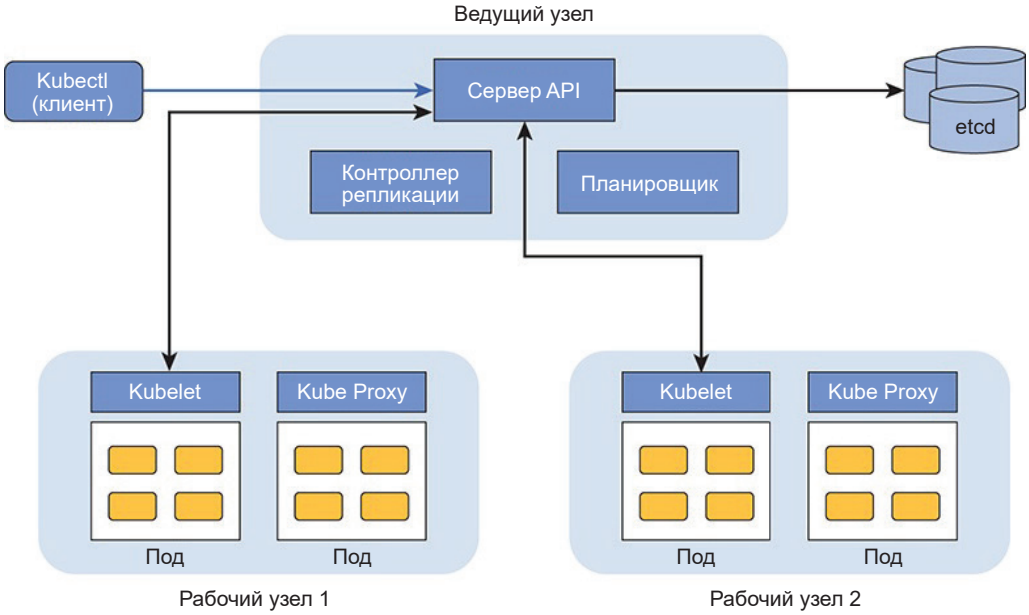


Рис. 9.1 ❖ Основные компоненты Kubernetes

Ведущий узел

Ведущий (или управляющий) узел – это мозг Kubernetes. Он координирует работу кластера с помощью некоторых вспомогательных служб. На этом узле выполняются сервер API, планировщик и контроллер репликации. Они управляют всеми действиями: планированием и поддержкой приложений в требуемом состоянии, масштабированием вверх или вниз и т. д.

Сервер API

Сервер API отвечает за поддержку REST API (Representational State Transfer – передача представительного состояния), используемого для взаимодействий с кластером Kubernetes. Все внешние взаимодействия между клиентом (`kubectl`) и кластером Kubernetes обслуживаются сервером API. Взаимодействия внутри кластера между рабочими узлами и ведущим узлом также обеспечиваются сервером API. Кроме того, это единственный компонент, обращающийся к распределенному хранилищу пар ключ/значение (etcd) для сохранения состояния объектов.

Выражаясь терминологией Kubernetes, мы используем объекты для описания того, чего мы хотим от кластера, или того, в каком состоянии должен находиться кластер. Например, объект может быть приложением, которое требуется запустить в кластере. Он может определять количество экземпляров приложения, которые должны выполняться в каждый конкретный момент времени, или описывать, как приложения должны взаимодействовать друг с другом.

Рассмотрим пример, как сервер API обрабатывает запросы. Допустим, нам нужно запустить контейнер Tomcat и в каждый момент иметь в кластере три выполняющихся экземпляра Tomcat:

```
kubectl run myTomcat --image=Tomcat --replicas=3
```

За кулисами `kubectl` отправит наше «намерение» (запрос) запустить три экземпляра сервера Tomcat в кластер (серверу API). Сервер API задействует компоненты планировщика и контроллер репликации для выполнения запроса и приведет кластер в нужное состояние.

Планировщик

Планировщик Kubernetes отвечает за размещение (планирование запуска) контейнеров на узлах кластера. С этой целью он создает «поды»[1] – простейшие единицы планирования в Kubernetes. Каждый под (pod) можно рассматривать как логический хост с собственным пространством имен и одним или несколькими контейнерами. Все контейнеры внутри пода (pod) совместно используют его пространство имен.

Когда сервер API получает запрос, он обращается к планировщику для размещения подов (pod) на узлах кластера. Перед размещением пода (pod) на рабочем узле планировщик проверяет следующие условия:

- какие узлы имеют достаточно ресурсов, таких как CPU и память, для запуска контейнеров в поде (pod);
- обладает ли хост достаточным количеством свободных портов для удовлетворения потребностей пода (pod);
- следует ли разместить под (pod) на определенном узле, чтобы избежать проблем с задержками в сети (привязка к узлу);
- следует ли распределять поды (pod) в кластере для обеспечения высокой доступности.

Как видите, при размещении подов (pod) планировщик должен учесть множество нюансов, чтобы принять грамотное и обоснованное решение. Это является одной из важнейших обязанностей планировщика Kubernetes. Он извлекает данные из подов (pod), описывающие требования (требуемую вычислительную мощность, объем памяти, необходимость обеспечения высокой доступности, привязку к узлу и т. д.), и выполняет свой алгоритм, определяющий лучший узел для пода (pod).

Вот как выглядит типичный процесс принятия планировщиком Kubernetes решения о выборе места для размещения указанного пода (pod):

1) определяет потребности пода (pod) в терминах ресурсов, привязки к узлу и т. д. и исследует список доступных узлов, извлекая информацию из базы данных etcd, тщательно отфильтровывая узлы, не соответствующие требованиям пода (pod).
   Например, узел имеет 12 Гбайт оперативной памяти, при этом на нем уже выполняется под (pod), потребляющий 8 Гбайт памяти. Соответственно, остаток памяти составляет 4 Гбайт. Если планировщику потребуется разместить еще один под (pod), требующий не менее 8 Гбайт памяти, тогда этот узел будет исключен из рассмотрения как не имеющий достаточного объема памяти для запуска указанного пода (pod);
2) узлы, оставшиеся после первого этапа, анализируются далее на соответствие другим критериям, после чего из них выбирается наилучший. На-

[1] От англ. *pod* – гроздь, связка, капсула. – *Прим. перев.*

пример, если приложение имеет два пода (pod), A и B, не рекомендуется запускать их оба на одном узле, потому что в случае остановки этого узла приложение может стать недоступным, что особенно нежелательно для микросервисов.
Другим примером может служить репликация. В этой ситуации по той же самой причине нежелательно, чтобы реплики (экземпляры одного и того же микросервиса) размещались на одном узле. Многие такие правила принимаются во внимание до того, как Kubernetes выберет наилучший узел для запуска указанного пода (pod);

3) после выбора узла планировщик размещает на нем под (pod) и запускает его.

Kubernetes имеет очень гибкую и расширяемую архитектуру. Если вам потребуется другой планировщик, лучше соответствующий вашим потребностям, можете подключить свой.

Контроллер репликации (управляющий контроллер)

Задача контроллера репликации – обеспечить одновременное выполнение в кластере заданного количества реплик пода (pod). Допустим, мы решили запустить три экземпляра контейнера Tomcat. В этом случае Kubernetes создаст 3 пода (pod) и запланирует их запуск в кластере, т. е. выполнит процедуру планирования и подберет лучшие узлы для запуска этих подов (pod). Теперь предположим, что один из узлов, где выполнялся под (pod) Tomcat, по какой-то причине остановился. В результате фактическое количество выполняющихся подов (pod) не будет соответствовать желаемому. Обнаружив эту разность, контроллер репликации обратится к планировщику Kubernetes и потребует запустить еще один экземпляр Tomcat где-нибудь в кластере вместе со всеми другими подами (pod), выполнявшимися на остановившемся узле.

А теперь представьте, что необходимость в выполнении трех экземпляров Tomcat отпала. Например, нагрузка снизилась, а вы решили сократить потребление ресурсов. В этом случае можно вновь запустить ту же команду с уменьшенным числом реплик:

```
kubectl run myTomcat --image=tomcat --replicas=2
```

Контроллер репликации обнаружит разницу и остановит лишние экземпляры (в данном случае – один), приведя кластер в желаемое состояние.

Рабочие узлы

Рабочими называются узлы, где размещаются и запускаются поды (pod). На каждом узле выполняется агент, который называется kubelet. Этот агент играет роль единой точки связи для каждого рабочего узла. Он отвечает за получение «заданий» от ведущего узла и их выполнение на рабочем узле. Под заданием в данном случае подразумевается запуск или остановка одного или нескольких подов (pod). Для передачи подробных сведений о подах (pod) планировщик обычно использует сервер API. После получения задания от ведущего узла kubelet обеспечивает успешный запуск требуемых подов (pod).

Также kubelet отвечает за предоставление информации о состоянии рабочего узла (его работоспособности, доступных ресурсах и т. д.) и каждого пода (pod), выполняющегося на этом узле. Эту информацию kubelet сохраняет в базе данных etcd посредством сервера API, откуда она затем извлекается планировщиком и используется для поиска доступных узлов (а также для определения объемов ресурсов, доступных на узлах) с целью запуска новых подов (pod). Эти же данные используются контроллером репликации для подсчета количества реплик, действующих в кластере. Если фактическое количество реплик не соответствует требуемому, контроллер предпринимает определенные шаги, чтобы привести кластер в нужное состояние.

Поды

Поды в Kubernetes имеют динамический характер. Иначе говоря, они создаются по мере необходимости, перемещаются на другие узлы, если другой узел по какой-то причине остановился, и масштабируются контроллером репликации в ответ на изменение трафика. Обсудим эту тему на конкретном примере, чтобы было понятнее.

Пример: кластер Kubernetes

Допустим, мы решили запустить в кластере Kubernetes три экземпляра MySQL, как показано на рис. 9.2.

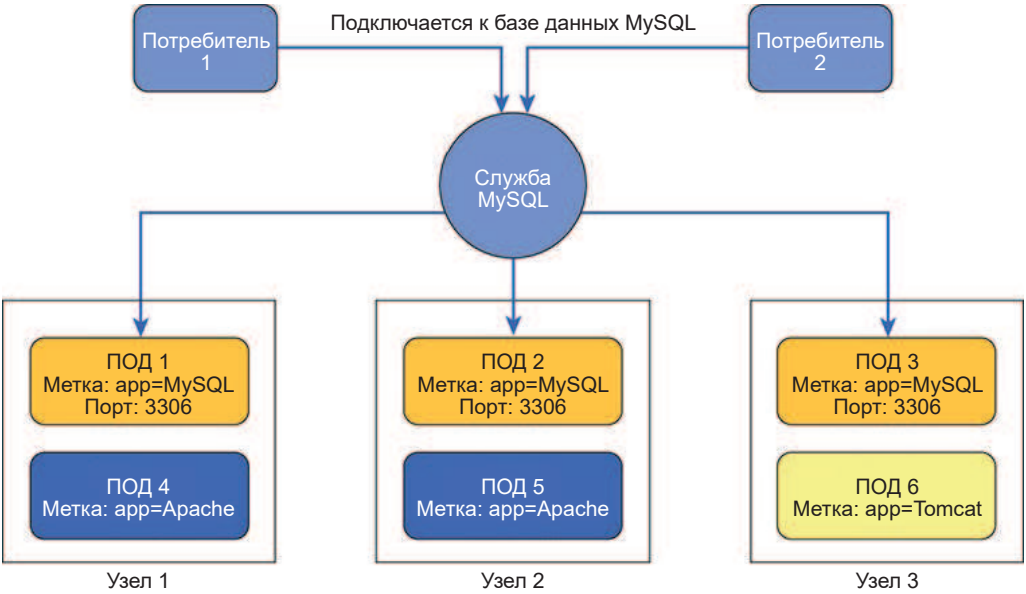


Рис. 9.2 ❖ Три экземпляра MySQL, выполняющиеся в кластере Kubernetes

Поды могут иметь метаданные, описывающие их. На рис. 9.2 можно видеть, что поды имеют метку `app=MySQL` и используют порт 3306. Поды 1, 2 и 3 отмечены совершенно одинаковыми метками. Таким образом, мы создаем логический набор

«связанных» подов, которые все вместе реализуют определенную услугу в кластере. В данном случае эти три пода предоставляют потребителям услуги базы данных.

Рассмотрим традиционный для трехуровневых приложений сценарий, когда сервер приложений, такой как Apache Tomcat (Потребитель 1), пытается получить данные из базы данных MySQL. В архитектуре микросервисов потребителю требуется узнать, где находится под MySQL. Как мы уже видели, поды размещаются на узлах кластера динамически. Таким образом, задача состоит в том, чтобы найти поды и получить возможность взаимодействовать с ними. На этом этапе в игру вступают службы Kubernetes.

Службы Kubernetes образуют уровень абстракции, служащий единой точкой входа для клиентских запросов, передаваемых связанной группе подов. Можно сказать, что службы являются интерфейсом доступа к соответствующим подам. Это очень мощная абстракция, потому что избавляет потребителя от необходимости точно знать, где находятся поды. Потребители могут просто обращаться к службам, каждая из которых имеет виртуальный IP-адрес и номер порта, не изменяющиеся в течение всего времени работы службы. Проще говоря, службы Kubernetes обеспечивают связь с наборами соответствующих им подов.

На сайте https://kubernetes.io вы найдете огромный объем документации, которая поможет вам установить и настроить Kubernetes. Наша цель в этом разделе – понять основную идею. Приступая к установке и настройке, всегда обращайтесь к самой свежей документации.

Apache Mesos и Marathon

Apache Mesos – открытый фреймворк для организации контейнеров, доказавший свою эффективность в крупномасштабных промышленных окружениях. Фреймворк Mesos напоминает ядро операционной системы, управляющее ресурсами в кластере. Он реализует архитектуру «ведущий/ведомый». Сам фреймворк Mesos управляет только ресурсами кластера. Проблемы планирования заданий должны решаться другими специализированными фреймворками, действующими над Mesos. Существует довольно много таких фреймворков, из которых наиболее известными считаются Marathon, Hadoop и Chronos. В этой главе мы рассмотрим фреймворк Marathon.

Архитектура Mesos включает ведущие узлы, ведомые узлы (*агенты*) и дополнительные фреймворки, как показано на рис. 9.3. Рассмотрим основные компоненты этой архитектуры.

Ведущий узел Mesos

Демон ведущего узла Mesos выполняется на управляющем узле. Он отвечает за управление демонами агентов на остальных узлах кластера, т. е. демон ведущего узла раздает задания демонам агентов. Демон ведущего узла также отвечает за обслуживание фреймворков, которые потребляют ресурсы кластера Mesos (вычислительную мощность, память, сетевой трафик, дисковое пространство). Поверх одного и того же кластера Mesos может работать произвольное число фреймворков. Эти фреймворки отвечают за ввод заданий в кластер Mesos. Задания передаются ведущему узлу, а тот распределяет их между агентами.

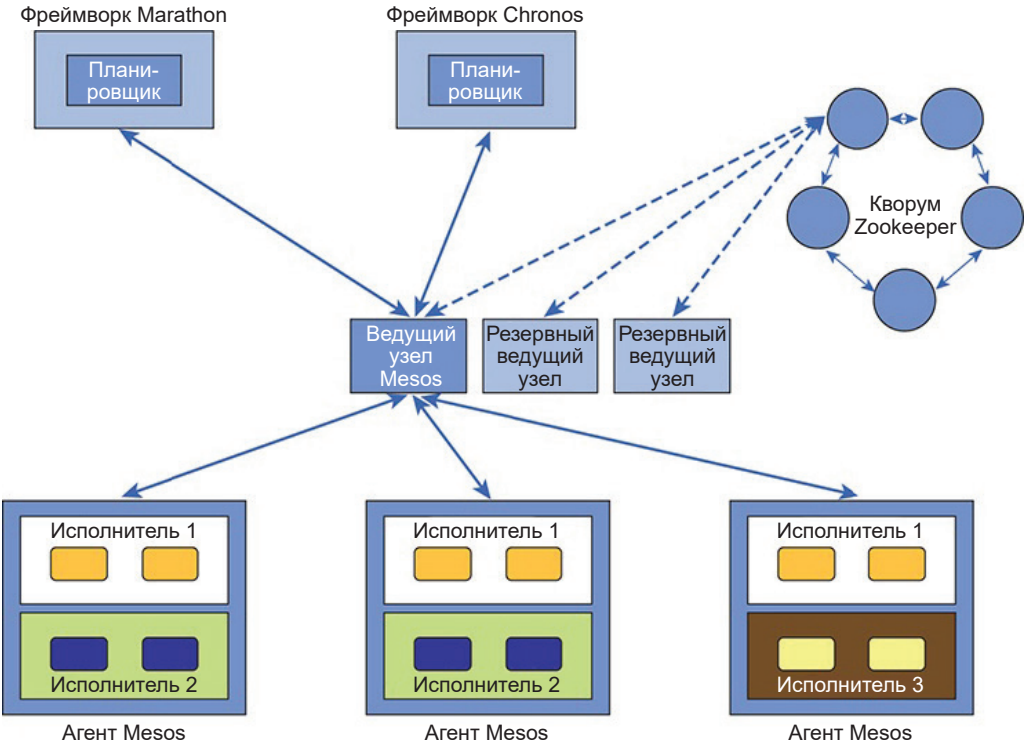


Рис. 9.3 ❖ Архитектура Mesos

Задача ведущего узла Mesos – обеспечить эффективное использование ресурсов кластера, таких как процессор и память, фреймворками, ожидающими запуска своих заданий. Это достигается путем распределения ресурсов кластера на так называемые *предложения*. Предложения содержат информацию об объеме ОЗУ и процессорного времени, которые доступны для выполнения задания. Предложения передаются зарегистрированным фреймворкам, которые, в свою очередь, могут принимать или отклонять их.

Предложения – это всего лишь способ, с помощью которого ведущий узел Mesos сообщает зарегистрированным фреймворкам информацию о доступных ресурсах. Например, предложение может включать такую информацию: «Для выполнения заданий доступно 12 Гбайт памяти и 8 ядер процессора». Получив такое предложение, фреймворк анализирует его, сопоставляя с заданием, которое нужно выполнить. Если предложение соответствует требованиям задания, тогда фреймворк принимает его, а если нет – отвергает.

Учитывая, что ресурсы одного и того же кластера Mesos может потреблять несколько фреймворков, возникает проблема учета доли ресурсов, переданных фреймворкам. Mesos весьма эффективно управляет ресурсами, позволяя настраивать этот процесс с применением правил. С их помощью администратор может определить, какой объем ресурсов можно выделить тому или иному фреймворку, основываясь на организационных приоритетах и/или важности заданий, которые данный фреймворк может выполнять в кластере Mesos.

Агенты

Агенты – это рабочие узлы, фактически выполняющие задания. На каждом рабочем узле выполняется демон ведомого узла. Этот демон отвечает за сбор статистики и ее отправку ведущему узлу Mesos.

Допустим, на вашей машине свободны 8 Гбайт ОЗУ и 4 ядра процессора. Агент отправит эту информацию ведущему узлу Mesos, который преобразует ее в предложение и передаст зарегистрированным фреймворкам. Задания, которые передаются фреймворками, фактически выполняются на рабочих узлах. Агенты получают задания от ведущего узла Mesos и запускают их с помощью *исполнителей*.

Исполнитель – это просто процесс или контейнер, способный выполнять команды оболочки, запускать контейнеры Docker и другие процессы. Mesos поддерживает упрощенную реализацию исполнителя, которая может выполнять команды оболочки и запускать контейнеры Docker. Однако многие фреймворки, такие как Marathon, имеют собственные реализации исполнителей, обладающих более широкими возможностями, чем исполнители по умолчанию, входящие в состав Mesos.

Фреймворки

Фреймворки являются потребителями ресурсов кластера. Как было показано выше, сам Mesos управляет только ресурсами, а задания запускаются фреймворками. Фреймворки включают 2 основных компонента: планировщик, самостоятельно регистрирующийся на ведущем узле Mesos и отвечающий за поиск предложений, их принятие или отклонение, и исполнитель, который фактически выполняет задания с помощью агентов. Если фреймворк не предоставляет свою реализацию исполнителя, он может по умолчанию использовать исполнителя, входящего в состав Mesos.

Пример: фреймворк Marathon

Допустим, нам нужно развернуть три экземпляра микросервиса каталога. Вот как можно записать соответствующий запрос и передать его фреймворку Marathon:

```
{
  "id": "catalog-svc",
  "cpus": 0.5,
  "mem": 8.0,
  "instances": 3,
  "container": {
    "type": "DOCKER",
    "Docker": {
      "image": "helpdesk/catalog-svc",
      "network": "BRIDGE",
      "portMappings": [
        {"containerPort": 80, "hostPort": 80, "protocol": "tcp"}
      ]
    }
  }
}
```

Обратите внимание, как здесь выполняется настройка сети в Docker. В соответствии с этим запросом нам требуется запустить в кластере три экземпляра микросервиса каталога. Раздел `container` описывает тип нужного нам контейнера (в данном случае – контейнер Docker). Этот раздел к тому же описывает, какой образ следует использовать для создания контейнера, а также необходимые номера портов. В дополнение ко всем этим деталям запрос также описывает, сколько памяти и процессоров необходимо для каждого экземпляра.

Вот как можно передать такой запрос фреймворку Marathon, если предположить, что он хранится в файле *application.json*:

```
curl -X POST http://hostip:port/v2/apps \
-d @application.JSON \
-H "Content-type: application/JSON"
```

Получив запрос, Marathon дождется предложения от ведущего узла Mesos (обратите внимание, что Marathon не хранит историю предложений). Как только появится предложение, соответствующее требованиям в запросе, он передаст в Mesos запрос, который затем будет передан процессам исполнителям на рабочих узлах для запуска контейнеров. В данном случае мы потребовали от Marathon запустить три экземпляра микросервиса каталога. Если по какой-то причине в кластере окажется меньше трех экземпляров микросервиса, Marathon продолжит работу с Mesos в цикле, чтобы обеспечить запуск дополнительных контейнеров для одновременного выполнения в кластере трех экземпляров.

Вы без труда сможете увеличить или уменьшить число экземпляров микросервиса, действующих в кластере. Для этого достаточно передать новый запрос с требуемым числом экземпляров. За более подробной информацией об установке и настройке обращайтесь к онлайн-документации проекта Mesos: https://mesosphere.com.

Docker Swarm

Docker Swarm – это механизм организации контейнеров, встроенный непосредственно в Docker. Swarm. Это всего лишь группа машин (механизмов Docker), выполняющих контейнеры в режиме Swarm. Swarm обеспечивает довольно эффективное управление узлами кластера. Рассмотрим основные понятия.

Узлы

Если выражаться простым языком, узел – это механизм Docker, входящий в состав кластера Swarm. Кластер Swarm имеет рабочие и управляющие узлы. Управляющие узлы – это мозг кластера Swarm. Они отвечают за управление рабочими узлами, где выполняются контейнеры.

В кластере Swarm имеется несколько управляющих узлов. Обычно их количество является нечетным числом, например 3, 5 и 7. Это помогает избежать проблемы с единой точкой отказа. Управляющие узлы выполняют так называе-

мый *алгоритм рафт-консенсуса* для «выбора» единственного лидера. Если по какой-то причине лидер остановится, на его роль будет выбран один из оставшихся управляющих узлов, благодаря чему удастся исключить возможность простоя из-за сбоев.

Службы

Служба – это просто описание того, что необходимо выполнить на узлах кластера. Определение службы включает:

- образ для запуска в контейнере;
- любые команды, которые должны быть выполнены в контейнере;
- число реплик, или экземпляров контейнера, которое требуется запустить.

Задание

Задание – это элементарная единица планирования в Swarm. Она содержит контейнер Docker и определяет команды для выполнения в контейнере. Управляющему узлу Swarm передается запрос на запуск службы, который просто определяет, какой контейнер следует запустить, а также количество его экземпляров. Затем управляющий узел передает задания (контейнер для запуска и список команд для выполнения в контейнере) рабочим узлам для выполнения. Он также следит за тем, чтобы в кластере было запущено указанное число реплик (экземпляров).

Мы, являясь конечными пользователями, только определяем наши намерения или желаемое состояние кластера, а управляющий узел Swarm обеспечивает достижение этого состояния и его поддержание.

Пример: кластер Swarm

А теперь займемся практикой и посмотрим, как организовать простой кластер Swarm. Самое замечательное, что для этого нам не понадобится устанавливать дополнительное программное обеспечение, кроме Docker. На момент написания этих строк самой свежей была версия Docker 17.06, и именно она будет использоваться в следующих примерах.

Запуск кластера Swarm

В этом примере мы запустим кластер Swarm с двумя узлами (одним управляющим и одним рабочим). На управляющем узле выполним следующую команду, чтобы инициализировать кластер Swarm:

```
docker swarm init --listen-addr 10.88.237.217:2377
```

В этой команде `10.88.237.217` – это IP-адрес машины, где выполняется команда, а `2377` – порт, на котором управляющий узел принимает запросы.

На рис. 9.4 можно видеть, что эта команда инициализировала кластер Swarm.

```
[root@swarm-master ~]# docker swarm init --listen-addr 10.88.237.217:2377
Swarm initialized: current node (ckmtounajpf06pglhv8jerlou) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join \
    --token SWMTKN-1-60v0219bqi48oeimlhbby39huseueu9redz94obklzzceazw43-6hlck485hgtifuw2u7lu3j2dy \
    10.88.237.217:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.
```

Рис. 9.4 ❖ Кластер Swarm инициализирован

На данном этапе в кластере пока нет ни одного рабочего узла. Все, что у нас есть, – единственный управляющий узел. Выведем список узлов, чтобы проверить, что у нас имеется:

```
docker node ls
```

Как показано на рис. 9.5, сейчас в кластере Swarm присутствует только 1 управляющий узел.

```
[root@swarm-master ~]# docker node ls
ID                            HOSTNAME        STATUS    AVAILABILITY    MANAGER STATUS
ckmtounajpf06pglhv8jerlou *   swarm-master    Ready     Active          Leader
```

Рис. 9.5 ❖ В кластере присутствует только 1 управляющий узел

Чтобы добавить рабочий узел, нужно перейти на машину, которая будет играть эту роль, и выполнить команду `swarm join`:

```
docker swarm join --token <tokenID> 10.88.237.217:2377
```

Как видите, чтобы сделать узел рабочим узлом кластера, достаточно лишь выполнить команду `swarm join` и указать IP-адрес и порт управляющего узла, как показано на рис. 9.6.

```
[root@swarm-worker1]# docker swarm join \
>     --token SWMTKN-1-60v0219bqi48oeimlhbby39huseueu9redz94obklzzceazw43-6hlck485hgtifuw2u7lu3j2dy \
>     10.88.237.217:2377
This node joined a swarm as a worker.
```

Рис. 9.6 ❖ Команда `swarm join` с IP-адресом и номером порта управляющего узла

Теперь еще раз выведем список узлов в кластере:

```
docker node ls
```

Сейчас в нашем кластере имеются 1 управляющий узел и 1 рабочий узел, как показано на рис. 9.7.

```
[root@swarm-master ~]# docker node ls
ID                            HOSTNAME            STATUS    AVAILABILITY    MANAGER STATUS
ckmtounajpf06pglhv8jerlou *   swarm-master        Ready     Active          Leader
p8c41ftcl9g0ugicxf7g5cnff     linux-dev.localdomain  Ready  Active
```

Рис. 9.7 ❖ Теперь в кластере имеются 1 управляющий узел и 1 рабочий узел

Создание службы

Чтобы создать и развернуть службу Tomcat в кластере Swarm, достаточно определить образ для создания контейнера и количество запускаемых экземпляров (реплик).

На рис. 9.8 можно видеть, что первоначально в кластере не выполняется ни одного контейнера (команда `docker ps -a` вернула пустую таблицу). Теперь создадим службу, передав образ (`tomcat : 7.0`, который уже имеется в репозитории) и указав, что управляющий узел должен запустить только один экземпляр (флаг `--replicas 1`).

```
[root@swarm-master ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
[root@swarm-master ~]#
[root@swarm-master ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
tomcat              7.0                 f8e399bdd39b        6 days ago          357MB
[root@swarm-master ~]#
[root@swarm-master ~]# docker service create --name TomcatService --replicas 1 tomcat:7.0
s7q73pnm2iko7njdmnlfnwp8o
Since --detach=false was not specified, tasks will be created in the background.
In a future release, --detach=false will become the default.
[root@swarm-master ~]#
[root@swarm-master ~]#
[root@swarm-master ~]# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
1525a4bd2b17        tomcat:7.0          "catalina.sh run"   5 seconds ago Up    3 seconds           8080/tcp            TomcatService.1.tlwr
pk878feb53r7wsh2lkbvw
[root@swarm-master ~]#
[root@swarm-master ~]# docker service ls
ID                  NAME                MODE                REPLICAS            IMAGE               PORTS
s7q73pnm2iko        TomcatService       replicated          1/1                 tomcat:7.0
```

Рис. 9.8 ❖ Первоначально в кластере не выполняется ни одного контейнера

После этого в кластере будет запущен 1 экземпляр Tomcat (как показывает команда `docker ps -a`, следующая за командой создания службы). Наконец, выполнив команду `docker service ls`, можно быстро убедиться, что служба TomcatService создана и запущена, а количество действующих экземпляров соответствует указанному.

Масштабирование

Сначала попробуем увеличить число выполняющихся экземпляров службы Tomcat:

```
docker service scale service TomcatService=2
```

Чтобы запустить дополнительный контейнер в кластере, потребуется некоторое время, как показано на рис. 9.9.

```
[root@swarm-master ~]# docker service scale TomcatService=2
TomcatService scaled to 2
[root@swarm-master ~]#
[root@swarm-master ~]# docker service ls
ID                  NAME                MODE                REPLICAS            IMAGE               PORTS
s7q73pnm2iko        TomcatService       replicated          1/2                 tomcat:7.0
[root@swarm-master ~]#
[root@swarm-master ~]#
[root@swarm-master ~]#
[root@swarm-master ~]# docker service ls
ID                  NAME                MODE                REPLICAS            IMAGE               PORTS
s7q73pnm2iko        TomcatService       replicated          2/2                 tomcat:7.0
```

Рис. 9.9 ❖ Увеличение числа выполняющихся экземпляров службы Tomcat

Чтобы уменьшить число выполняющихся экземпляров, достаточно простой команды

```
docker service scale TomcatService=1
```

За дополнительной информацией о параметрах настройки обращайтесь к онлайн-документации на странице проекта: https://docs.docker.com.

Обнаружение служб

Мы уже много раз говорили об обнаружении служб, поэтому остановимся и поговорим о том, почему это так важно. Если говорить простым языком, обнаружение служб – это механизм, помогающий узнать, как получить доступ к конкретной службе, такой как сервер баз данных, сервер кеша или любой другой сервер.

В то время когда приложения разворачивались на физических машинах, службы получали имена, по которым легко было определить, где они выполняются. Например, сервер баз данных для справочного приложения, выполняющийся на физической машине, мог бы иметь имя «helpdesk-db.domain.com». В этом случае если клиенту, такому как сервер приложений Tomcat, требуется доступ к базе данных, администратор обычно задает настройки, определяющие сервер баз данных, в виде свойств или конфигурационных файлов.

Однако с появлением потребности быстро раскручивать службы появились виртуальные машины (ВМ). Виртуальные машины открыли новые возможности, недоступные для физических машин, такие, например, как динамическое добавление узлов для обслуживания дополнительной нагрузки. В результате стали набирать популярность облачные технологии. Теперь, когда у нас есть несколько серверов, предлагающих одну общую услугу (например, кластер базы данных), как клиент может узнать, с каким сервером он должен взаимодействовать? В роли такой точки входа используется балансировщик нагрузки (например, NGINX или HAProxy), на котором определяются узлы, представляющие данную службу.

Допустим, что у нас есть балансировщик, распределяющий нагрузку между двумя серверами Tomcat. С увеличением трафика сценарии автоматизации могут динамически развернуть новую виртуальную машину с сервером Tomcat и обновить настройки балансировщика нагрузки. После этого балансировщик будет знать о существовании еще одного экземпляра службы Tomcat и направлять часть трафика ему. Клиентским приложениям не нужно знать о добавлении новой виртуальной машины, как не требуется и информация о ней, такая как IP-адрес и т. д. Клиент продолжает взаимодействовать с балансировщиком, который, в свою очередь, скрывает от клиентов любые изменения в службе Tomcat (например, добавление или удаление узлов).

Вернемся в наше время. Мы живем в эпоху контейнеров и микросервисов. Появление контейнеров усложнило проблему определения местоположения требуемой службы. Контейнеры могут запускаться и останавливаться очень быстро, и их местоположение отнюдь не статично, что затрудняет для клиентов определение местоположения службы в кластере. К счастью, для решения этой проблемы было создано множество инструментов.

Прежде чем перейти к знакомству с этими инструментами, рассмотрим пару типичных шаблонов обнаружения служб. Существуют минимум два способа обнаружения служб:

- **обнаружение на стороне клиента.** Реестр служб – это инструмент (или база данных), хранящий список всех служб с информацией об их местоположении (IP-адрес, порт и т. д.), как показано на рис. 9.10. Местоположения конкретных служб записываются в реестр в момент их запуска и удаляются из реестра в момент их остановки. Кроме событий запуска и остановки служб, используется некоторый механизм определения работоспособности, помогающий периодически проверять доступность той или иной службы.
  Главный недостаток этого подхода заключается в том, что клиент должен знать о существовании реестра служб и обращаться к нему, чтобы получить возможность обменяться данными со службой;

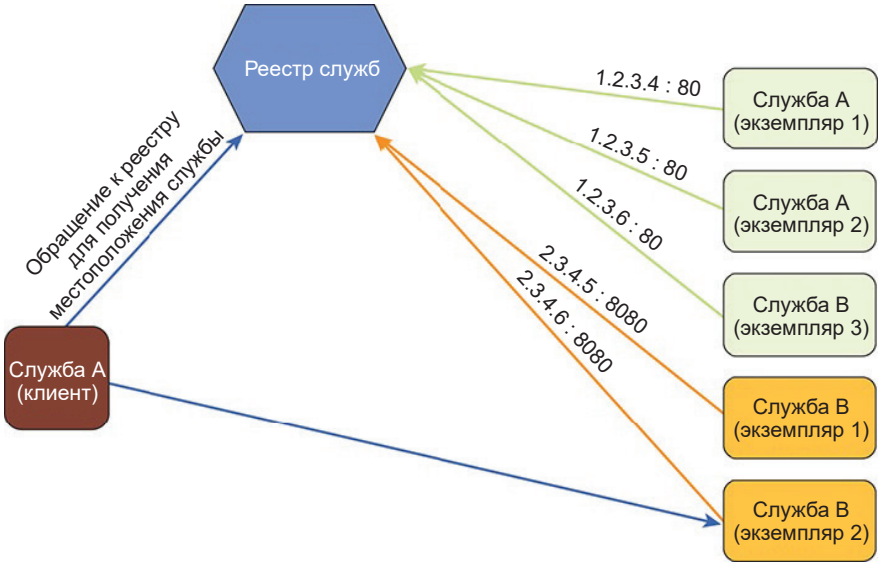


Рис. 9.10 ❖ Обнаружение служб на стороне клиента

- **обнаружение служб на стороне сервера.** В этом случае клиент посылает запрос непосредственно шлюзу API или балансировщику нагрузки, не заботясь о подключении к нужной службе. Балансировщик нагрузки сам обратится к реестру, узнает местоположение требуемой службы и передаст ей запрос для обработки, одновременно балансируя нагрузку между несколькими экземплярами службы, как показано на рис. 9.11. Классическим примером реализации такого подхода может служить популярный балансировщик Amazon ELB.
  Теперь представим, что мы настроили в Amazon Web Services кластер EC2 (Amazon Elastic Compute Cloud) с четырьмя узлами, где выполняется сервер приложений (Tomcat). Чтобы разделить трафик между этими экземплярами EC2 Tomcat, мы должны зарегистрировать их в ELB, указав такие сведения, как название экземпляра, номер порта, механизм проверки работоспособ-

ности службы (для этого должен использоваться механизм ELB) и частота такой проверки. После настройки ELB возьмет на себя всю работу по обработке входящих запросов и передаче их соответствующим экземплярам Tomcat.

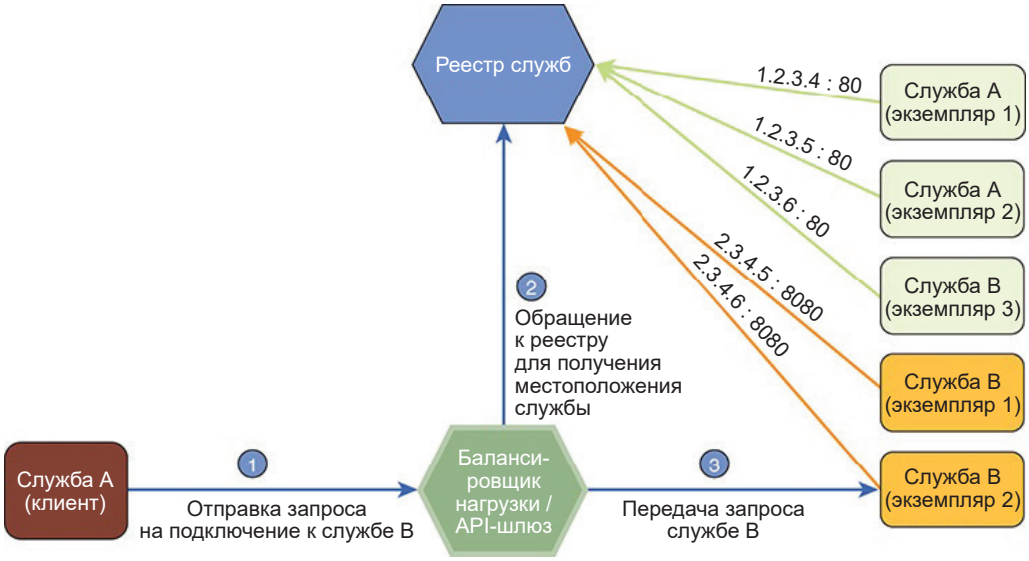


Рис. 9.11 ❖ Обнаружение служб на стороне сервера

Механизмы микросервисов и обнаружения служб не отделяются друг от друга. В реальности существует масса инструментов с открытым исходным кодом для обнаружения служб, включая Consul (HashiCorp), Zookeeper (Apache), etcd, SmartStack (AirBnB), Eureka (Netflix) и SkyDNS. Эти инструменты имеют много общего, отличаясь в основном размерами (бывают легковесные и тяжеловесные) и поддерживаемыми протоколами для передачи запросов (DNS, HTTP/TCP и т. д.).

Реестр служб

Реестр служб можно сравнить с телефонным справочником, т. к. в нем перечислены все службы, действующие в окружении. Он содержит информацию о местоположении службы в кластере (например, имя хоста и номер порта). Как мы уже знаем, микросервисы могут запускаться (чтобы справиться с увеличившейся нагрузкой) или останавливаться (при снижении нагрузки или в случае сбоя), а затем повторно запускаться на другом узле. Таким образом, местоположение микросервисов не является статичным и может меняться. Существуют минимум два способа передать местоположение данного микросервиса в реестр:

- **самостоятельная регистрация.** Когда микросервис сам отправляет информацию о себе в реестр, как показано на рис. 9.12. Например, Consul – популярная реализация реестра служб, которая предлагает свой программный интерфейс (API) для взаимодействий. При самостоятельной регистрации каждый микросервис должен передать в Consul API информацию о своем

местоположении. Согласно рекомендациям по организации и реализации микросервисов, каждый микросервис должен сосредоточиться на решении единственной задачи. Однако принуждение микросервисов к отправке информации об их местонахождении нарушает правило единственной ответственности. По этой причине самостоятельная регистрация редко используется на практике;

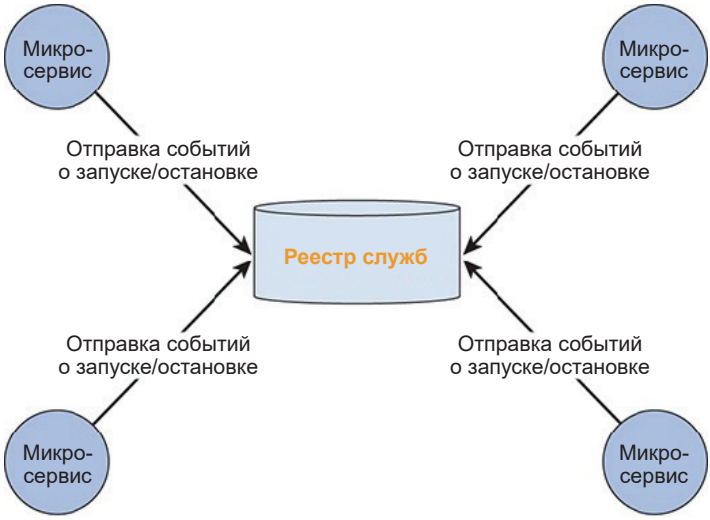


Рис. 9.12 ❖ Самостоятельная регистрация службы в реестре

- **регистрация внешними или сторонними инструментами.** Регистрация в реестре с использованием внешних инструментов – лучший способ по одной простой причине: микросервисы могут сосредоточиться исключительно на своей прямой обязанности и не заботиться об отправке в реестр информации о своем местоположении. Этот подход обеспечивает четкое разделение обязанностей. Если завтра вы решите заменить механизм обнаружения микросервисов и хранения реестра, то сможете сделать это, не прикасаясь к коду самих микросервисов.

Рассмотрим последний способ регистрации (с использованием сторонних инструментов) на примере того же реестра Consul. Registrator (https://github.com/gliderlabs/registrator) – компонент с открытым программным кодом, который служит мостом между реестром служб и контейнерами Docker. Он автоматически регистрирует службы, следя за запуском и остановкой контейнеров Docker.

Когда контейнер Docker запускается или останавливается, он генерирует событие (уведомление), при этом любой сторонний инструмент может подписаться на эти события и выполнять определенные действия при их получении. Registrator просто прослушивает события, рассылаемые механизмом Docker, и, подобно команде `docker inspect`, исследует контейнеры, чтобы опознать службы, которые они реализуют. Затем он подключается к реестру служб (например, Consul, etcd, SkyDNS2) и посылает ему информацию об обнаруженной службе.

Как показано на рис. 9.13, компонент Registrator устанавливается на все рабочие узлы, где могут выполняться контейнеры. Он настраивается для взаимодей-

ствия с реестром служб (где хранится актуальная информация о службах, выполняющихся в кластере), куда должна отправляться информация об обнаруженных службах. Когда на каком-то узле происходит запуск или остановка контейнера, Docker посылает соответствующее событие, а компонент Registrator, действующий на этом узле, перехватывает его и исследует контейнеры, чтобы получить дополнительную информацию о службах.

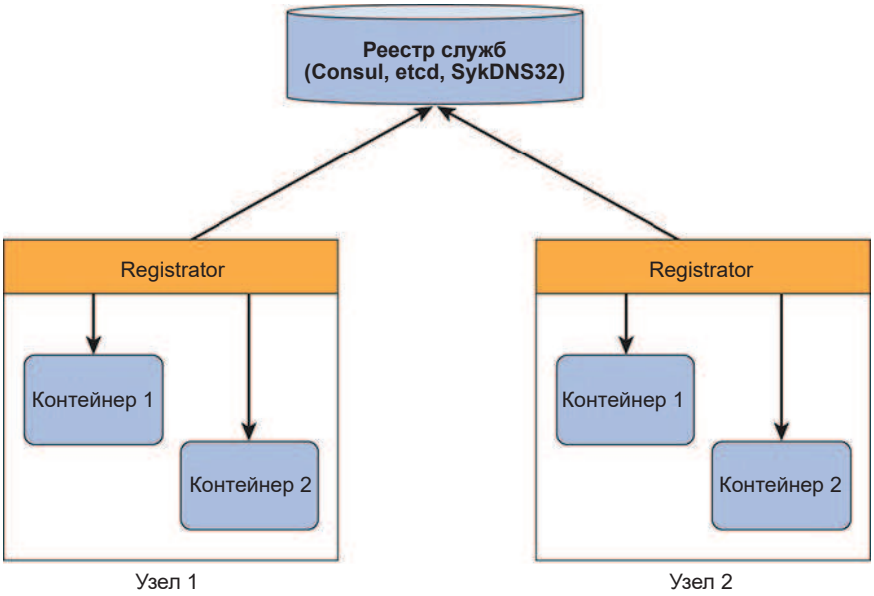


Рис. 9.13 ❖ Registrator устанавливается на все рабочие узлы, где могут выполняться контейнеры

На этом мы завершаем обсуждение вопросов развертывания и обнаружения служб. Эти знания будут широко использоваться для перевода примера проекта на рельсы Docker в третьей части книги.

Глава 10

Управление контейнерами

Теперь, зная, как организовать и масштабировать контейнеры и как реализовать поддержку сетевых взаимодействий между ними, перейдем к обсуждению ситуаций, когда что-то идет не так, как надо. Нередки случаи, когда в промышленном окружении выполняются сотни и тысячи контейнеров, при этом необходимо знать, как эффективно управлять ими. По этой причине мы завершим наше глубокое погружение в контейнеры обсуждением деталей, связанных с мониторингом и управлением контейнерами, включая сбор журналов и информации о ресурсах, с использованием некоторых систем мониторинга для кластерных окружений. Но для начала рассмотрим общие аспекты мониторинга контейнеров и постараемся разобраться в особенностях, отличающих его от мониторинга обычных приложений.

Мониторинг

Мониторинг окружения с контейнерами реализовать несложно. Сложно добиться приемлемой скорости и качества мониторинга. Рынок инструментов для мониторинга и управления физическими хостами, сетями и виртуальными машинами достиг значительной зрелости. Но контейнеры появились относительно недавно, поэтому рынок соответствующих инструментов мониторинга все еще находится в стадии становления. Отличие мониторинга контейнеров обусловлено такими аспектами и проблемами, как:

- **среда развертывания.** Организация может запускать часть контейнеров на своем оборудовании, в своих вычислительных центрах, а часть – на виртуальных машинах, пользуясь услугами таких провайдеров, как AWS Managed Service Partners, что добавляет сложности в организацию мониторинга;
- **масштабируемость контейнеров.** Монолитные приложения могут запускаться на физической машине или на нескольких виртуальных машинах. Практика применения контейнеров требует, чтобы в одном контейнере выполнялась одна служба, а приложение может состоять из сотен и тысяч служб. Это означает необходимость запуска сотен и тысяч контейнеров. В архитектуре микросервисов для масштабирования приложений необходимо организовать автоматическое изменение числа контейнеров вверх и вниз, учитывая фактические потребности;
- **скорость изменения.** В отличие от физических хостов и виртуальных машин, продолжительность жизни одного контейнера может варьироваться

от нескольких секунд до нескольких суток. Закончив решение задачи, контейнер завершается;

- **разнообразие используемых инструментов.** Контейнеры обеспечивают высокую скорость и эффективность, но в ущерб простоте. Для развертывания, управления и обнаружения контейнеров приходится привлекать множество инструментов. Например, вы можете использовать один или несколько инструментов организации контейнеров, таких как Docker Swarm, Kubernetes или Mesos, определять настройки сети, число выполняющихся экземпляров контейнеров и т. д., а инструменты организации обеспечат запуск, остановку и управление контейнерами с учетом доступных ресурсов на хостах. Каждый раз, когда запускается новый контейнер, он получает новый IP-адрес. Из-за всего этого становится очень сложно настроить мониторинг и сбор параметров;
- **распределенные данные.** Данные должны собираться из разных мест и объединяться в одном хранилище, чтобы проще было разбираться в них и находить потенциальные проблемы. Docker предлагает некоторые возможности для получения таких данных с целью упреждающего мониторинга контейнеров и всей системы в целом.

Существует множество инструментов на выбор от разных производителей, и каждый имеет свои достоинства. Недавно участниками проекта Docker была запущена программа партнерства в развитии технологической экосистемы (Ecosystem Technology Partner, ETP), в рамках которой сторонним компаниям предлагается интегрировать свои инструменты мониторинга с Docker API. Познакомиться со списком компаний-участниц и их инструментами можно на сайте: https//www.docker.com.

А теперь начнем обсуждение доступных возможностей по извлечению журналов и параметров. Эти данные можно экспортировать в уже имеющиеся у вас инструменты мониторинга и с их помощью создать свои информационные панели (дашборды).

Журналирование

В промышленном окружении, которое поддерживает несколько приложений на нескольких кластерах с несколькими копиями действующей службы, может иметься очень большое число действующих контейнеров. Иногда что-то может пойти не так. Когда это происходит, журналы превращаются в важный инструмент, помогающий найти источник проблемы. Например, как вы помните, типичное крупномасштабное приложение может состоять из тысяч микросервисов. Контейнеры Docker прекрасно подходят для запуска такого большого числа микросервисов благодаря достоинствам, которые мы уже обсудили. Вопрос в том, как организовать управление журналированием, учитывая, что каждый контейнер выводит в журнал все, что попадает в `stdout` и `stderr`? Как синхронизировать все эти журналы и где их разметить, чтобы сделать устранение проблем простым и эффективным?

Для этой цели Docker предоставляет разные драйверы, которые помогут упростить нашу задачу. Драйверы позволяют извлекать информацию из журналов кон-

тейнеров и служб. Они отличаются способами представления и форматирования информации, а также используемыми механизмами ее передачи. В качестве примеров таких драйверов можно назвать json-file, Syslog, Splunk, Amazon CloudWatch Logs и др. Мы обсудим все эти варианты, но за более подробной информацией советую обратиться к онлайн-документации Docker.

На момент написания этих строк поддерживались следующие драйверы журналирования:

- **json-file.** Драйвер, используемый демоном Docker по умолчанию. Все контейнеры используют json-file, если явно не определено иное. Выводимый им файл журнала имеет формат JSON;
- **None.** Журналирование отключено;
- **Syslog.** Посылает журналируемые сообщения серверу syslog, локальному или удаленному. Как уже обсуждалось выше, вы можете изменить файл *daemon.json* на хосте и настроить использование драйвера syslog, определив необходимые параметры в разделе `options`. То же самое можно сделать на уровне контейнера. Syslog отправляет все сообщения в одно место, помогая тем самым в поиске проблем. Однако этого недостаточно, когда имеется несколько сотен контейнеров, как в случае с микросервисами;
- **awslogs.** Отправляет записи в Amazon CloudWatch Logs. Для этого вы должны выбрать драйвер awslogs и определить необходимые настройки;
- **Splunk.** Отправляет записи в систему Splunk с помощью коллектора событий. Для этого вы должны выбрать драйвер Splunk и определить ключ и URL Splunk в файле конфигурации или в команде запуска контейнера;
- **Journald.** Отправляет записи в системный журнал. Для этого вы должны выбрать драйвер journald. Записи из журнала в этом случае можно извлекать с помощью `journalctl` или командой `docker logs`;
- **gcplogs.** Отправляет записи в систему Google Cloud Platform, поддерживающую поиск и анализ сообщений. Для этого вы должны выбрать драйвер gcplogs. Также можно определить настройки, включая управляющие подробностью журналирования;
- **GELF.** Отправляет записи в конечные точки Graylog Extended Log Format (GELF), например на сервер Logstash. Для этого вы должны выбрать драйвер gelf и определить некоторые настройки. GELF часто используется в связке ELK (Elasticsearch, Logstash и Kibana).

Как упоминалось выше, по умолчанию используется драйвер json-file. Проверить выбранный драйвер можно командой

```
docker info | grep 'Logging Driver'
```

В данном случае вы должны увидеть результат – `Logging Driver: json-file`.

Давайте запустим контейнер Ubuntu и проверим выбранный драйвер журналирования:

```
docker run -it ubuntu:latest sh
```

Откройте еще один терминал, найдите идентификатор контейнера и скопируйте его:

```
docker ps
```

Теперь запустите следующую команду, чтобы узнать, какой драйвер журналирования используется контейнером Ubuntu:

```
docker inspect -f '{{.HostConfig.LogConfig.Type}}' ec5e917eb9b0
```

Вы должны увидеть результат, изображенный на рис. 10.1.

```
[root@linux-dev pkocher]# docker inspect -f '{{.HostConfig.LogConfig.Type}}' ec5
e917eb9b0
json-file
```

Рис. 10.1 ❖ Определение драйвера журналирования с помощью команды `docker inspect`

Выбрать другой драйвер журналирования можно на уровне демона или контейнера. Чтобы выбрать другой драйвер на уровне демона, измените значение параметра `log-driver` в файле *daemon.json*, который в Linux находится в каталоге */etc/Docker*. Вот как выглядит его структура:

```
"log-driver":
"log-opts":{ options like syslog server info, etc. }
```

Чтобы выбрать другой драйвер на уровне контейнера, нужно указать драйвер в команде `run`, как показано ниже.

Также есть возможность вообще отключить журналирование. Давайте снова запустим наш контейнер Ubuntu с параметром `none` и вновь выполним команду `logs`.

```
docker run -it --log-driver none ubuntu:latest sh
```

Теперь выполним пару команд, чтобы сгенерировать несколько записей, как показано на рис. 10.2.

```
# ps
  PID TTY          TIME CMD
    1 ?        00:00:00 sh
    8 ?        00:00:00 ps
# ls
bin  boot dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
#
```

Рис. 10.2 ❖ Создание нескольких записей

Теперь проверим журнал:

```
docker ps # скопируйте идентификатор контейнера
docker logs 73c1b74d6091
```

Можете убедиться сами, что журнал пуст, потому что журналирование для данного контейнера было отключено. Это не влияет на другие контейнеры, потому что изменения были произведены на уровне контейнера.

Имейте в виду, что обсуждаемые варианты журналирования не относятся к приложениям или службам, которые ничего не выводят в потоки `stderr` и `stdout`. Кроме того, некоторые из упомянутых драйверов в своей работе опираются на службы, действующие на хосте, что может быть рискованно.

Также следует помнить, что с увеличением числа контейнеров в приложении вам может потребоваться использование очень сложной системы централизованного журналирования, хранящей всю информацию, начиная с системных данных о доступных процессорах и памяти и заканчивая данными об особенностях работы приложений. Таким образом, разрабатывая приложение, вы должны предусмотреть правильную маркировку сообщений для идентификации фрагментов кода, которые их производят. Подобная централизованная система журналирования должна поддерживать такие функции, как фильтрация, индексирование, классификация, сортировка и поиск, чтобы упростить и ускорить поиск неполадок в приложениях и контейнерах.

Сбор параметров

В этом разделе мы обсудим механизмы сбора параметров, использующие основные утилиты Docker, а также некоторые инструменты с открытым программным кодом, которые можно применяать для решения задачи мониторинга. Начнем с `docker stats`.

docker stats

Команда `docker stats` возвращает оперативные данные о работе контейнера, выполняющегося в хост-системе в данный момент времени:

```
docker stats [options] [containers]
```

Команде можно передать идентификаторы конкретных интересующих контейнеров или ключ `-a`, чтобы получить сведения о всех контейнерах. Если вызвать команду без параметров, она вернет сведения о всех контейнерах.

Выполним эту команду:

```
docker stats
```

На рис. 10.3 показана статистика использования ресурсов, полученная данной командой.

CONTAINER	CPU%	MEM USAGE / LIMIT	MEM%	NET I/O	BLOCK I/O	PIDS
b4831186f0d4	0.00%	1.98 MiB / 47.08 GiB	0.00%	0 B / 0 B	2.12 MB / 0 B	1

Рис. 10.3 ❖ Результат выполнения команды `docker stats`

Нажмите комбинацию **Ctrl**+**C**, чтобы выйти из потока. При желании можно настроить формат вывода с помощью параметра `--format`. Например:

```
docker stats --format "table {{.Name }} \t {{.ID }} \t {{.CPUPerc}} \t {{.MemUsage}}"
```

Параметр `--format` принимает следующие аргументы:

- `.Name` добавляет имя контейнера;
- `.ID` добавляет идентификатор контейнера;
- `.CPUPrec` добавляет процент используемого процессорного времени;
- `.MemUsage` добавляет объем используемой памяти;

- `.NetIO` добавляет используемую ширину сетевого канала ввода/вывода;
- `.BlockIO` добавляет используемые блокировки ввода/вывода;
- `.MemPerc` добавляет процент используемой памяти;
- `.PIDs` добавляет число идентификаторов процессов.

Как можно видеть на рис. 10.4, этот параметр позволяет получить довольно полную картину о расходовании ресурсов хоста.

```
NAME              CONTAINER ID                                                     CPU %     MEM USAGE / LIMIT
nervous_bhaskara  b4831186f0d47a248caabf44a4b7cf469bf0d6e6c7c7f975b0b93195461l    0.00%     1.98 MiB / 47.08 GiB
```

Рис. 10.4 ❖ Применение параметра `--format` для получения информации об используемых ресурсах хоста

Конечные точки API

Команда `docker stats` дает отличную возможность получить поток оперативной информации. Но, кроме нее, имеется также REST API, который можно использовать для создания дашбордов, информирующих о состоянии кластеров. Эти API открывают доступ к таким же и даже более детальным потокам оперативной информации, что и `docker stats`.

```
GET /containers/(ID/Name)/stats
```

Обратиться к этой конечной точке API, чтобы получить статистику о расходовании ресурсов действующим контейнером, можно, например, командой:

```
curl --unix-socket /var/run/docker.sock -X GET
'http:/v1.24/containers/<container ID>/stats'
```

Эта конечная точка, как и команда `docker stats`, возвращает непрерывный поток данных, обновляющийся каждую секунду. Однако программа должна максимально осторожно использовать эту конечную точку, чтобы не нанести ущерба общей производительности. Например, было бы неплохо организовать получение среза через более долгие интервалы времени. Надеемся, что в ближайшее время эта проблема будет решена с помощью какого-нибудь флага в API.

Как мы узнали благодаря предыдущим главам, одним из приемов эффективного мониторинга контейнеров является их маркировка тегами с осмысленными строками. Теги можно определять при сборке образов. Благодаря этому появляется возможность работать не с конкретными хостами или идентификаторами контейнеров, а с именами в тегах.

За более подробной информацией о REST API обращайтесь к документации на сайте docker.com.

cAdvisor

cAdvisor, также известный как Container Advisor, – это решение для мониторинга, разработанное в компании Google. Оно позволяет получить подробные данные о расходовании ресурсов в контейнере и его производительности в графическом интерфейсе. Этот продукт сам является контейнером, который можно развернуть

на хостах. cAdvisor собирает данные из всех контейнеров, действующих на его хосте, объединяет их и подготавливает для вас. Кроме того, cAdvisor предлагает также программный интерфейс для получения этих же данных. Для быстрого опробования cAdvisor на своем компьютере с Docker можно воспользоваться готовым образом, включающим все необходимое для этого (за дополнительной информацией обращайтесь по адресу: https://github.com/google/cadvisor). Одного экземпляра cAdvisor достаточно для мониторинга всей машины. Нужно лишь выполнить следующую команду:

```
sudo docker run \
--volume=/:/rootfs:ro \
--volume=/var/run:/var/run:rw \
--volume=/sys:/sys:ro \
--volume=/var/lib/docker/:/var/lib/docker:ro \
--publish=8080:8080 \
--detach=true \
--name=cadvisor \
google/cadvisor:latest
```

Она запустит cAdvisor в фоновом режиме. Графический веб-интерфейс в этом случае будет доступен по адресу: *http://localhost:8080*.

Два последних решения для получения информации просты и удобны, но они ориентированы на работу с одним хостом. Да, они имеют программный интерфейс, опираясь на который, можно реализовать централизованную систему мониторинга, но есть множество других систем, ориентированных на мониторинг всего кластера в целом. Далее мы рассмотрим пару таких систем. Это очень быстро меняющаяся область, поэтому мы познакомимся лишь с ключевыми идеями. Вам следует самостоятельно продолжить поиски в интернете, чтобы отыскать более перспективные решения.

Инструменты мониторинга кластеров

Рассмотрим несколько инструментов с открытым исходным кодом, предназначенных для мониторинга кластеров.

Heapster

Heapster – еще одно решение, разработанное в Google для мониторинга на уровне кластера. Оно основано на cAdvisor и хорошо подходит для случаев, когда для организации кластера используется Kubernetes. На момент публикации этой книги Heapster поддерживал только Kubernetes и CoreOS.

В Kubernetes решение cAdvisor интегрировано в двоичный файл Kubelet. Как уже говорилось выше, cAdvisor автоматически обнаруживает контейнеры на своем хосте и собирает статистику об использовании ресурсов. Kubelet получает все эти данные от cAdvisor, объединяет их и передает в Heapster через REST API. Heapster, в свою очередь, обрабатывает и группирует данные и передает указанному процессу для визуализации. В настоящее время в роли таких процессов визуализации поддерживаются InfluxDB и Grafana.

Более подробную информацию ищите на таких сайтах, как https://kubernetes.io/ и https://github.com/kubernetes/heapster.

Prometheus

Prometheus – это решение с открытым исходным кодом для мониторинга кластеров и рассылки предупреждений. Оно немного отличается от других решений тем, что реализует модель извлечения данных. Согласно этой модели, агент мониторинга рассылает целевым агентам настройки, определяющие частоту сбора данных, продолжительность их хранения, а также настройки, касающиеся уведомлений. Приложение должно предоставлять доступ к этим данным, а не рассылать их автоматически. Дополнительно поддерживается гибкий язык запросов PromQL. Прежде чем рассматривать, как Prometheus работает с Docker, познакомимся с его основными компонентами, такими как:

- **сервер Prometheus.** Этот компонент собирает и хранит данные, следуя заданным правилам. Его можно также настроить на создание предупреждений, которые доступны с помощью Alertmanager;
- **пользовательский веб-интерфейс.** В этом качестве Prometheus использует решение Grafana, с помощью которого можно создавать интерактивные дашборды;
- **шлюз для передачи информации.** Эта промежуточная служба позволяет передавать параметры работы короткоживущих служб серверу Prometheus, извлечь которые иным способом невозможно. Будьте осторожны, используя эти шлюзы, потому что они могут превратиться в единственные точки отказа для подобных короткоживущих источников;
- **экспортеры.** Специальные плагины или библиотеки, используемые для экспортирования параметров в системы, не поддерживающие работу с Prometheus. Вот несколько примеров таких экспортеров:
  - *HAProxy* – простейший сервер, собирающий статистику HAProxy и экспортирующий ее в Prometheus через регулярные интервалы времени посредством HTTP/JSON;
  - *Memcached* – сервер, экспортирующий статистику из сервера Memcached в Prometheus.

  При желании можно создать свой компонент экспортера для стороннего приложения. Кроме того, существует множество готовых экспортеров, полный список которых можно найти по адресу: https://github.com/prometheus/docs/blob/master/content/docs/instrumenting/exporters.m;
- **Alertmanager.** Обрабатывает предупреждения, рассылаемые сервером Prometheus и другими приложениями, согласно настройкам. В процессе обработки удаляет повторяющиеся предупреждения, выполняет группировку и маршрутизацию заданным в настройках потребителям (например, приложениям рассылки по электронной почте или на мобильные устройства).

Зная перечень компонентов и их функции, легко понять, как можно использовать Prometheus для мониторинга контейнеров. Вот короткий пример:

- запустить Prometheus и компоненты, перечисленные выше;
- добавить узел с контейнером экспортера для экспортирования статистики из контейнеров и контейнера cAdvisor;

- настроить узел экспортера и контейнеры cAdvisor и Prometheus как цели для мониторинга (в данном случае Prometheus будет выполнять мониторинг самого себя);
- установить и настроить Grafana;
- посмотреть получаемую статистику;
- интегрировать с Alertmanager для настройки предупреждений.

Шаг 1: запуск Prometheus

Первый шаг – запуск сервера Prometheus. Запустим этот сервер как контейнер Docker. Для сбора статистики мы настроим этот контейнер как цель для Prometheus, чтобы сервер мог осуществлять и мониторинг самого себя.

Начнем с создания файла компоновщика Docker, *docker-compose.yml*, который определяет параметры запуска Prometheus в контейнере:

```
version: '2'

networks:
- pk_network:
  driver:bridge

volumes:
prometheus_data: {}

services:
prometheus:
image: prom/prometheus
container_name: pk_prometheus
volumes:
- ./prometheus/:/etc/prometheus/
- prometheus_data:/prometheus
command:
- '-config.file=/etc/prometheus/prometheus.yml'
- '-storage.local.path=/prometheus'
- '-storage.local.memory-chunks=100000'
restart: unless-stopped
expose:
- 9090
ports:
- 9090:9090
networks:
- pk_network
labels:
  org.label-schema.group: "monitoring for PK containers"
```

Здесь мы извлекаем образ Prometheus и запускаем его как `pk_prometheus`. Мы также создаем сеть с драйвером bridge, `pk_network`, куда будут добавляться контейнеры. Затем отображаем конфигурационный файл, *prometheus.yml*, определяющий настройки для сбора информации, а также отображаем и экспортируем сетевой порт.

Ниже приводится содержимое файла *prometheus.yml*:

```
global:
scrape_interval: 20s
evaluation_interval: 20s

# Добавить метку для графического представления
external_labels: monitor: 'Docker-pk-monitor'

# Конечные точки для сбора информации
- job_name: 'pk_prometheus'
scrape_interval: 25s
static_configs:
- targets: ['localhost:9090']
```

Код достаточно прост и понятен. Здесь мы настроили интервалы сбора и обработки данных. Интервал сбора `scrape_interval` определяет, как часто будет опрашиваться целевой контейнер, а интервал обработки `evaluation_interval` задает частоту выполнения правил обработки. Обратите внимание, что мы добавили цель с контейнером Prometheus, чтобы сервер осуществлял мониторинг самого себя.

Теперь запустим Prometheus командой `docker-compose`, как показано на рис. 10.5.

```
docker-compose up -d
```

```
[ANUJSIN-M-T2H9: dockprom anujsin$
ANUJSIN-M-T2H9: dockprom anujsin$ docker-compose up -d
WARNING: The Docker Engine you're using is running in swarm mode.

Compose dose not use swarm mode to deploy services to multiple nodes in a swarm.

to deploy your application across the swarm, use `docker stack deploy`.

Pulling prometheus (prom/prometheus:lastest)...
lastest:pulling from prom/prometheus
4b0bc1c4050b : pull complete
a3ed95caeb02: pull complete
d6ab6c75ce17 : pull complete
96eeb64debe6: pull complete
1e7ee99aa461 : pull complete
8d3b35efed41 : pull complete
be179630d433 : pull complete
63e7097 0c133 : pull complete
83449160ff0d : pull complete
Digest: sha256:4f6d3a525f030e598016be765283c6455c3c830997a5c916b27a5d727be718e1
Status: Downloaded newer image for prom/prometheus: latest
Creating prometheus ...
Creating prometheus ... done
ANUJSIN-M-T2H9:dockprom anujsin$
```

Рис. 10.5 ❖ Создание контейнера Prometheus

Чтобы убедиться, что Prometheus запущен и работает, выполним команду `docker ps`, как показано на рис. 10.6.

```
ANUJSIN-M-T2H9:dockprom anujsin$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
 PORTS                    NAMES
48d1efc55881        prom/prometheus     "/bin/promethus -..."    54 seconds ago      up 53 seconds
 0.0.0.0:9090->9090/tcp    promethus
ANUJSIN-M-T2H9:dockprom anujsin$
```

Рис. 10.6 ❖ Prometheus запущен и работает

Пока все выглядит неплохо. Чтобы увидеть пользовательский интерфейс Prometheus, откройте в браузере страницу *http://localhost:9090/*. На рис. 10.7 показано, как она должна выглядеть.

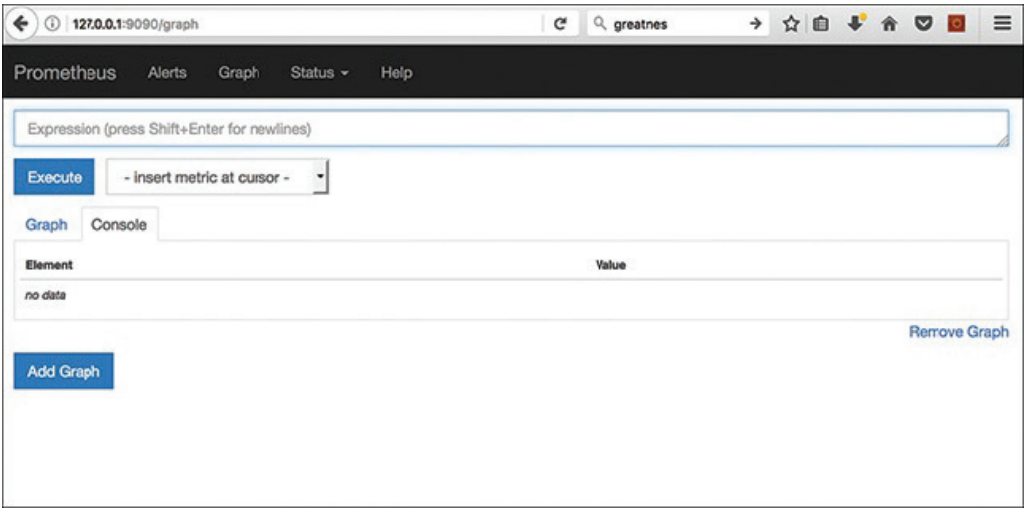


Рис. 10.7 ❖ Пользовательский интерфейс Prometheus

Шаг 2: добавление узла экспортера и cAdvisor

Теперь добавим в файл компоновщика другие компоненты и настроим цели. Сначала добавим в файл компоновщика узел экспортера и cAdvisor, чтобы запустить их в контейнерах. Обратите внимание, что эти контейнеры тоже будут использоваться как цели для сервера Prometheus.

```
nodeexporter:
image: prom/node-exporter
container_name: pk_nodeexporter
restart: unless-stopped
expose:
- 9100
networks:
- pk_network
labels:
  org.label-schema.group: "monitoring for PK containers"
```

```
cadvisor:
image: google/cadvisor:v0.26.1
container_name: pk_cadvisor
volumes:
- /:/rootfs:ro
- /var/run:/var/run:rw
- /sys:/sys:ro
- /var/lib/docker/:/var/lib/docker:ro
restart: unless-stopped
expose:
- 8080
networks:
- pk_network
labels:
  org.label-schema.group: "monitoring for PK containers"
```

Пока ничего сложного. Мы развернули контейнеры экспортера и cAdvisor и экспортировали порты в ту же сеть.

Шаг 3: добавление целей

На следующем шаге мы должны добавить узел экспортера и cAdvisor в список целей Prometheus. Добавим их в имеющийся файл *prometheus.yml*:

```
scrape_configs:
- job_name: 'pk_nodeexporter'
scrape_interval: 15s
static_configs:
- targets: ['nodeexporter:9100']

- job_name: 'pk_cadvisor'
scrape_interval: 20s
static_configs:
- targets: ['cadvisor:8080']
```

Запустим файл компоновщика еще раз и убедимся, что новые контейнеры запустились:

```
docker-compose up -d
```

Как показано на рис. 10.8, пока все идет замечательно.

```
Status: Downloaded newer image for google/cadvisor:v0.26.1
Creating prometheus ...
Creating cadvisor ...
Creating nodeexporter ...
Creating prometheus
Creating nodeexporter
Creating cadvisor ... done
```

Рис. 10.8 ❖ Результат выполнения команды `docker compose`

Шаг 4: настройка пользовательского интерфейса Grafana

Чтобы настроить Grafana для просмотра статистики, вернемся к файлу компоновщика Docker и добавим туда настройки для Grafana:

```
...
volumes:
prometheus_data: {}
grafana_data: {}

...
grafana:
image: grafana/grafana
container_name: grafana
volumes:
- grafana_data:/var/lib/grafana
env_file:
- user.config
restart: unless-stopped
expose:
- 3000
ports:
- 3000:3000
networks:
- pk_network
labels:
  org.label-schema.group: "monitoring for PK containers"
```

Теперь определим пользователя с привилегиями администратора для Grafana. Добавим настройки в файл *user.config*, как было указано прежде в `env_file`, и разместим его в одном каталоге с файлом компоновщика Docker:

```
GF_ SECURITY_ADMIN_USER=admin
GF_ SECURITY_ADMIN_PASSWORD=admin
GF_ USERS_ALLOW_SIGN_UP=false
```

Теперь запустим Grafana и проверим:

```
docker-compose up -d
```

Как показано на рис. 10.9, все наши контейнеры благополучно запустились.

```
Creating pk_prometheus ...
Creating pk_nodeexporter ...
Creating pk_alertmanager ...
Creating pk_cadvisor ...
Creating pk_grafana ...
Creating pk_nodeexporter
Creating pk_prometheus
Creating pk_alertmanager
Creating pk_cadvisor
Creating pk_prometheus ... done
```

Рис. 10.9 ❖ Все контейнеры благополучно запустились

Для проверки состояния контейнеров воспользуемся командой `docker ps`, как показано на рис. 10.10.

```
ANUJSIN-M-T2H9:dockprom anujsin$ docker ps
CONTAINER ID        IMAGE                    COMMAND                  CREATED             STATUS
    PORTS                    NAMES
1c98a5683541        grafana/grafana          "/run.sh"                About a minute ago  Up 58 seconds
    0.0.0.0:3000->3000/tcp   pk_grafana
b6935f85ce88        google/cadvisor:v0.26.1  "/usr/bin/cadvisor..."   About a minute ago  Up 58 seconds
    8080/tcp                 pk_cadvisor
2e6535fda4ef        prom/alertmanager        "/bin/alertmanager..."   About a minute ago  Up 58 seconds
    0.0.0.0:9093->9093/tcp   pk_alertmanager
dba0e0aa1ce5        prom/prometheus          "/bin/prometheus -..."   About a minute ago  Up 57 seconds
    0.0.0.0:9000->9090/tcp   pk_prometheus
99703482e361        prom/node-exporter       "/bin/node_exporter"     About a minute ago  Up 58 seconds
    9100/tcp                 pk_nodeexporter
ANUJSIN-M-T2H9:dockprom anujsin$
```

Рис. 10.10 ❖ Проверка состояния контейнеров командой `docker ps`

Проверим наши приложения. Откройте страницу *http://localhost:9090/* (см. рис. 10.11). Перейдите по адресу: *http://localhost:3000/*, чтобы увидеть интерфейс Grafana (см. рис. 10.12).

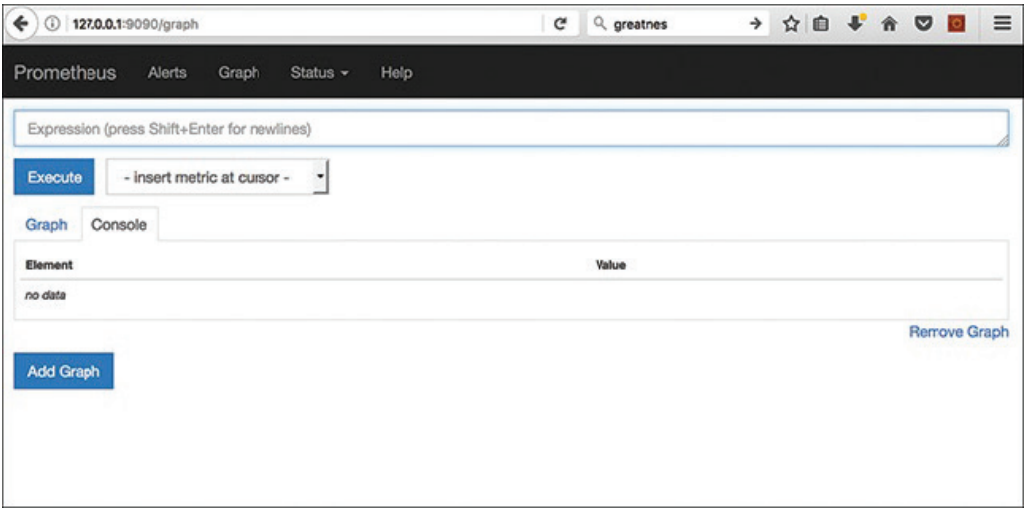


Рис. 10.11 ❖ Проверка Prometheus

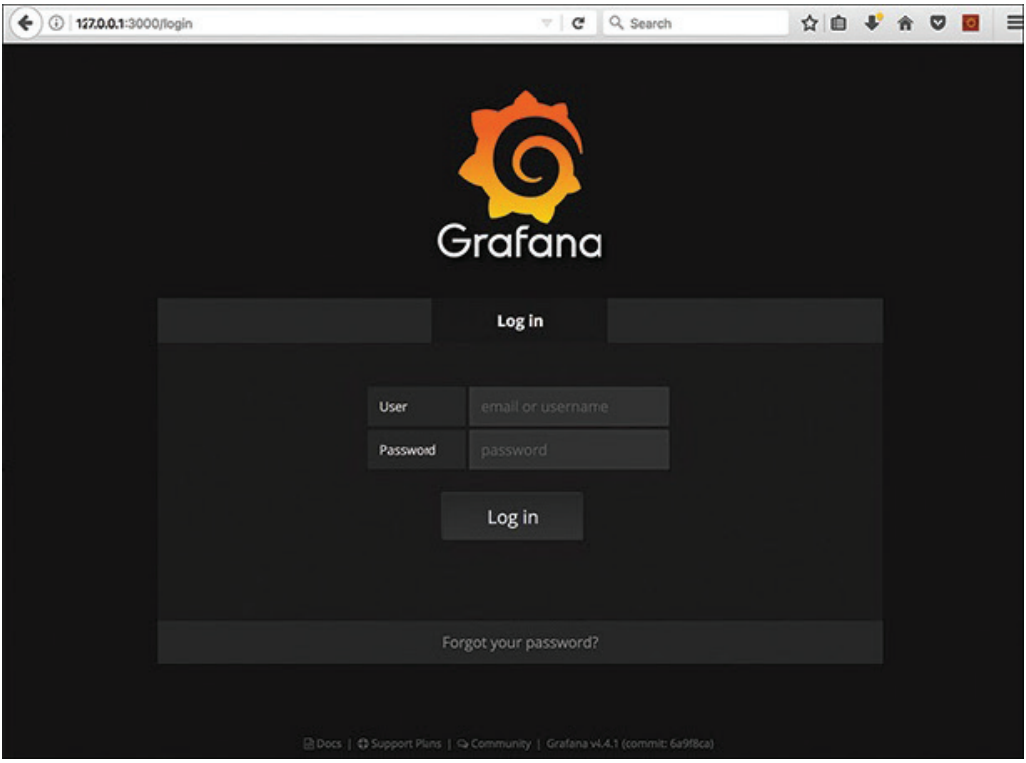


Рис. 10.12 ❖ Проверка Grafana

Как видите, приложение запущено и работает.

Настройка Grafana

Для визуализации данных нужно настроить Grafana. Для начала выполните вход с именем пользователя и паролем, указанными в конфигурационном файле Grafana (на данный момент `admin/admin`).

Затем добавьте источники данных, как показано на рис. 10.13.

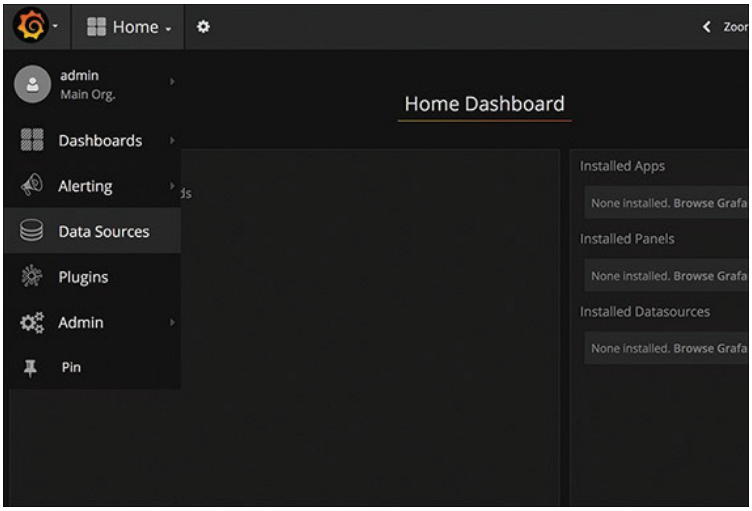


Рис. 10.13 ❖ Добавление источников данных для Grafana

Заполните поля информацией об источнике данных, такой как тип источника и учетные данные, как показано на рис. 10.14:

- **Name**: `Prometheus`;
- **Type**: `Prometheus`;
- **URL**: `http://prometheus:9090`;
- **Access**: `proxy`.

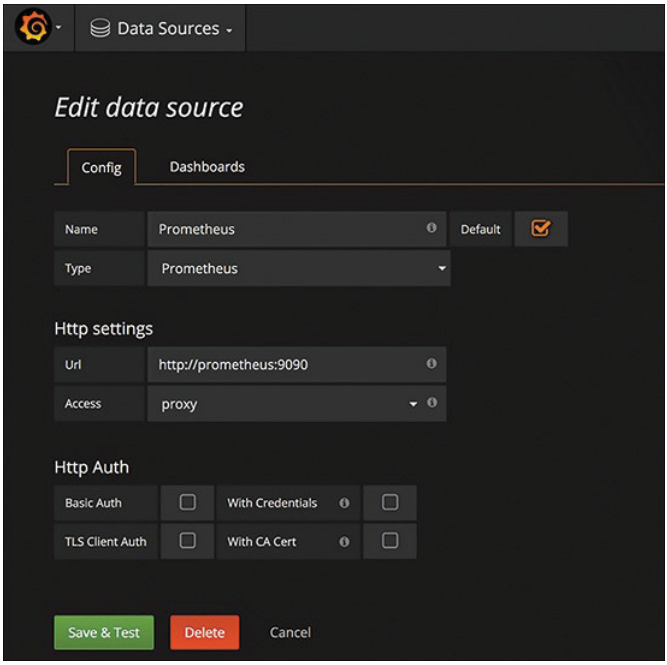


Рис. 10.14 ❖ Настройки источника данных

Щелкните на кнопке **Save & Test** (Сохранить и проверить), после чего должно появиться сообщение об успехе. Теперь Grafana и Prometheus связаны друг с другом.

Шаг 5: просмотр статистики

Все необходимые настройки выполнены, и теперь можно посмотреть статистику, собранную сервером Prometheus из трех его целей: cAdvisor, узла экспортера и самого сервера Prometheus.

Запустите пользовательский интерфейс Prometheus, открыв в браузере страницу *http://localhost:9090*. Щелкните на раскрывающемся списке рядом с кнопкой **Execute** (Выполнить) и выберите запрос. Затем щелкните на кнопке **Execute** (Выполнить), как показано на рис. 10.15.

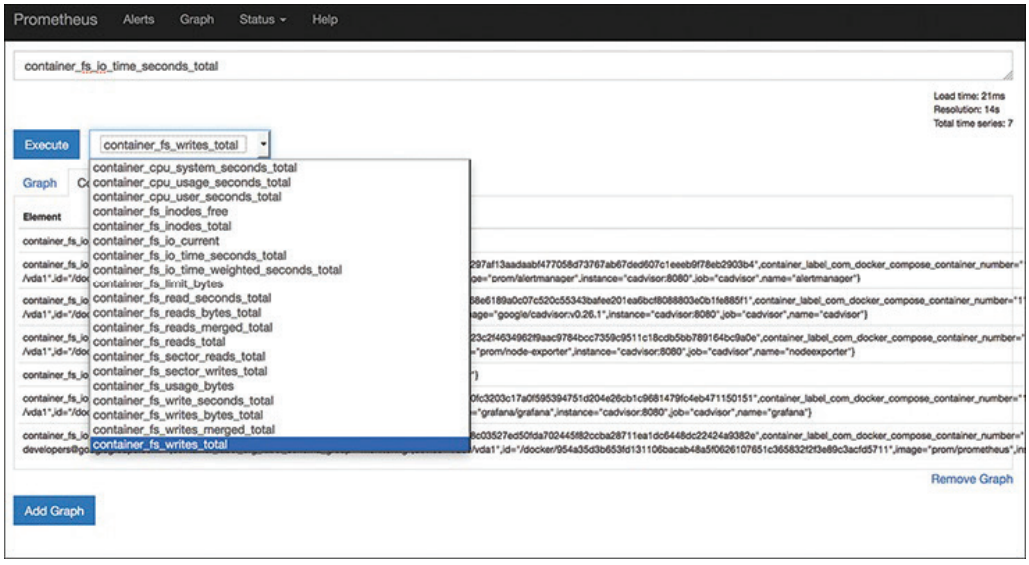


Рис. 10.15 ❖ Как просмотреть собранную статистику

В примере на рис. 10.16 был выбран запрос `container_cpu_system_seconds_total`, запрашивающий затраты времени CPU всеми контейнерами.

Информация ценная и важная, но ее отображение оставляет желать лучшего. Давайте добавим эстетики, импортировав статистику из Prometheus в Grafana. Запустите пользовательский интерфейс Grafana, открыв в браузере страницу *http://localhost:3000*. Выполните вход со своим именем пользователя и паролем (на данный момент `admin/admin`). Щелкните на раскрывающемся списке вверху и выберите пункт **Data Sources** (Источники данных). Щелкните на вкладке **Dashboards** (Дашборды), как показано на рис. 10.17.

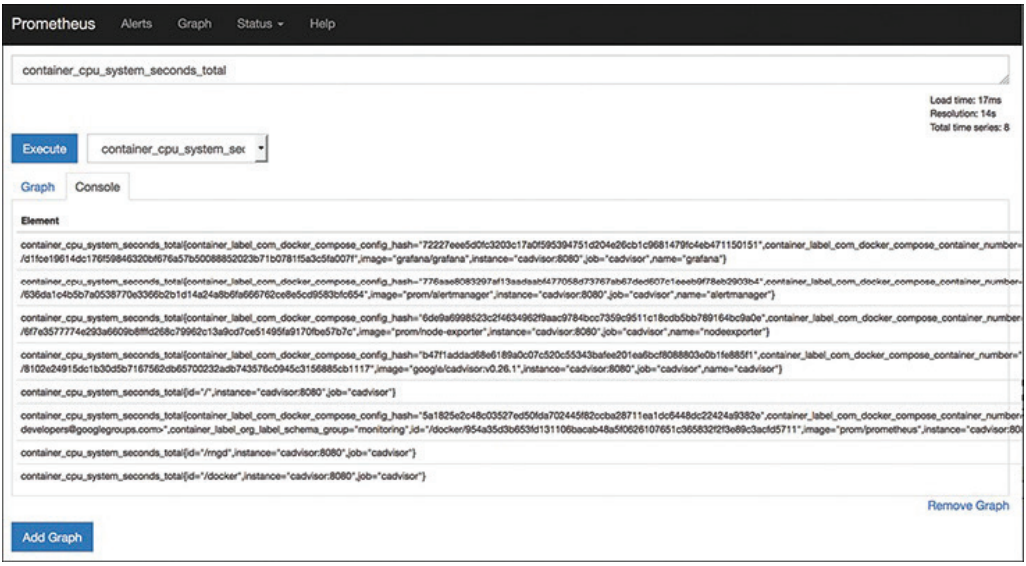


Рис. 10.16 ❖ Полученные результаты:
общее время CPU в секундах, затраченное контейнерами

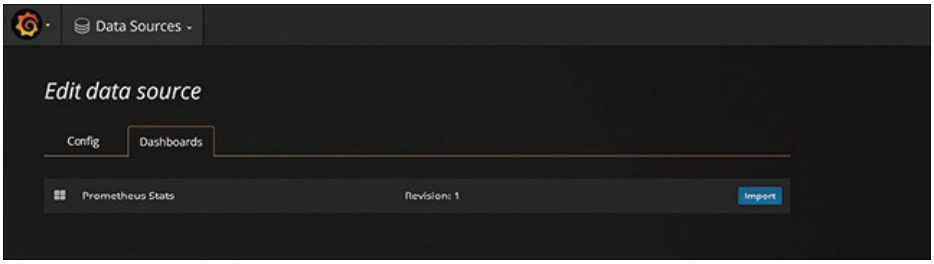


Рис. 10.17 ❖ Правка источника данных

В таблице на этой вкладке вы увидите строку **Prometheus Stats**, которая соответствует настройкам, выполненным нами выше (см. рис. 10.14). Щелкните на кнопке **Import** (Импортировать) справа в этой строке. В результате из базы данных Prometheus будут импортированы все статистики и предупреждения. Этот шаг требуется выполнить только один раз, впоследствии новые данные будут автоматически добавляться при каждом обновлении Grafana.

Чтобы увидеть только что импортированную статистику, щелкните на строке **Prometheus Stats**. После этого вы должны увидеть новые, более привлекательные дашборды, как показано на рис. 10.18.

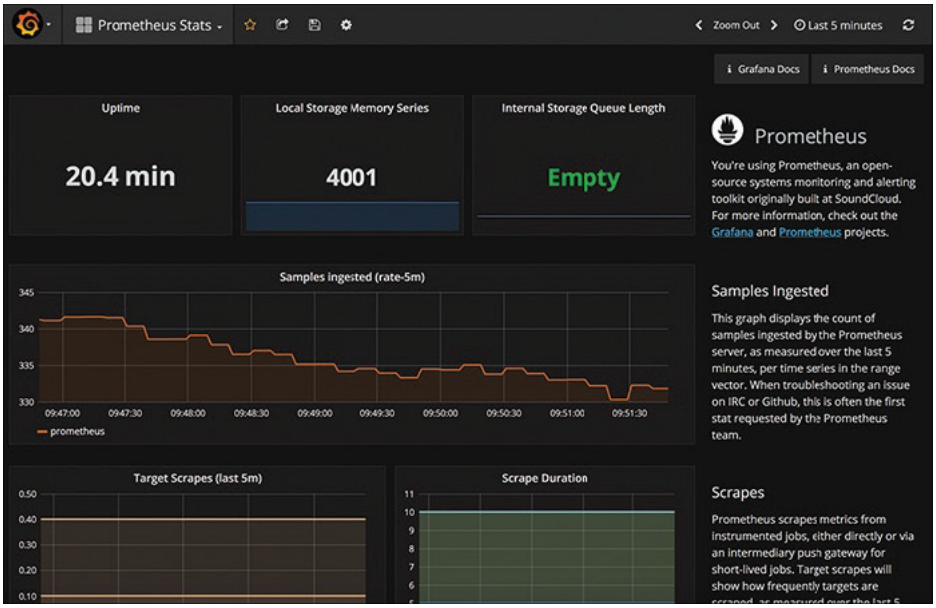


Рис. 10.18 ❖ Несколько новых привлекательных дашбордов

Смотрится здорово! И все это благодаря Grafana!

Теперь сделаем еще шаг вперед и создадим свой простенький дашборд, отображающий накопленную нагрузку контейнеров на CPU. Запустите пользовательский интерфейс Grafana, щелкните на значке с изображением спирали в левом верхнем углу и в открывшемся меню выберите **Dashboards** (Дашборды) и **New** (Создать), как показано на рис. 10.19.

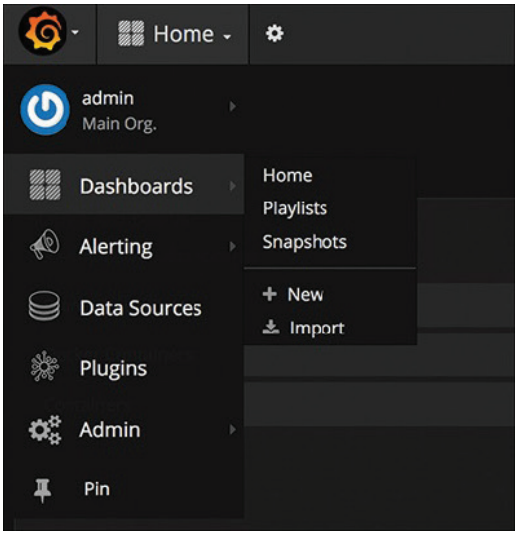


Рис. 10.19 ❖ Создание простого дашборда для отображения накопленной нагрузки контейнеров на CPU

Щелкните на **Single Stat** (Простая статистика) и добавьте следующие настройки:

```
sum(rate(container_cpu_user_seconds_total{image!=""}[1m])) /
count(node_cpu{mode="system"}) * 100
```

Запрос извлечет информацию об использованном времени CPU на данный момент, как показано на рис. 10.20. Этот дашборд будет отображать информацию в масштабе реального времени.

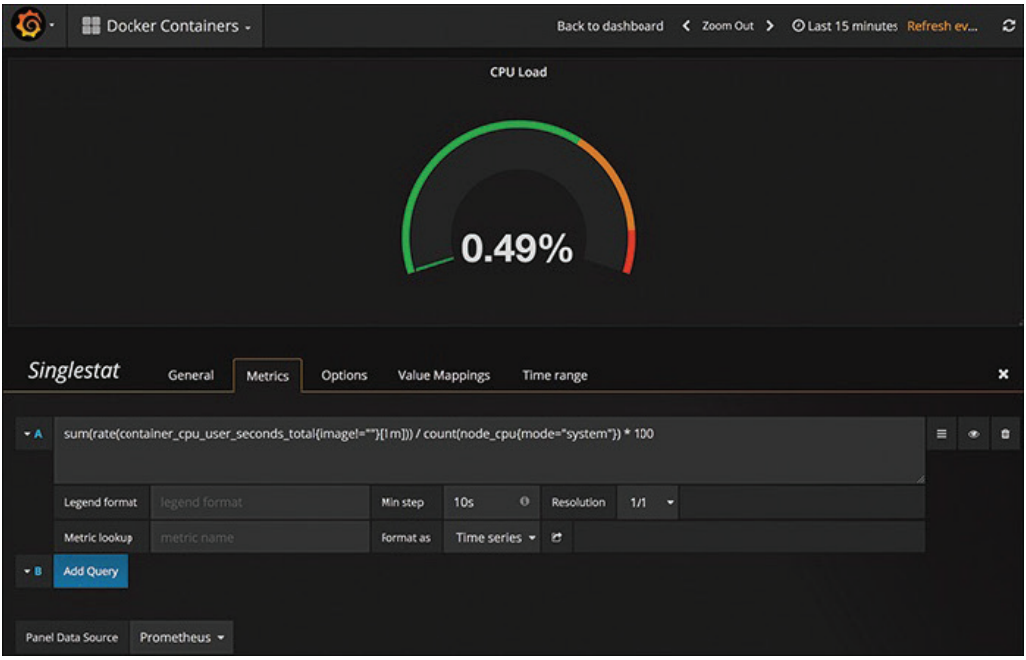


Рис. 10.20 ❖ Информация об использованном времени CPU

Аналогично можно создать дашборды, отображающие потребление памяти и системных ресурсов, как показано на рис. 10.21 и 10.22.

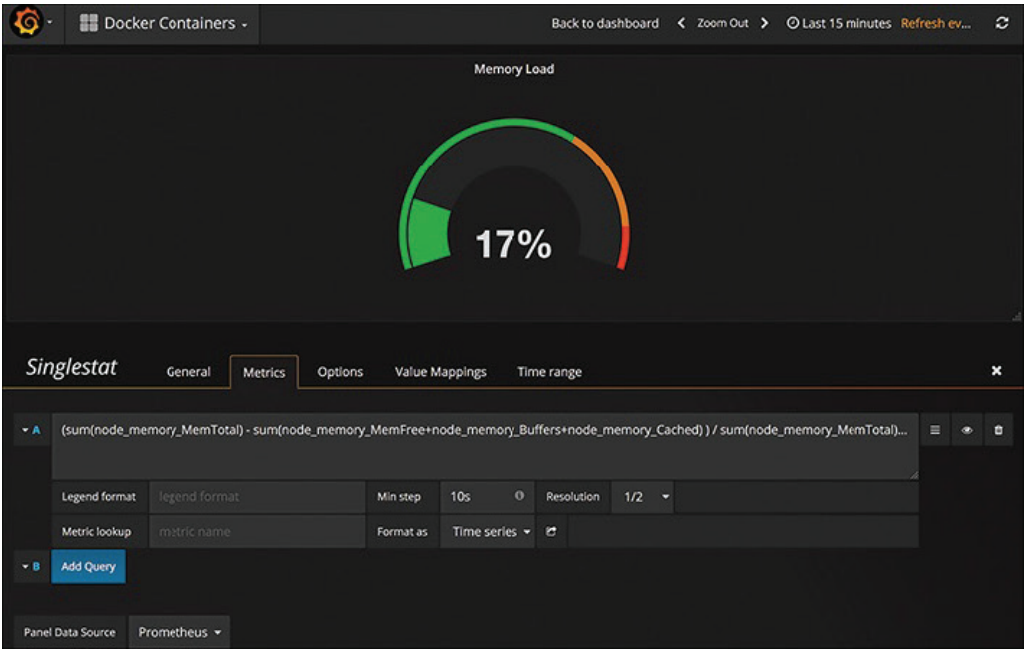


Рис. 10.21 ❖ Потребление памяти

Рис. 10.22 ❖ Потребление системных ресурсов

Шаг 6: интеграция Alertmanager

В заключение интегрируем в эту конфигурацию Alertmanager. В Alertmanager можно настроить рассылку предупреждений, основываясь на информации, собранной сервером Prometheus.

Давайте сделаем это:

1) откройте файл компоновщика Docker и добавьте следующие строки:

```
alertmanager:
image: prom/alertmanager
container_name: alertmanager_pk
volumes:
- ./alertmanager/:/etc/alertmanager/
command:
- '-config.file=/etc/alertmanager/config.yml'
- '-storage.path=/alertmanager'
restart: unless-stopped
expose:
- 9093
ports:
- 9093:9093
networks:
- pk_network
labels:
  org.label-schema.group: "monitoring for PK containers"
```

2) здесь же, в файле компоновщика Docker, добавьте Alertmanager в контейнер службы Prometheus:

```
prometheus:
image: prom/prometheus
container_name: Prometheus_pk
volumes:
- ./prometheus/:/etc/prometheus/
- prometheus_data:/prometheus
command:
- '-config.file=/etc/prometheus/prometheus.yml'
- '-storage.local.path=/prometheus'
- '-alertmanager.url=http://alertmanager:9093'
- '-storage.local.memory-chunks=100000'
restart: unless-stopped
expose:
- 9090
ports:
- 9090:9090
networks:
- pk_network
labels:
  org.label-schema.group: "monitoring for PK containers"
```

3) создайте файл *container.rules* и добавьте в него следующее правило:

```
ALERT tomcat_down
  IF absent(container_memory_usage_bytes{name="tomcat"})
  FOR 10s
  LABELS { severity = "critical" }
  ANNOTATIONS {
  summary= "tomcat down",
  description= "tomcat container is down for more than 10 seconds."
  }
```

Это правило проверяет состояние контейнера Tomcat и генерирует предупреждение, если он не запущен. Проверка выполняется запросом объема памяти, потребляемой контейнером Tomcat. Если эта статистика отсутствует, значит, контейнер не запущен, требуется сгенерировать предупреждение;

4) добавьте ссылку на правило в файл *prometheus.yml*:

```
# Загружать и выполнять правила из этого файла
# каждые 'evaluation_interval' секунд.
rule_files:
- "containers.rules"
```

5) запустите файл компоновщика Docker еще раз:

```
docker-compose up -d
```

Запустите пользовательский интерфейс Prometheus, открыв в браузере страницу *http://localhost:9090*. Щелкните на меню **Alerts** (Предупреждения) вверху, после чего должны появиться активные предупреждения, как показано на рис. 10.23.

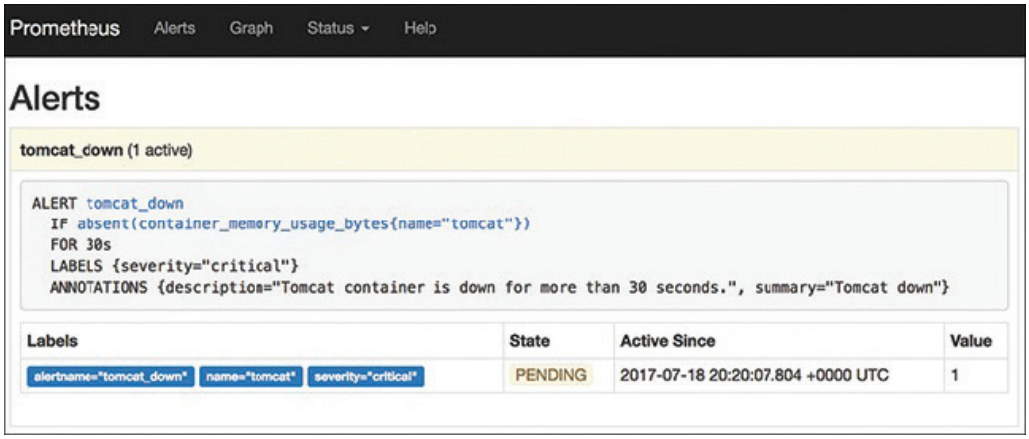


Рис. 10.23 ❖ Вывод активных предупреждений

Вы можете продолжить совершенствование этой конфигурации, используя инструменты для работы с предупреждениями по своему выбору. Более подробную информацию ищите на сайте Prometheus: https://prometheus.io/.

Как уже говорилось, мониторинг – очень важный аспект, и его организация должна быть в числе первых задач, решаемых при переходе к использованию контейнеров. Это совершенно новая область, страдающая от некоторых проблем из-за недостатков, описанных выше в этой главе. Не унывайте, потому что каждый день на рынке появляются новые решения. Поэтому следите за новостями и продолжайте учиться!

Часть III

ПРАКТИЧЕСКИЙ ПРОЕКТ – ПРИМЕНЕНИЕ ТЕОРИИ НА ПРАКТИКЕ

Глава 11

Практический пример: монолитное приложение Helpdesk

В этой главе мы создадим традиционное веб-приложение для службы поддержки, следуя общепринятым стандартам. Кроме того, мы создадим его без использования идей, которые рассматривали до сих пор, т. е. напишем монолитное приложение. Главная цель этого примера – получить практический опыт. Закончив создание приложения, мы рассмотрим некоторые сложности, порождаемые монолитной архитектурой и возникающие, например, при развертывании приложения, управлении обновлениями и масштабировании. Разобравшись со сложностями, характерными для монолитной архитектуры, в следующих двух главах мы посмотрим, как от них избавиться, использовав архитектуру микросервисов и контейнеры Docker.

Обзор приложения Helpdesk

В современном цифровом мире большинство компаний предлагает своим клиентам модель поддержки, основанной на самообслуживании, которая реализуется как мобильное и/или веб-приложение. Высокая доступность, производительность и мощные средства поиска – вот ключевые аспекты и важнейшие характеристики, помогающие быстро решать проблемы.

Это приложение обеспечивает необходимую поддержку для управления проблемами клиентов и их разрешения. Имейте в виду, что приложение, представленное здесь, сильно упрощено, чтобы облегчить объяснение понятий, архитектуры и сложностей монолитных приложений.

Предположим, что в реальном мире это приложение будет обеспечивать поддержку клиентов компании, продающей мобильные телефоны. Оно будет обеспечивать возможности, перечисленные ниже:

- **управление учетными записями.** Каждому пользователю будет предоставляться возможность создать / изменить / удалить свою учетную запись. Аутентификация будет осуществляться по имени пользователя и паролю, хранящимся в локальной базе данных;

- **создание и управление претензиями.** Пользователи будут иметь возможность посылать новые и просматривать имеющиеся заявки с претензиями;
- **управление каталогом продуктов (только для администраторов)** – хранение каталога продуктов и управление им;
- **консультации** – дают возможность проконсультироваться у специалиста в указанное время и в указанном магазине;
- **поиск** – поиск проблем и методов их решения, а также поиск в каталоге продуктов;
- **доска объявлений.** Доска объявлений для клиентов, используя которую, они смогут помогать друг другу.

При создании приложения будут использоваться следующие технологии:

- **пользовательский интерфейс** (HTML, JavaScript и JQuery);
- **промежуточный уровень** (Java 7, Spring 3.x, Jersey 1.8 и Hibernate);
- **база данных** (MySQL 5.x).

Более подробно рабочий процесс описан в приложении А. Весь код и ресурсы доступны в репозитории GitHub по адресу: https://github.com/kocherMSD/Helpdesk_Monolithic.git.

Загрузить код на локальную машину можно командой `git clone`. Мы будем использовать данный код на протяжении всей этой главы.

Архитектура приложения

Теперь, получив представление о функциях, предлагаемых приложением, обратимся к техническим деталям. На рис. 11.1 показана компонентная архитектура приложения.

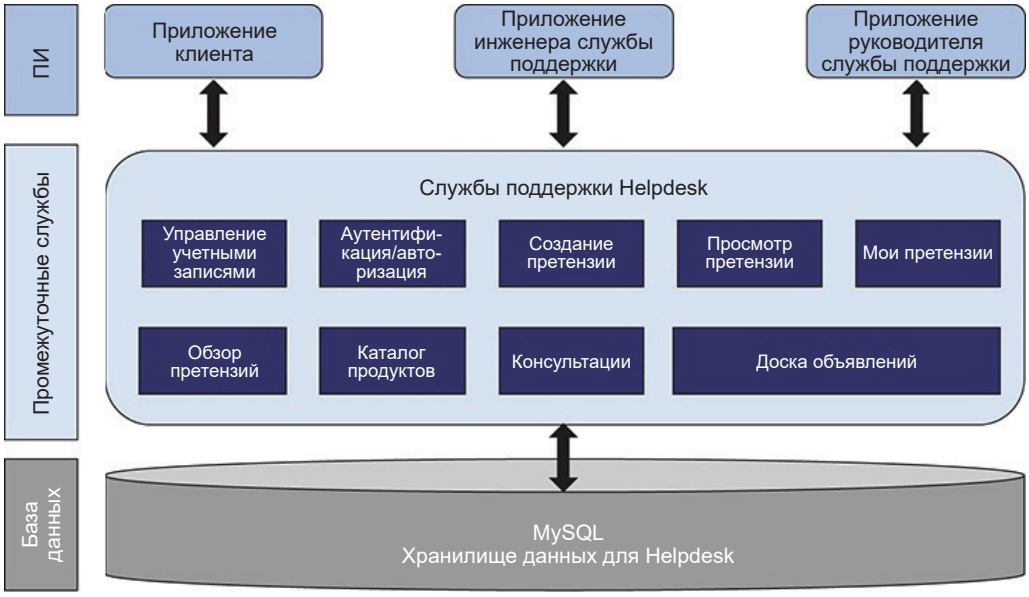


Рис. 11.1 ❖ Базовая архитектура и компоненты приложения Helpdesk

Как видите, приложение имеет трехуровневую архитектуру, включающую базу данных, службы с бизнес-логикой и пользовательский интерфейс (ПИ). Теперь рассмотрим в общих чертах список служб, составляющих приложение. Детали реализации вы сможете найти в исходном коде, хранящемся в репозитории GitHub.

Аутентификация, интерцептор и авторизация

Как можно судить по названию, этот модуль предлагает услуги по аутентификации и авторизации пользователей и определению прав доступа к той или иной информации в зависимости от их ролей. Для простоты мы реализовали аутентификацию и роли просто как хранящиеся в базе данных имя пользователя и пароль и имя пользователя и роль соответственно. Проверка аутентификации всех запросов выполняется с помощью интерцептора Spring. Роль и идентификатор пользователя сохраняются в сеансе и извлекаются оттуда, когда это необходимо.

Далее демонстрируется псевдокод аутентификации.

Аутентификация

Эта служба осуществляет аутентификацию пользователя, принимая его имя и пароль из текстовых полей ввода на странице входа. Имя пользователя и пароль сопоставляются с записью в базе данных.

- контекст: `authenticate`;
- метод: `POST`;
- потребляет: `application/xml`, `application/json`;
- производит: `application/json`;
- вход: `HttpHeaders`, `request`;
- выход: код ответа (например, успех или неудача).

Далее следует псевдокод службы аутентификации:

```
@Override
@POST
@Consumes({"application/xml", "application/json"})
@Produces({"application/json"})
@Path("/authenticate/")
public AuthenticationResponse authenticate(
    @Context HttpHeaders headers,
    AuthenticationRequest request)
    //TODO: Реализовать
}
```

Интерцептор

Интерцептор перехватывает все входящие запросы, поступающие в сервер приложения, благодаря использованию шаблона `./*` и вызывает функцию `prehandle`. Ниже приводится псевдокод на XML и Java, характерный при использовании фреймворка Spring:

```
<interceptors>
  <interceptor>
    <mapping path="/*"/>
    <beans:bean>
```

```
      class="org.spring.controller.AuthenticationInterceptor"
    <beans:bean/>
  </interceptor>
</interceptors>
```

```
@Override
public boolean preHandle(
    HttpServletRequest request,
    HttpServletResponse response,
    Object handler) throws Exception {
    //TODO: Реализовать
}
```

Авторизация

Наше приложение поддерживает несколько ролей, и, когда пользователь выполняет вход, его роль сохраняется в сеансе HTTP логикой интерцептора. Следующий код используется в контроллере авторизации для получения сеанса HTTP.

```
LoginForm userData = (LoginForm)
context.getSession().getAttribute("LOGGEDIN_USER");
```

Следующий код реализует интерфейс авторизации JavaServer Pages (JSP):

```
<%
  LoginForm loginform=(LoginForm)session.getAttribute ("LOGGEDIN_USER");
  String user=loginform.getUsername();
  if(session.getAttribute("ACCESS_LEVEL").equals("4"))
%>
```

Управление учетными записями

Этот компонент предлагает услуги, связанные с управлением учетными записями пользователей, сведениями о договорах или правах сторон, сведениями о покупках, такими, например, как информация о продукте, серийный номер и т. д. Например, клиент может приобрести один или несколько мобильных телефонов с гарантией и техническим обслуживанием. Эти сведения будут доступны через API для проверки прав и предоставления гарантийных услуг. Вот полный список предоставляемых услуг:

- `getAccount` – возвращает информацию о подключенном пользователе;
- `addAccount` – подключает нового пользователя;
- `updateAccount` – обновляет информацию о подключенном пользователе;
- `deleteAccount` – удаляет пользователя из системы.

Ниже приводится сигнатура класса компонента:

```
@Component
@Path("/AccountService")
public class AccountServiceImpl implements AccountService {
```

getAccount

Эта служба возвращает информацию об учетной записи зарегистрированного пользователя, если она доступна в системе. Данная информация извлекается из базы данных и возвращается в формате JSON:

- контекст – AccountService/getAccount/{customerId};
- метод – GET;
- потребляет – application/xml, application/json;
- производит – application/json;
- вход – HttpHeaders, customerId;
- выход – данные в формате JSON с учетной записью пользователя, устройством и сведениями о порядке обслуживания.

Далее показан псевдокод с реализацией getAccount:

```
@Override
@GET
@Consumes({"application/xml", "application/json"})
@Produces({"application/json"})
@Path("/getAccount/{customerId}")
public AccountViewResponse getAccount(
       @Context HttpHeaders headers,
       @PathParam("customerId")String customerId)
       throws ServiceInvocationException {
    //TODO: реализовать объект DAO
}
```

addAccount

Добавляет учетную запись для нового клиента, сохраняя ее в базе данных. Входные данные извлекаются из текстовых полей формы, преобразуются в формат JSON и сохраняются в соответствующих таблицах базы данных:

- контекст – AccountService/addAccount;
- метод – POST;
- потребляет – application/xml, application/json;
- производит – application/json;
- вход – HttpHeaders, данные в формате JSON с учетной записью пользователя, устройством и сведениями о порядке обслуживания;
- выход – код ответа (например, успех или неудача).

Далее показан псевдокод с реализацией addAccount:

```
@Override
@POST
@Consumes({"application/xml", "application/json"})
@Produces({"application/json"})
@Path("/addAccount/")
public AccountResponse addAccount(
       @Context HttpHeaders headers,
       AccountRequest req)
       throws ServiceInvocationException {
    //TODO: реализовать объект DAO
}
```

updateAccount

Обновляет информацию в учетной записи пользователя. Обновленная информация сохраняется в базе данных:

- контекст – AccountService/updateAccount;

- метод – POST;
- потребляет – application/xml, application/json;
- производит – application/json;
- вход – HttpHeaders, данные в формате JSON с учетной записью пользователя, устройством и сведениями о порядке обслуживания;
- выход – код ответа (например, успех или неудача).

Далее показан псевдокод с реализацией updateAccount:

```
@Override
@POST
@Consumes({"application/xml", "application/json"})
@Produces({"application/json"})
@Path("/updateAccount/")
public AccountResponse updateAccount(
       @Context HttpHeaders headers,
       AccountRequest req)
       throws ServiceInvocationException {
    //TODO: реализовать объект DAO
}
```

deleteAccount

Удаляет учетную запись пользователя. Если требуемая учетная запись существует, она удаляется из базы данных:

- контекст – AccountService/deleteAccount;
- метод – POST;
- потребляет – application/xml, application/json;
- производит – application/json;
- вход – HttpHeaders, данные в формате JSON с учетной записью пользователя, устройством и сведениями о порядке обслуживания;
- выход – код ответа (например, успех или неудача).

Далее показан псевдокод с реализацией deleteAccount:

```
@Override
@POST
@Consumes({"application/xml", "application/json"})
@Produces({"application/json"})
@Path("/deleteAccount/")
public AccountResponse deleteAccount(
       HttpHeaders headers,
       AccountRequest req)
       throws ServiceInvocationException {
    //TODO: реализовать объект DAO
}
```

Претензии

Эта группа служб дает зарегистрированному пользователю возможность создавать претензии и просматривать касающиеся приобретенного продукта ответы специалистов из отдела поддержки. Данный компонент включает следующие службы:

- `createTicket` – создает претензию;
- `viewTicket` – возвращает претензию для просмотра;
- `viewAllTicket` – возвращает все претензии для просмотра.

Ниже приводится определение класса компонента:

```
@Component
@Path("/TicketService")
public class HelpDeskTicketServiceImpl
    implements HelpDeskTicketService, ApplicationContextAware {
```

createTicket

Создает претензию от имени указанного пользователя. Входные данные извлекаются из текстовых полей формы на странице создания претензии и преобразуются в формат JSON. Далее запрос в формате JSON преобразуется в соответствии с моделью данных и сохраняется в базе данных с использованием Hibernate:

- контекст – `TicketService/createTicket`;
- метод – `POST`;
- потребляет – `application/xml`, `application/json`;
- производит –– `application/json`;
- вход – `HttpHeaders`, данные в формате JSON с претензией (например, номер договора, информация о проблеме, идентификатор пользователя);
- выход – номер созданной претензии, код ответа (например, успех или неудача).

Далее показан псевдокод с реализацией `createTicket`:

```
@Override
@POST
@Consumes({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
@Produces({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
@Path("/createTicket/")
public TicketResponse createHdTicket(
       @Context HttpHeaders headers,
       TicketRequest ticketRequest)
       throws ServiceInvocationException{
    //TODO: реализовать объект DAO
}
```

viewTicket

Возвращает информацию о претензии по номеру и роли пользователя. Извлекает информацию из базы данных, если клиент указал действительный номер претензии. Данные извлекаются с помощью Hibernate и возвращаются в формате JSON:

- контекст – `TicketServices/viewTicket/{userId}`;
- метод – `GET`;
- потребляет – `application/xml`, `application/json`;
- производит – `application/xml`, `application/json`;
- вход – `HttpHeaders`;
- выход – данные в формате JSON с претензией (например, номер договора, информация о проблеме, идентификатор пользователя).

Далее показан псевдокод с реализацией viewTicket:

```
@Override
@GET
@Consumes({"application/xml", "application/json"})
@Produces({"application/json"})
@Path("/viewTicket/{userId}/{ticketId}")
public ViewTicketResponse viewTicket(
       @Context HttpHeaders headers,
       @PathParam("userId")String userId,
       @PathParam("ticketId")String ticketId)
       throws ServiceInvocationException {
}
```

viewAllTicket

Возвращает все претензии, зарегистрированные в системе текущим пользователем. Извлекает информацию из базы данных с помощью Hibernate и возвращает ее в формате JSON:

- контекст – TicketServices/viewAllTicket;
- метод – GET;
- потребляет – application/xml, application/json;
- производит – application/xml, application/json;
- вход – HttpHeaders;
- выход – данные в формате JSON с претензиями (например, номера договоров, информация о проблемах, идентификатор пользователя).

Далее показан псевдокод с реализацией viewAllTicket:

```
@Override
@GET
@Consumes({"application/xml", "application/json"})
@Produces({"application/json"})
@Path("/viewAllTicket/")
public ViewAllTicketResponse viewAllTicket(
       @Context HttpHeaders headers)
       throws ServiceInvocationException {
    //TODO: реализовать объект DAO
}
```

Для пользователей с разными ролями предусмотрены разные варианты обслуживания:

- **мои претензии.** Сотруднику отдела поддержки возвращается список претензий, переданных ему для рассмотрения, а пользователю – список претензий, поданных им;
- **обзор претензий.** Руководителю возвращается полный список претензий.

Каталог продуктов

Компонент каталога продуктов позволяет администраторам управлять списком продуктов, предлагаемых компанией, а пользователям – просматривать каталог приобретенных ими продуктов, для которых они могут подать претензию. Этот компонент включает следующие службы:

- getCatalog – возвращает каталог продуктов;
- addCatalog – создает новый пункт в каталоге продуктов;
- updateCatalog – обновляет указанный пункт в каталоге продуктов;
- deleteCatalog – удаляет указанный пункт из каталога продуктов.

Ниже приводится определение класса компонента:

```
@Path("/CatalogService")
public class CatalogServiceImpl implements CatalogService {
```

getCatalog

Возвращает список продуктов, имеющихся в системе. Извлекает информацию из базы данных с помощью Hibernate и возвращает ее в формате JSON:

- контекст – CatalogService/getCatalog/{customerId};
- метод – GET;
- потребляет – application/xml, application/json;
- производит – application/json; ;
- вход– HttpHeaders, customerId (все вхождения заголовка должны быть заменены на HttpHeaders);
- выход – данные в формате JSON с информацией о продуктах, приобретенных пользователем.

Далее показан псевдокод с реализацией getCatalog:

```
@Override
@GET
@Consumes({"application/xml", "application/json"})
@Produces({"application/json"})
@Path("/getCatalog/{customerId}")
public ProductDetailsResponse getCatalog(
       @Context HttpHeaders headers,
       @PathParam("customerId") String customerId)
       throws ServiceInvocationException {
    //TODO: реализовать объект DAO
}
```

addCatalog

Добавляет новый продукт в каталог. Входные данные извлекаются из текстовых полей формы на странице создания продукта и преобразуются в формат JSON. Далее запрос в формате JSON преобразуется в соответствии с моделью данных и сохраняется в базе данных с использованием Hibernate:

- контекст – CatalogService/addCatalog;
- метод – POST;
- потребляет – application/xml, application/json;
- производит – application/json;
- вход – HttpHeaders, данные в формате JSON с информацией о продуктах, приобретенных пользователем;
- выход – код ответа (например, успех или неудача).

Далее показан псевдокод с реализацией addCatalog:

```
@Override
@POST
```

```
@Consumes({"application/xml", "application/json"})
@Produces({"application/json"})
@Path("/addCatalog/")
public CatalogResponse addCatalog(
       @Context HttpHeaders headers,
       CatalogRequest req)
       throws ServiceInvocationException {
    //TODO: реализовать объект DAO
}
```

updateCatalog

Обновляет пункт каталога с информацией о продукте, если он имеется в системе. Обновленная информация сохраняется в базе данных:

- контекст – `CatalogService/updateCatalog`;
- метод – `POST`;
- потребляет – `application/xml`, `application/json`;
- производит – `application/json`;
- вход – `HttpHeaders`, данные в формате JSON с информацией о продуктах, приобретенных пользователем;
- выход: код ответа (например, успех или неудача).

Далее показан псевдокод с реализацией `updateCatalog`:

```
@Override
@POST
@Consumes({"application/xml", "application/json"})
@Produces({"application/json"})
@Path("/updateCatalog/")
public CatalogResponse updateCatalog(
       HttpHeaders headers,
       CatalogRequest req)
       throws ServiceInvocationException {
    //TODO: реализовать объект DAO
}
```

deleteCatalog

Удаляет из каталога пункт с описанием указанного продукта. Удаление выполняется в базе данных с помощью Hibernate:

- контекст – `CatalogService/deleteCatalog`;
- метод – `POST`;
- потребляет – `application/xml`, `application/json`;
- производит – `application/json`;
- вход – `HttpHeaders`, данные в формате JSON с информацией о продуктах, приобретенных пользователем;
- выход – код ответа (например, успех или неудача).

Далее показан псевдокод с реализацией `deleteCatalog`:

```
@Override
@POST
@Consumes({"application/xml", "application/json"})
@Produces({"application/json"})
```

```
@Path("/deleteCatalog/")
public CatalogResponse deleteCatalog(
       HttpHeaders headers,
       CatalogRequest req)
       throws ServiceInvocationException {
    //TODO: реализовать объект DAO
}
```

Консультации

Консультации реализованы по аналогии с Apple Genius Bar[1]. Пользователи могут зарезервировать время для получения консультации у специалиста в указанное время и в указанном магазине. Этот компонент реализует следующие службы:

- `getAvailableTimeSlots` – возвращает все интервалы времени в указанную дату, когда специалисты свободны от консультаций;
- `getAvailableDates` – возвращает дни, когда есть хотя бы один свободный интервал времени;
- `saveAppointment` – резервирует время для консультации.

Ниже приводится определение класса компонента:

```
@Component
@Path("/AppointmentService")
public class AppointmentServiceImpl {
```

getAvailableTimeSlots

Возвращает все свободные интервалы времени в указанную дату, которая передается в формате JSON:

- контекст – `AppointmentService/getAvailableTimeSlots`;
- метод – `GET`;
- потребляет – `application/xml`, `application/json`;
- производит – `application/json`;
- вход – `HttpHeaders`, `TITLE`;
- выход – код ответа (например, успех или неудача).

Далее показан псевдокод с реализацией `getAvailableTimeSlots`:

```
@Override
@POST
@Consumes({"application/xml", "application/json"})
@Produces({"application/json"})
@Path("/getAvailableTimeSlots/")
public
AppointmentAvailableTimeSlotResponse getAvailableTimeSlots(
      @Context HttpHeaders headers,
      AppointmentAvailableTimeSlotRequest Request) {
    //TODO: Реализовать
}
```

[1] Genius Bar – специальная секция в сети розничных магазинов Apple Store, отведенная для бесплатных консультаций и технической поддержки. – *Прим. перев.*

getAvailableDates

Возвращает все дни, когда есть хотя бы один свободный интервал времени для консультаций:

- контекст – `AppointmentService/getAvailableDates`;
- метод – `POST`;
- потребляет – `application/xml`, `application/json`;
- производит – `application/json`;
- вход – `HttpHeaders`, `TITLE`;
- выход – код ответа (например, успех или неудача).

Далее показан псевдокод с реализацией `getAvailableDates`:

```
@Override
@POST
@Consumes({"application/xml", "application/json"})
@Produces({"application/json"})
@Path("/getAvailableDates/")
public
AppointmentAvailableDateResponse getUnAvailableDates(
      @Context HttpHeaders headers,
      AppointmentAvailableDateRequest request) {
    //TODO: Реализовать
}
```

saveAppointment

Резервирует время для консультации и сохраняет в базе данных:

- контекст – `AppointmentService/saveAppointment`;
- метод – `POST`;
- потребляет – `application/xml`, `application/json`;
- производит – `application/json`; ;
- вход – `HttpHeaders`, `TITLE`, запрос;
- выход – код ответа (например, успех или неудача).

Далее показан псевдокод с реализацией `saveAppointment`:

```
@Override
@POST
@Consumes({"application/xml", "application/json"})
@Produces({"application/json"})
@Path("/saveAppointment/")

    //TODO: Реализовать
}
```

Доска объявлений

Доска объявлений позволяет организовать сотрудничество между членами сообщества и экспертами. Этот компонент реализует следующие службы:

- `getMessage` – извлекает сообщение/объявление, доступное в системе;
- `getAllMessage` – извлекает все сообщения/объявления, опубликованные в указанный интервал времени;

- `createMessage` – сохраняет сообщение, объявление, вопрос или ответ, переданный пользователем.

Ниже приводится определение класса компонента:

```
@Component
@Path("/MessageService")
public class MessageServiceImpl implements MessageService {
```

getMessage

Извлекает объявления, сообщения, вопросы и ответы, доступные в системе, опираясь на вопрос, заданный пользователем:

- контекст – `MessageService/getMessage/{title}`;
- метод – `GET`;
- потребляет – `application/xml`, `application/json`;
- производит – `application/json`;
- вход – `HttpHeaders`, `TITLE`;
- выход – код ответа (например, успех или неудача).

Далее показан псевдокод с реализацией `getMessage`:

```
@Override
@GET
@Consumes({"application/xml", "application/json"})
@Produces({"application/json"})
@Path("/getMessage/{title}")
public MessageViewResponse getMessage(
       @Context HttpHeaders headers,
       @PathParam("title")String title)
       throws ServiceInvocationException {
    //TODO: реализовать объект DAO
}
```

getAllMessage

Извлекает все объявления, сообщения или вопросы за указанный интервал времени и возвращает их в формате JSON:

- контекст – `MessageService/getAllMessage`;
- метод – `GET`;
- потребляет – `application/xml`, `application/json`;
- производит – `application/json`;
- вход – `HttpHeaders`;
- выход – код ответа (например, успех или неудача).

Далее показан псевдокод с реализацией `getAllMessage`:

```
@Override
@GET
@Consumes({"application/xml", "application/json"})
@Produces({"application/json"})
@Path("/getAllMessage/")
public MessageViewAllResponse getAllMessage(
       @Context HttpHeaders headers)
       throws ServiceInvocationException {
    // TODO: реализовать логику
}
```

createMessage

Сохраняет объявление, сообщение, вопрос или ответ, отправленный пользователем:

- контекст – `MessageService/createMessage`;
- метод – `POST`;
- потребляет – `application/xml`, `application/json`;
- производит – `application/json`;
- вход – `HttpHeaders`, `MessageRequest`;
- выход – код ответа (например, успех или неудача).

Далее показан псевдокод с реализацией `createMessage`:

```
@Override
@POST
@Consumes({"application/xml", "application/json"})
@Produces({"application/json"})
@Path("/createMessage/")
public RestResponse createMessage(
       @Context HttpHeaders headers,
       MessageRequest req)
       throws ServiceInvocationException {
    // TODO: реализовать логику
}
```

Поиск

Служба поиска позволяет пользователям выполнять поиск по текстовым фразам. В процессе поиска просматриваются все сущности (таблицы в базе данных), такие как претензии, каталог и доска объявлений. Поиск взаимодействует с базой данных посредством слоя объектов доступа к данным (Data Access Object, DAO) и извлекает всю найденную информацию, используя механизм объектно-реляционного отображения в Hibernate.

Ниже приводится определение класса компонента:

```
@Component
@Path("/Search/Service")
public class SearchServiceImpl implements SearchService {
```

- контекст – `SearchService/search`; ;
- метод – `GET`;
- потребляет – `application/xml`, `application/json`;
- производит – `application/json`;
- вход – `HttpHeaders`, текст для поиска;
- выход – код ответа (например, успех или неудача).

Далее показан псевдокод с реализацией компонента:

```
@Override
@GET
@Consumes({"application/xml", "application/json"})
@Produces({"application/json"})
@Path("/search")
```

```
public MessageViewResponse search(
       @Context HttpHeaders headers,
       @PathParam("title")String title)
       throws ServiceInvocationException {
    //TODO: реализовать DAO
}
```

Сборка приложения

Теперь, познакомившись с архитектурой, веб-службами и разными зависимостями приложения, загрузите код, скомпилируйте и опробуйте его.

Настройка Eclipse

Для разработки этого приложения была выбрана интегрированная среда Eclipse IDE. Вы можете использовать любой другой, более привычный для вас инструмент, но далее приводятся инструкции по настройке Eclipse в Windows (пропустите этот раздел, если у вас уже установлена Eclipse):

1) загрузите Eclipse со страницы https://eclipse.org/downloads/index-developer.php;
2) распакуйте архив. Eclipse требует, чтобы путь к каталогу с JRE был включен в переменную `PATH`;
3) выполните двойной щелчок на файле *eclipse.exe* (см. рис. 11.2);

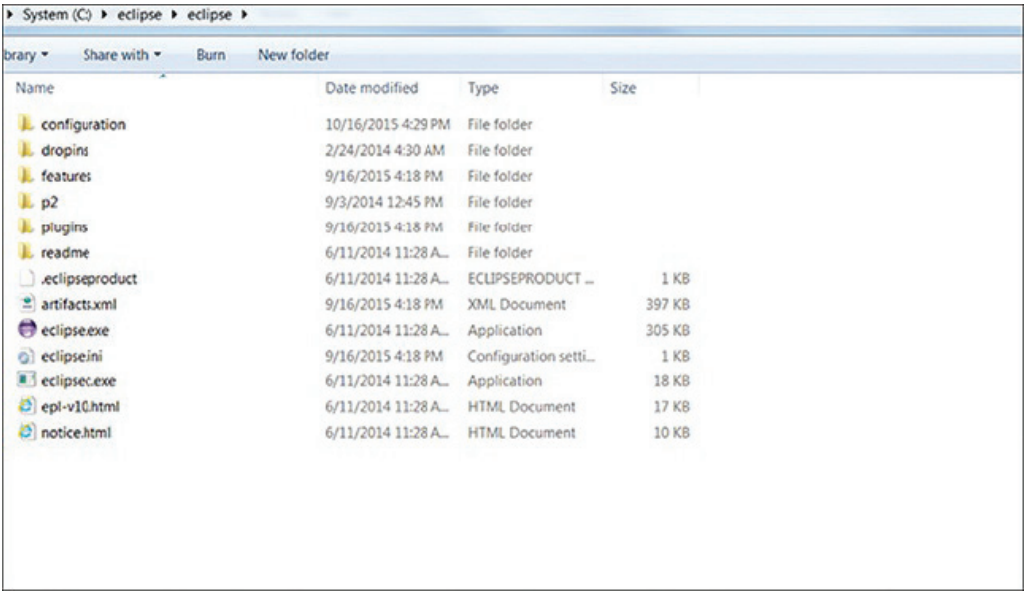


Рис. 11.2 ❖ Распакованный архив с Eclipse IDE

4) щелкните правой кнопкой мыши на обозревателе пакетов и в контекстном меню выберите пункт **New Java Projects** (Новые проекты Java). В открывшем-

ся диалоге дайте проекту имя `Helpdesk`, как показано на рис. 11.3, и оставьте значения по умолчанию в остальных полях;

5) снимите флажок **Use default location** (Использовать местоположение по умолчанию), выберите каталог, куда вы скопировали код приложения (как говорилось в начале главы), и щелкните на кнопке **Open** (Открыть), а затем на кнопке **Next** (Далее);

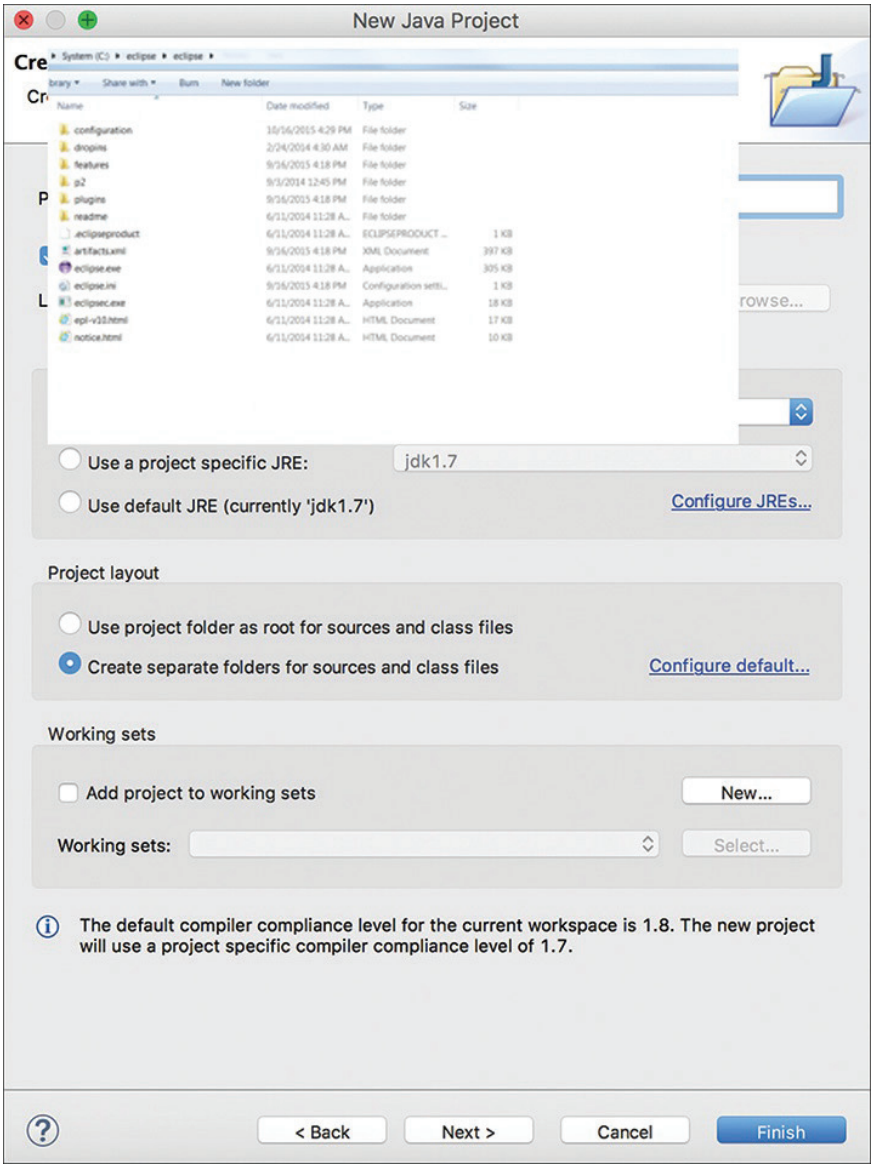


Рис. 11.3 ❖ Создание нового проекта на Java (с именем Helpdesk)

6) щелкните на кнопке **Finish** (Завершить), как показано на рис. 11.4.

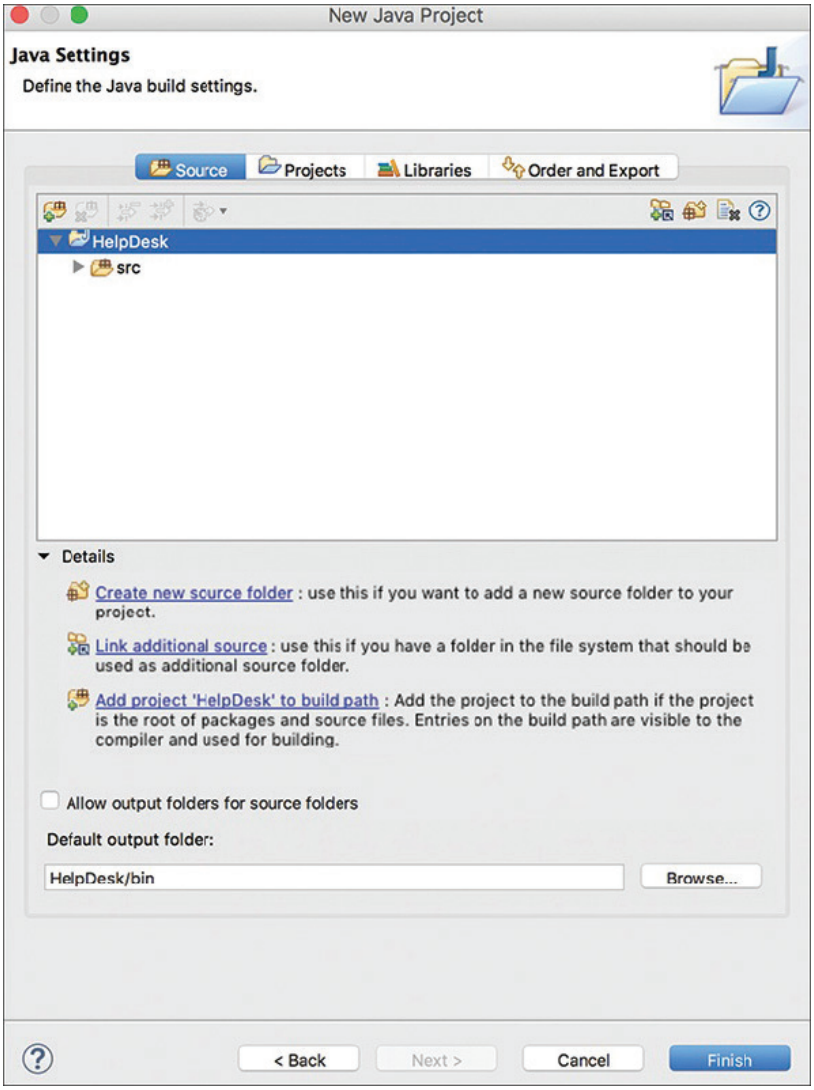


Рис. 11.4 ❖ Завершение настройки приложения

На рис. 11.5 показано, как выглядит дерево исходных файлов приложения после завершения настройки.

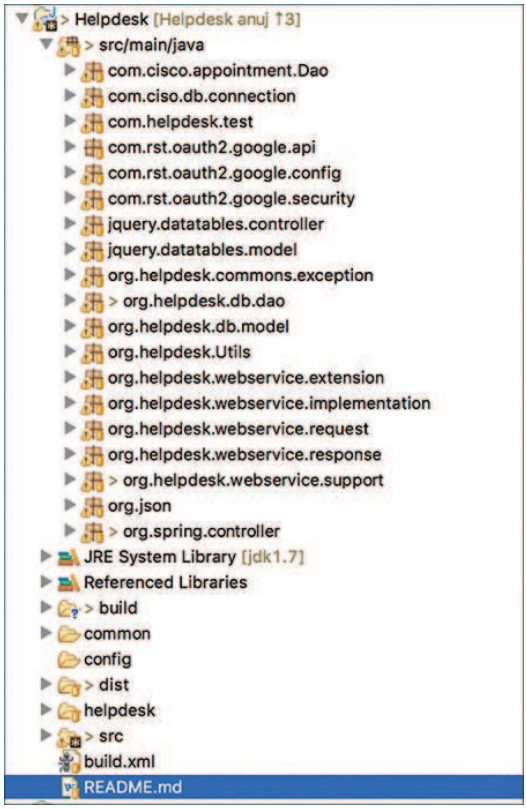


Рис. 11.5 ❖ Все исходные файлы приложения

Компиляция приложения

Далее описывается, как запустить компиляцию и получить файл WAR, готовый для развертывания. Компиляция будет выполняться с помощью старого проверенного инструмента сборки Apache Ant:

1) перед сборкой WAR-файла нужно настроить базу данных в файле *applicationContext.xml*. Он находится в *<Project Location>/src/main/webapp/WEB-INF/applicationContext.xml*. Измените свойства `url`, `username` и `password` компонента `DataSource`, как показано ниже. Здесь вы должны указать те же учетные данные, что были настроены после установки MySQL (подробнее об этом говорится ниже):

```
<bean id="DataSource" destroy-method="close"
      class="org.apache.tomcat.jdbc.pool.DataSource">
  <property name="driverClassName"
            value="com.mysql.jdbc.Driver" />
  <property name="url"
            value="jdbc:mysql://<dbhost>:<dbport>/<dbname>" />
```

```
  <property name="username" value="<Username>" />
  <property name="password" value="<Password>" />
  <property name="initialSize" value="5" />
  <property name="maxActive" value="50" />
  <property name="validationQuery"
            value="select 1 from dual" />
  <property name="testWhileIdle" value="true" />
  <property name="testOnBorrow" value="true" />
  <property name="minIdle" value="020000" />
  <property name="minEvictableIdleTimeMillis"
            value="30000000" />
  <property name="timeBetweenEvictionRunsMillis"
            value="6000000" />
  <property name="removeAbandoned" value="true"/>
  <property name="removeAbandonedTimeout" value="30000" />
  <property name="logAbandoned" value="true" />
  <property name="maxWait" value="120000" />
</bean>
```

2) создайте новый файл *Build.xml* в корневом каталоге проекта и добавьте в него следующий код:

```
<project name="projects" default="jar" basedir=".">

  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="dist" location="dist"/>
  <property name="jar.location" location="${dist}/lib"/>

  <dirname property="projects.basedir"
           file="${ant.file.projects}"/>
  <echo>projects.basedir=${projects.basedir}</echo>

  <echo>Inside smartview project:
        smartview.basedir=${smartview.basedir}</echo>

        <path id="project.classpath">
          <fileset refid="sv.jars"/>
          <fileset refid="common.dist"/>
        </path>

  <filelist id="project.build.files" dir="${projects.basedir}">
    <file name="build.xml" />
  </filelist>

  <fileset id="sv.jars" dir="${projects.basedir}">
    <include name="src/main/lib/*.jar"/>
  </fileset>

  <fileset id="common.jars" dir="${projects.basedir}">
    <include name="src/main/lib/*.jar"/>
  </fileset>

  <fileset id="common.dist" dir="${projects.basedir}">
    <include name="dist/lib/*.jar"/>
  </fileset>
```

3) запустите компиляцию и создайте JAR-файл с этими целями:

```
<target name="compile.individual" depends="init">
  <javac includeantruntime="false"
         debug="true"
         compiler="javac1.6"
         srcdir="${src}" destdir="${build}">
    <classpath refid="project.classpath"/>
  </javac>
</target>

<target name="jar.individual" depends="compile.individual">
  <mkdir dir="${jar.location}"/>
  <mkdir dir="${build}/META-INF"/>

  <copy todir="${build}/META-INF">
    <fileset dir="${src}/main/resource/META-INF"
             includes="*.xml"/>
  </copy>

  <jar jarfile=
       "${jar.location}/org-${ant.project.name}.jar"
       basedir="${build}"/>
</target>

<!-- Методы, используемые только на верхнем уровне -->
<target name="jar" depends="init">
  <mkdir dir="${dist}/lib"/>
  <subant target="jar.individual">
    <filelist refid="project.build.files"/>
  </subant>
</target>
```

4) создайте WAR-файл с помощью следующих целей:

```
<target name="copy.files" depends="jar">
  <copy todir="${stage.war.lib}" flatten="true">
    <fileset dir="${projects.basedir}"
             includes="*/dist/lib/*.jar"
             excludes="*test*.jar" />
  </copy>

  <copy todir="${stage.war.lib}" flatten="true">
    <fileset dir="${projects.basedir}"
             includes="common/configproperties/*.xml" />
  </copy>

  <copy todir="${stage.war.lib}" flatten="true">
    <fileset refid="common.jars"/>
  </copy>

  <copy todir="${stage.war.lib}" flatten="true">
    <fileset refid="sv.jars"/>
  </copy>
</target>

<target name="war" depends="init.war,copy.files">
```

```
  <war destfile="dist/lib/helpdesk.war"
       webxml="src/main/webapp/WEB-INF/web.xml">
    <fileset dir="src/main/webapp">
      <exclude name="**/.svn"/>
    </fileset>
    <lib dir="src/main/webapp/WEB-INF/lib" />
    <classes dir="${build}/classes" />
  </war>
</target>
```

5) щелкните правой кнопкой мыши на файле *Build.xml* и выберите в контекстном меню **Run As** → **Ant Build...** (Запустить как → Сборка Ant...), как показано на рис. 11.6;

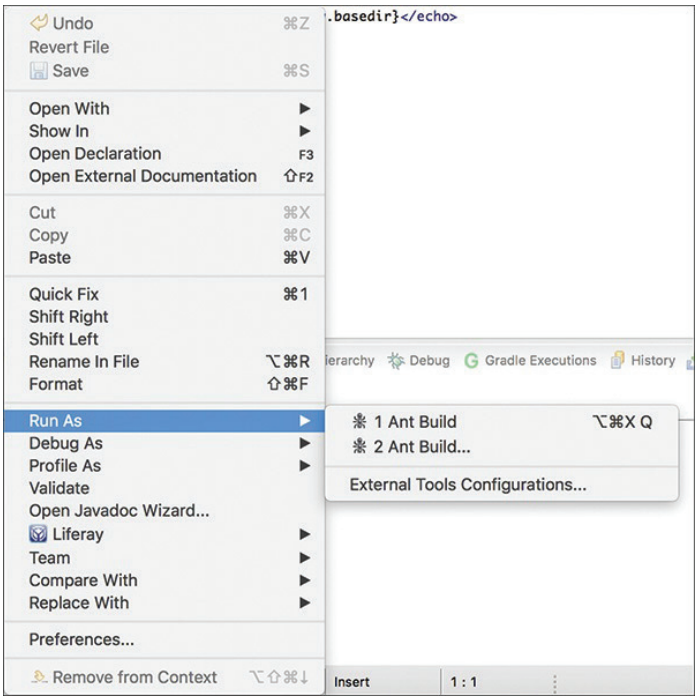


Рис. 11.6 ❖ Варианты запуска файла Build.xml

6) в следующем окне выберите все цели.

Локальная среда готова, и теперь у нас имеется готовый WAR-файл с именем *helpdesk.war* в каталоге *<Project Location>/helpdesk/dist/lib*.

Развертывание и настройка

Мы разместим приложение и все его службы на единственной виртуальной машине в Amazon Web Services (AWS). Я предлагаю следовать инструкциям в этом разделе, чтобы получить практический навык. Для начала нужно запустить

экземпляр EC2 в AWS. Для наших целей вполне подойдет средняя виртуальная машина с операционной системой Ubuntu. Tomcat 7 и MySQL должны быть установлены заранее (вместе с Tomcat 7 будут также установлены Java и другие зависимости):

1) чтобы установить Tomcat 7 в каталог */var/lib/tomcat7*, выполните следующую команду:

   ```
   sudo apt-get install tomcat7;
   ```

2) она должна установить и запустить службу. Убедиться в этом можно следующей командой:

   ```
   sudo service tomcat7 status.
   ```

   Механизм обслуживания сервлетов Tomcat должен быть запущен в отдельном процессе. Запустить и остановить Tomcat можно командами:

   ```
   sudo service tomcat7 start;
   sudo service tomcat7 stop;
   ```

3) выполните следующую команду, чтобы установить сервер MySQL:

   ```
   sudo apt-get install mysql-server;
   ```

4) в процессе установки вам будет предложено ввести пароль пользователя root. Введите пароль, чтобы завершить установку;
5) после установки сервер MySQL должен быть запущен. Убедиться в этом можно с помощью команды

   ```
   sudo service mysql status.
   ```

   Запустить и остановить MySQL можно командами:

   ```
   sudo service mysql start;
   sudo service mysql stop;
   ```

6) создайте базу данных с именем `helpdesk`:

   ```
   create database helpdesk;
   ```

7) скопируйте файл *application.properties* из каталога *<Project Location>/src/main/webapp/WEB-INF/* в каталог *lib* внутри каталога установки Tomcat. Эти свойства используются в проекте;
8) скопируйте файл *jstl.1.2.jar* из каталога *<Project Location>/src/main/lib/* в каталог *lib* внутри каталога установки Tomcat. Это библиотека для поддержки тегов `jsp`;
9) экземпляр Tomcat по умолчанию прослушивает порт с номером 8080. Проверьте это, открыв в браузере страницу *http://<yourhost>:8080/console*;
10) теперь можно развернуть веб-приложение из консоли Tomcat. Щелкните на кнопке **Browse** (Обзор) и отыщите WAR-файл, созданный ранее. Затем щелкните на кнопке **Deploy** (Развернуть), как показано на рис. 11.7.

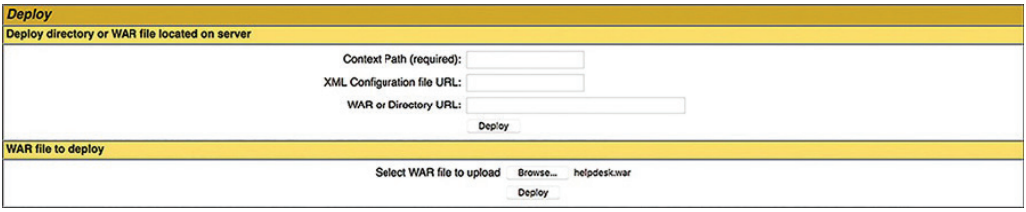


Рис. 11.7 ❖ Развертывание WAR-файла

Как можно видеть на рис. 11.8, приложение развернуто на сервере Tomcat и доступно по адресу: *http://<yourhost>:8080/helpdesk*.

Рис. 11.8 ❖ Местоположение приложения на сервере Tomcat

Все приложение целиком упаковано в один WAR-файл. В этой точке приложение должно быть запущено в вашей системе. На рис. 11.9 показаны все зависимости между модулями в приложении.

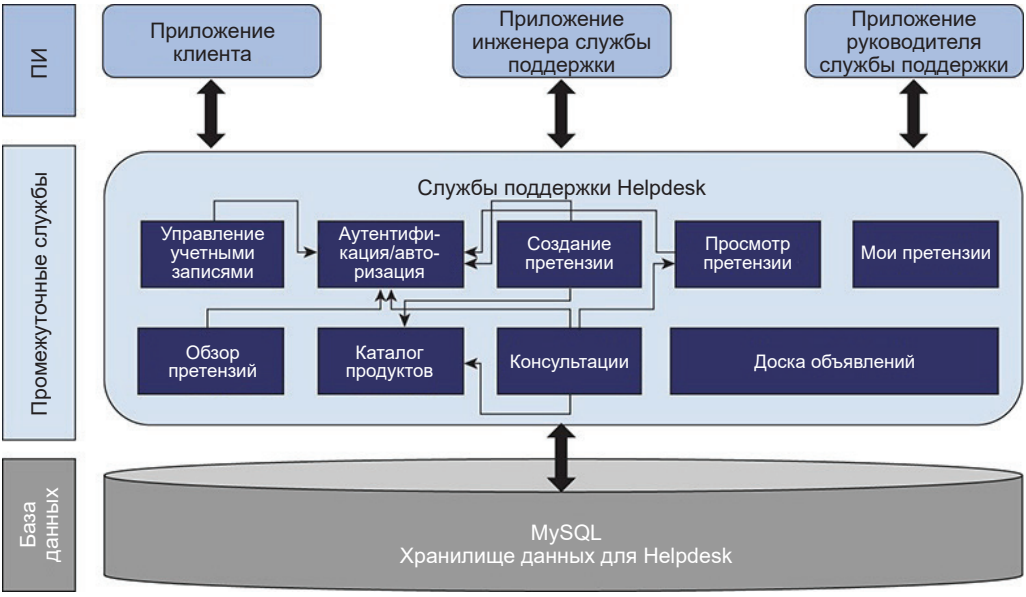


Рис. 11.9 ❖ Зависимости между модулями в приложении Helpdesk

Новые требования и исправление ошибок

Вообразите, что приложение запущено в эксплуатацию и обслуживает клиентов (начался цикл поддержки программного обеспечения). Со временем стали появляться предложения по расширению и изменению функциональных возможностей, а также стали обнаруживаться ошибки. Все эти предложения влекут за собой

изменение программного кода и пересборку приложения. Давайте посмотрим, какие проблемы нас поджидают в процессе эксплуатации монолитного приложения.

Допустим, нам потребовалось добавить дополнительный параметр в службу просмотра претензий, который никак не влияет на другие компоненты. Следующий код иллюстрирует изменение запроса:

```
public TicketResponse createHdTicket(
       @Context HttpHeaders headers,
       TicketRequest ticketRequest)
       throws ServiceInvocationException{
```

Добавим новое свойство в обычный объект Java (веб-модель):

```
@Component
private String emailAddress;

@XmlElement
public String getEmailAddress() {
    return emailAddress;
}

public void setEmailAddress(String emailAddress) {
    this.emailAddress = emailAddress;
}
```

Добавим логику в слой DAO для получения свойства из базы данных:

```
private String saveToDatabase(TicketRequest ticketRequest){
    // добавить к существующему
    ticket.setEmailAddress(ticketRequest.getEmailAddress());
}
```

Все эти изменения выглядят довольно простыми. Однако в дальнейшем придется выполнить следующие действия:

1) снова скомпилировать приложение целиком и повторно развернуть его на сервере;
2) выполнить регрессионное тестирование, чтобы убедиться, что все другие службы работают без ошибок;
3) исправить любые ошибки и удовлетворить зависимости;
4) развернуть приложение в тестовом окружении и проверить его работоспособность под высокой нагрузкой, чтобы гарантировать неизменность качества обслуживания;
5) развернуть приложение в эксплуатационном окружении.

Если нет возможности развернуть приложение в режиме поддержки высокой доступности, значит, на время развертывания новой версии будет иметь место период простоя.

Эти шаги увеличивают время до ввода в эксплуатацию исправленной версии и минимизируют все преимущества гибкой разработки. Все это без учета постоянных изменений, когда вы создаете отдельную ветку с новым кодом, объединяете ее с основной и вновь выполняете цикл тестирования.

Существуют и другие проблемы, перечисленные ниже:

- **устранение ошибок.** После исправления каждой ошибки требуется заново развертывать приложение целиком, что чревато дополнительными про-

стоями, если не внедрить в инфраструктуру развертывания поддержку высокой доступности. Кроме того, в зависимости от используемой методологии управления циклом разработки время, необходимое для исправления ошибки, может еще больше увеличиться. В случае критической ошибки, например, возникнет потребность создать ветвь кода с «горячими исправлениями», что может усложнить базу кода и впоследствии вызвать проблемы при слиянии ветвей;

- **замена компонентов приложения.** Это еще одна ситуация, когда может потребоваться выполнение полного рефакторинга всего приложения. Представьте, что в организации решили воспользоваться облачными услугами для управления претензиями. В текущей реализации довольно трудно отделить от приложения модули, связанные с этой службой;
- **замена или добавление нового стека технологий.** В данном случае нет возможности свободно выбирать технологию для реализации новых модулей/функций без реализации всего приложения заново. Монолитная архитектура связала организацию по рукам и ногам с ранее выбранной технологией;
- **выборочное масштабирование.** Допустим, вам понадобилось масштабировать только модуль управления претензиями, чтобы справиться с увеличившейся нагрузкой. В данной реализации это сложно сделать, потому что компоненты монолитного приложения тесно связаны друг с другом. Чтобы отделить службу управления претензиями, например, придется провести гигантский рефакторинг кода, реализовать интеграцию с отдельной системой управления претензиями, произвести тестирование, развернуть новую архитектуру и т. д.;
- **обработка неисправностей.** Выход из строя одного компонента в монолитном приложении способен вызвать аварийное завершение всего приложения. Представьте, что в службе каталога продуктов возникла ошибка. В результате пользователи не смогут оставлять новые претензии. Вообще, претензия должна содержать ссылку на продукт, с которым она связана, чтобы ускорить процесс разрешения проблемы. Однако ошибка в службе каталога продуктов не должна препятствовать созданию претензий, т. е. она не должна вызывать сбоев в работе службы претензий. В монолитной архитектуре в том случае, если требуется заполнить поле из каталога продуктов, а сделать это невозможно из-за ошибки в службе, пользователь бессилен, даже если теоретически будет в состоянии грамотно описать проблему.

Это самые обычные потребности современного цифрового мира. И удовлетворять их в монолитных приложениях очень трудно и дорого. В следующих двух главах мы посмотрим, как избавиться от этих сложностей с помощью микросервисов и контейнеров.

Глава 12

Практический пример: миграция на архитектуру микросервисов

В главе 11 «Практический пример: монолитное приложение Helpdesk» мы создали традиционное веб-приложение для службы поддержки, следуя общепринятым стандартам. Цель главы состояла в том, чтобы на практике продемонстрировать сложности, с которыми может столкнуться организация, используя монолитную архитектуру. В этой главе мы изменим приложение, воспользовавшись полученными знаниями о микросервисах, а затем посмотрим, как избавиться от обозначенных сложностей.

В главе 4 «Миграция и реализация микросервисов» мы обсудили два возможных сценария, таких как создание совершенно нового приложения в виде комплекса микросервисов и постепенная миграция от монолитной архитектуры к архитектуре на основе микросервисов. Поскольку у нас уже имеется монолитное приложение, здесь мы пойдем путем постепенной миграции.

Планирование миграции

Допустим, что в далеком 2005 г., когда создавалось приложение, бизнес предъявлял к нему следующие требования:

- поддержка примерно 500 000 клиентов с возможностью для каждого оставлять свои претензии;
- все функции одинаково важны и должны быть доступны в любое время;
- приложение должно масштабироваться по горизонтали;
- сокращение числа претензий за счет поддержки поиска существующих решений.

Сейчас 2019 год. Мы накопили некоторый опыт и можем проанализировать, как клиенты используют приложение:

- число пользователей увеличилось до 1,5 млн и продолжает расти. С учетом роста популярности мобильных устройств ожидается, что через 2 года их число достигнет 3 млн;
- большинство пользователей использует две основные функции, такие как создание претензий и поиск решений;

- в некоторые функции, такие как доска объявлений, было внесено очень мало изменений;
- пики трафика наблюдаются два раза в году (в начале лета и в период праздников в ноябре и декабре);
- значительно возросло количество случаев, когда служба управления претензиями оказывалась неработоспособной из-за влияния других служб, таких как служба каталога продуктов;
- прогресс в области обработки естественного языка привел к тому, что пользователей больше не удовлетворяет простой поиск по ключевым словам. Они хотят, чтобы система понимала текст на простом разговорном языке, могла отыскивать решения и оказывать соответствующую помощь. Иначе говоря, пользователи хотят иметь возможность семантического поиска;
- наибольшее число запросов на улучшение касается службы управления претензиями.

Совершенно очевидно, что наше приложение прекрасно справляется со своими обязанностями, потому что число клиентов увеличилось, а приложение все еще способно обслуживать их. Кроме того, можно предположить, что приложение хорошо масштабируется по горизонтали для поддержки увеличившегося числа пользователей. Под масштабированием по горизонтали в данном случае понимается запуск дополнительных экземпляров приложения на отдельных машинах с соответствующей балансировкой нагрузки. Здесь важно отметить, что речь идет о равномерной масштабируемости, т. е. о масштабируемости всего приложения целиком, а не отдельных его компонентов, таких как служба управления претензиями.

Теперь предположим, что вам дали задание изменить приложение (привести его в соответствие с современными потребностями и обеспечить возможность масштабирования для поддержки до 3 млн пользователей). Кроме того, приложение с легкостью должно поддаваться дальнейшему развитию (быть открытым для изменения компонентов) по мере изменения технологий.

С учетом имеющихся у нас знаний выбор архитектуры микросервисов выглядит отличным решением. Приложение, развернутое в главе 11, не очень старое. Фактически оно уже использует архитектуру «модель – представление – контроллер» (Model – View – Controller, MVC) и веб-службы, поэтому было бы неразумно начинать все с самого начала. Кроме того, обратите внимание, что новые требования касаются лишь некоторых компонентов приложения, что тоже говорит в пользу выбора микросервисов. Итак, как мы поступим? Есть много возможных путей. Давайте используем знания, полученные в главе 4, и преобразуем имеющийся проект в приложение на основе микросервисов.

Оценка критериев выделения микросервисов

Критерии выделения микросервисов, описанные в главе 4, определяют один из возможных способов выбора компонентов монолитного приложения, которые следует в первую очередь преобразовать в микросервисы. Рассматривая новые требования и результаты анализа поведения пользователей, можно выделить соответствие таким семи критериям, как:

- **масштабирование.** Из первых двух новых требований со всей очевидностью вытекает необходимость масштабирования приложения. Два наиболее важных и часто используемых компонента – это службы поиска и управления претензиями, поэтому их стоит преобразовать в микросервисы;
- **лучшие технологии (или языки) программирования.** Из новых требований также вытекает, что мы должны реализовать семантический поиск, а для этой цели как нельзя лучше подходит Apache Solr – инструмент с открытым исходным кодом. Его использование поможет улучшить возможности поиска поддержкой релевантного и контекстно-зависимого поиска;
- **лучшие механизмы хранения данных.** Наше монолитное приложение использует базу данных MySQL для хранения всех своих данных. Конечно, претензии имеет смысл хранить в реляционной базе данных. Однако приложение можно было бы улучшить, поместив каталог продуктов в кеш в оперативную память и сохранив резервную копию в простом файле. Подобная организация имеет свои преимущества:
  - такой каталог легко обновить, просто скопировав обновленные файлы;
  - поскольку при работе с каталогом не требуется выполнять реляционные операции, простое чтение файла в памяти в виде списка ключей могло бы значительно увеличить быстродействие;
- **частота изменения.** Учитывая, что большая часть изменений приходится на службу претензий, есть смысл первой преобразовать ее в микросервис. Следуя той же логике и учитывая новые требования, не нужно преобразовывать в микросервис доску объявлений;
- **сложность развертывания.** Сложность развертывания нашего приложения никак не зависит ни от одного из его компонентов, поэтому данный критерий можно пропустить;
- **вспомогательные службы.** В соответствии с проведенным анализом служба претензий часто страдала из-за недоступности службы каталога продуктов или из-за проблем в ней. Мы должны замкнуть эту службу, т. е. обеспечить ее нормальное функционирование даже в том случае, если каталог продуктов окажется недоступен. Этот критерий явно указывает на необходимость выделения службы каталога продуктов в микросервис.

Единственное, что мы не обсудили, – это сезонные колебания трафика. Эту проблему легко решить и в текущей версии приложения, добавляя серверы приложений и базы данных в пиковые сезоны и убирая их в периоды трафика с обычной интенсивностью. Однако, опираясь на уже имеющиеся знания о микросервисах и подходах к преобразованию служб в микросервисы, мы можем сказать, что гораздо эффективнее было бы масштабировать компоненты, нагрузка на которые возрастает с увеличением трафика. Мы затронем этот аспект миграции на микросервисы в главе 13 «Практический пример: перенос приложения Helpdesk в контейнеры».

Выводы о миграции

Согласно новым требованиям и с учетом оценки критериев выделения микросервисов, мы приходим к выводу, что в микросервисы следует преобразовать следующие компоненты:

- каталог продуктов;
- управление претензиями;
- поиск.

Кроме того, мы добавим в приложение механизм поиска Solr. В настоящее время поиск выполняется методом сканирования содержимого базы данных, что является очень грубой реализацией поиска. Этот метод просто сопоставляет запрос с текстом в базе данных. Ни качество результатов, ни производительность не соответствуют современному уровню технологий.

Давайте кратко обсудим Solr. (Подробные инструкции по установке и настройке вы найдете в приложении B.) Solr – это механизм поиска, основанный на библиотеке Apache Lucene, написанный на языке Java и использующий библиотеку Lucene для индексации. Обращаться к этому механизму можно посредством разных REST API, включая XML и JSON. Вот основные возможности этого механизма:

- продвинутый полнотекстовый поиск;
- оптимизированная обработка объемного веб-трафика;
- полноценный HTML-интерфейс администратора;
- поддержка статистики о работе сервера через Java Management Extensions (JMX) для мониторинга;
- линейная масштабируемость, репликация с автоматическим индексированием и автоматическая обработка ошибочных ситуаций с последующим восстановлением работоспособности;
- быстрый механизм индексирования в режиме реального времени;
- гибкие настройки в формате XML;
- расширяемая архитектура с возможностью подключения сторонних плагинов.

Дополнительную информацию ищите на сайте проекта: http://lucene.apache.org/solr.

Влияние на архитектуру

После преобразования каталога продуктов, управления претензиями и поиска в автономные микросервисы архитектура приложения будет выглядеть, как показано на рис. 12.1.

Как видите, службы каталога, поиска и претензий выделены из монолитной парадигмы и развернуты как независимые микросервисы. Отдельные микросервисы разворачиваются за балансировщиком нагрузки (БН), таким как HAProxy, что обеспечивает их высокую доступность и возможность масштабирования.

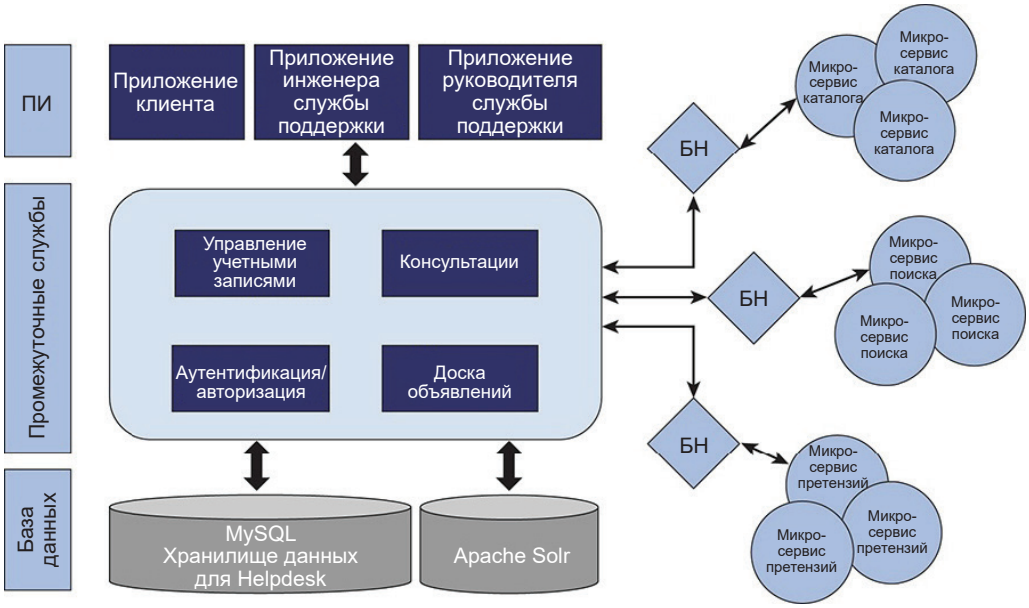


Рис. 12.1 ❖ Новая архитектура приложения с автономными микросервисами

Преобразование в микросервисы

Определив новую архитектуру на основе микросервисов, можно приступить к преобразованию трех компонентов, выделенных из монолитного приложения, в независимые микросервисы. Мы подробно рассмотрим преобразование служб каталога и поиска, но оставим преобразование службы претензий вам в качестве самостоятельного упражнения, чтобы вы могли получить практический опыт. При желании вы можете подсмотреть окончательный результат в репозитории GitHub: https://github.com/kocherMSD/Helpdesk_Microservices.git.

Каталог продуктов

В этом разделе мы перенесем код службы каталога продуктов из монолитного приложения Helpdesk в отдельную единицу сборки, включая интерфейсы, классы реализации службы, вспомогательные классы и файлы конфигурации. Эта новая единица сборки будет зависеть только от сторонних библиотек, действительно необходимых службе.

Затем мы настроим сборку службы каталога с помощью Apache Maven вместо Apache Ant. Основная причина состоит в том, что Apache Maven – более современная и более гибкая система сборки, обладающая превосходным механизмом управления внешними зависимостями.

Наконец, мы модифицируем единицу сборки службы каталога, обновив внешние зависимости до последних версий. Благодаря этому мы получим возможность использовать улучшенные реализации сторонних библиотек.

Далее подробно описываются шаги преобразования службы каталога продуктов в микросервис.

Шаги по преобразованию

Мы решили создать микросервис каталога продуктов, используя существующий код из монолитного приложения. По сути, это будут отдельный проект и служба. Достижению этой цели помогут следующие шаги:

1) создайте в Eclipse новый проект с именем `catalog-svc`;
2) загрузите и установите Apache Maven, как описывается на странице https://maven.apache.org/install.html;
3) создайте в корневом каталоге проекта файл *pom.xml* для Maven и определите в нем необходимые зависимости. Подробности смотрите в файле на GitHub: https://github.com/kocherMSD/Helpdesk_Microservices/blob/master/catalogsvc/pom.xml;
4) создайте интерфейс, реализацию, вспомогательный класс, объект доступа к данным (Data Access Object, DAO) и XML-файл *applicationContext.xml*, описывающий контекст приложения.

   Согласно определению микросервиса, нам понадобится один интерфейс, реализация службы, вспомогательный класс и Java-класс DAO. Ниже приводится псевдокод с определением и реализацией микросервиса, однако мы настоятельно советуем взять код из репозитория GitHub:

   a. Интерфейс службы:

```
public interface CatalogService extends BeanFactoryAware,
ApplicationContextAware {
    public abstract ProductDetailsResponse getCatalog(
        @Context HttpHeaders headers,
        String userId)
        throws ServiceInvocationException;

    public abstract CatalogResponse addCatalog(
        @Context HttpHeaders headers,
        CatalogRequest req)
        throws ServiceInvocationException;

    public abstract CatalogResponse updateCatalog(
        @Context HttpHeaders headers,
        CatalogRequest req)
        throws ServiceInvocationException;

    public abstract CatalogResponse deleteCatalog(
        @Context HttpHeaders headers,
        CatalogRequest req)
        throws ServiceInvocationException;
}
```

   b. Реализация службы:

```
@Component
@Path("/CatalogService")
public class CatalogServiceImpl implements CatalogService {

    @Override
```

```
    @GET
    @Consumes({"application/xml", "application/json"})
    @Produces({"application/json"})
    @Path("/getCatalog/{customerId}")
    public ProductDetailsResponse getCatalog(
        @Context HttpHeaders headers,
        @PathParam("customerId") String customerId)
        throws ServiceInvocationException {
        //TODO: реализовать
    }
```

c. Вспомогательный класс:

```
public class CatalogServiceHelper {
    CatalogDao dao=null;
    //TODO:
}
```

d. Класс DAO:

```
public class CatalogDao extends DataService{
    //TODO:
}
```

5) измените файл *applicationContext.xml*, оставив в нем только компоненты микросервиса. Структура нового проекта должна выглядеть, как показано на рис. 12.2;

Рис. 12.2 ❖ Структура нового проекта

6) выполните команду `mvn install` в каталоге, где хранится *pom.xml*. Она создаст WAR-файл *catalog-svc*;
7) разверните WAR-файл на сервере Tomcat, где действует монолитное приложение: *http://<host>:<port>/catalog-svc/rest/catalogservice/<Rest Verb>*. Конечная точка веб-службы для нашего микросервиса изменится;
8) не забудьте, что мы используем ту же самую базу данных. Поэтому перед сборкой WAR-файла измените настройки базы данных в *applicationContext*.

xml. Значения свойств `url`, `username` и `password` компонента `DataSource` должны соответствовать вашим учетным данным:

```
<bean id="DataSource" destroy-method="close"
      class="org.apache.tomcat.jdbc.pool.DataSource">
    <property name="driverClassName"
              value="com.mysql.jdbc.Driver" />
    <property name="url"
              value="jdbc:mysql://<dbhost>:<dbport>/<dbname>"/>
    <property name="username" value="<Username>"/>
    <property name="password" value="<Password>"/>
    <property name="initialSize" value="5"/>
    <property name="maxActive" value="50"/>
    <property name="validationQuery"
              value="select 1 from dual"/>
    <property name="testWhileIdle" value="true"/>
    <property name="testOnBorrow" value="true"/>
    <property name="minIdle" value="020000"/>
    <property name="minEvictableIdleTimeMillis"
              value="30000000"/>
    <property name="timeBetweenEvictionRunsMillis"
              value="6000000"/>
    <property name="removeAbandoned" value="true"/>
    <property name="removeAbandonedTimeout"
              value="30000"/>
    <property name="logAbandoned" value="true"/>
    <property name="maxWait" value="120000"/>
</bean>
```

Служба поддержки претензий

Чтобы превратить службу претензий в отдельный микросервис, нужно перенести соответствующий код из монолитного приложения в отдельную единицу сборки, включая интерфейсы, классы реализации службы, вспомогательные классы и файлы конфигурации. Эта новая единица сборки будет зависеть только от сторонних библиотек, действительно необходимых службе.

Далее, руководствуясь теми же причинами, нужно настроить сборку службы претензий с помощью Apache Maven вместо Ant.

Процесс преобразования включает те же шаги, что и в случае со службой каталога продуктов.

Поиск

Как уже отмечалось, эта служба выполняет очень упрощенный поиск в базе данных. Теперь мы решили добавить механизм поиска Solr, чтобы получить более мощные возможности поиска. Мы сохраним поиск старым способом и просто добавим к нему поиск с использованием Solr. Результаты обоих поисков будут отображаться в одном интерфейсе. По этой причине нам нужно изменить представление поиска, добавив данное усовершенствование.

Простой поиск в базе данных

Перенос кода службы поиска осуществляется точно так же, как код двух предыдущих служб. Как вы уже знаете, для этого нужно перенести код из монолитного приложения в отдельную единицу сборки, включая интерфейсы, классы реализации службы, вспомогательные классы и файлы конфигурации. И снова будем использовать систему сборки Apache Maven, которая поможет в получении внешних зависимостей, необходимых службе.

Поиск с применением Solr

Чтобы добавить поддержку семантического поиска, сначала нужно установить и настроить механизм Solr. Все необходимые для этого инструкции вы найдете в приложении B. После этого можно создать новый микросервис. Мы реализуем продвинутый поиск в виде отдельного микросервиса и самостоятельной единицы сборки, снова используя систему Apache Maven.

Далее следует код реализации веб-службы поиска, использующей Solr:

```
@POST
@Consumes({"application/xml", "application/json"})
@Produces({"application/json"})
@Path("/solrSearch")
public QueryResponse search(
    @Context HttpHeaders headers,
    SearchRequest request)
```

Вот как выполняется запрос к интерфейсу Solr:

```
HttpSolrServer solr = new HttpSolrServer(
                    "http://<ip of solr host>
                    :8983/solr/helpdesk");
SolrQuery query = new SolrQuery();
query.setQuery(request.getQuery());
query.setStart(0);
QueryResponse response = solr.query(query);
```

Механизм Solr обладает массой возможностей, таких, например, как применение фильтров, но их обсуждение выходит далеко за рамки книги. Если вам интересно, за дополнительной информацией обращайтесь на страницу проекта: http://lucene.apache.org/solr.

Теперь рассмотрим процесс сборки и развертывания приложения.

Сборка и развертывание приложения

Итак, мы выделили три компонента из монолитного приложения и оформили их в виде трех микросервисов:

- каталог продуктов;
- управление претензиями;
- поиск.

Теперь посмотрим, какие изменения коснулись этих микросервисов, включая сборку, настройку и развертывание.

Настройка кода

Для сборки исходного монолитного приложения использовался инструмент Apache Ant. Поскольку новый проект приобрел более модульную организацию и возникла потребность расширенного управления зависимостями, для сборки отдельных микросервисов стала использоваться система Apache Maven. Ant не имеет встроенных средств управления зависимостями, хотя этот недостаток можно восполнить с помощью инструмента Ivy. Этот подход демонстрирует главную идею: каждый микросервис может собираться из исходного кода отдельно, если это потребуется.

Код отдельных микросервисов доступен в репозитории GitHub: https://github.com/kocherMSD/Helpdesk_Microservices.git.

Сборка микросервисов

Собирать отдельные микросервисы можно двумя способами: из командной строки или автоматически, из интегрированной среды разработки, такой как Eclipse:

- **сборка из командной строки.** Чтобы собрать проект Maven из командной строки, выполните команду `mvn`. Она должна выполняться в каталоге проекта, где хранится соответствующий файл *pom.xml*. Сборку отдельных микросервисов рекомендуется выполнять командой `mvn clean package`. Она удалит старые и ненужные файлы и упакует результат сборки в WAR-файл, готовый к развертыванию;
- **сборка из Eclipse**. После импортирования проекта в Eclipse щелкните правой кнопкой на имени проекта, выберите в контекстном меню пункт **Run As** (Запустить как), а затем пункт **Run configurations** (Настройка запуска). В окне **Run configuration** введите `clean package` в поле цели и щелкните на кнопке **Run** (Запустить). После этого Eclipse должна собрать код и упаковать его в WAR-файл, готовый для развертывания в контейнере приложений, таком как Tomcat.

Развертывание и настройка

Существует несколько способов развертывания микросервисов, и каждый имеет свои достоинства и недостатки. Вкратце познакомимся с ними, чтобы вы могли составить собственное мнение. В главе 13 мы глубже исследуем вопросы развертывания, познакомившись с приемами автоматизированного развертывания, масштабирования и т. д.:

- **несколько микросервисов на одной машине.** В этом случае стратегия сводится к развертыванию нескольких микросервисов на одной машине (физической или виртуальной). Главным преимуществом этого подхода является более эффективное распределение ресурсов за счет совместного использования процессора, памяти, сети и т. д. несколькими микросервисами или экземплярами. Недостатком же является отсутствие изолированности микросервисов друг от друга. Кроме того, неправильно действующий микросервис может задействовать всю память и все процессоры хоста, ничего не оставив другим;

- **один микросервис в отдельной виртуальной машине.** Главным преимуществом этого подхода является полная изоляция каждого микросервиса вследствие заключения их в капсулы виртуальных машин. Каждый микросервис имеет полный доступ ко всей выделенной памяти, процессорам и сети. Основной недостаток – неэффективность использования ресурсов. Мощности виртуальных машин могут использоваться не полностью. Впрочем, этот недостаток можно ослабить, выделив виртуальной машине лишь необходимый объем ресурсов и настроив ее автоматическое масштабирование;
- **каждый микросервис в отдельном контейнере.** Чтобы развернуть микросервис в контейнере, его достаточно упаковать для запуска внутри этого контейнера. После этого вы сможете запустить столько контейнеров, сколько потребуется. Главным преимуществом данного подхода является изоляция служб друг от друга контейнерами. Существует возможность контролировать ресурсы, потребляемыми контейнером, и управлять ими. Однако, в отличие от виртуальных машин, контейнеры очень легковесны. Их легко собирать, упаковывать и запускать. Запуск контейнера происходит очень быстро, потому что при этом не требуется загружать операционную систему, как это происходит в виртуальной машине. Главный недостаток контейнеров – недостаточная технологическая зрелость. С появлением Docker в 2013 г. контейнеры стали доступнее, однако технология продолжает развиваться, и в ней пока остаются нерешенными такие проблемы, как безопасность, управление масштабированием контейнеров и др.

Для простоты развернем наши новые микросервисы на том же сервере Tomcat, где находится монолитное приложение. В главе 13 мы упакуем эти микросервисы в контейнер Docker и развернем их по отдельности.

Ниже перечислены шаги, которые нужно выполнить, чтобы развернуть приложение Helpdesk с новыми микросервисами:

1) для начала перечислите вновь созданные микросервисы, выделенные из монолитного приложения, в файле *application.properties*, как показано ниже:

```
endPoints.serachEndPoint=
    http://host:port/search-svc/rest/SerachService/search
endPoints.getCatalog=
    http://host:port/ticketing-svc/rest/CatalogService/getCatalog
endPoints.createTicket=
    http://host:port/catalog-svc/rest/TicketService/
createTicket
```

2) измените реализацию представления поиска в файле *search.jsp*, входящем в состав монолитного приложения, и добавьте дополнительную кнопку семантического поиска. Вызовите службу поиска на основе Solr из функции на JavaScript:

```
function solrsearch()
{
    var solrSearchEndPoint=
        <%= props.getProperty(
            "endPoints.solrSearchEndPoint") %>';
    var searchText=document.getElementById("searchText").value;
```

```
    if(searchText=='')
    {
        alert('Empty text. Please provide value in text');
    }

    var dataToSend= {"query":searchText};
    $.ajax({headers: {
                'Accept': 'application/json',
                'Content-Type': 'application/json'
            },
            url: solrSearchEndPoint,
            type: 'POST',
            dataType: 'json',
            data: JSON.stringify(dataToSend),
            success: function(data, textStatus, jqXHR) {
                $("#solrresults").empty();
                var docs = data.results;
                $.each(docs, function(i, item) {
                    $('#solrresults').prepend($('<div>' +
                        objToString(item) + '</div>'));
                });
                var total = 'Found ' + docs.length + ' results';
                $('#solrresults').prepend('<div>' + total + '</div>');
            }
        }).fail(function (jqXHR, textStatus, error) {

        // Обработать ошибки
        alert(jqXHR.responseText);
    });
}
```

3) создайте отдельный WAR-файл для каждого микросервиса, как показано на рис. 12.3. Используйте для этого файл *pom.xml* для Apache Maven, как описывалось в этой главе;

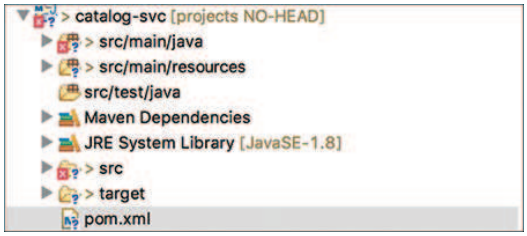


Рис. 12.3 ❖ Структура микросервиса в Eclipse

4) выполните сборку, используя систему Maven, как показано на рис. 12.4.

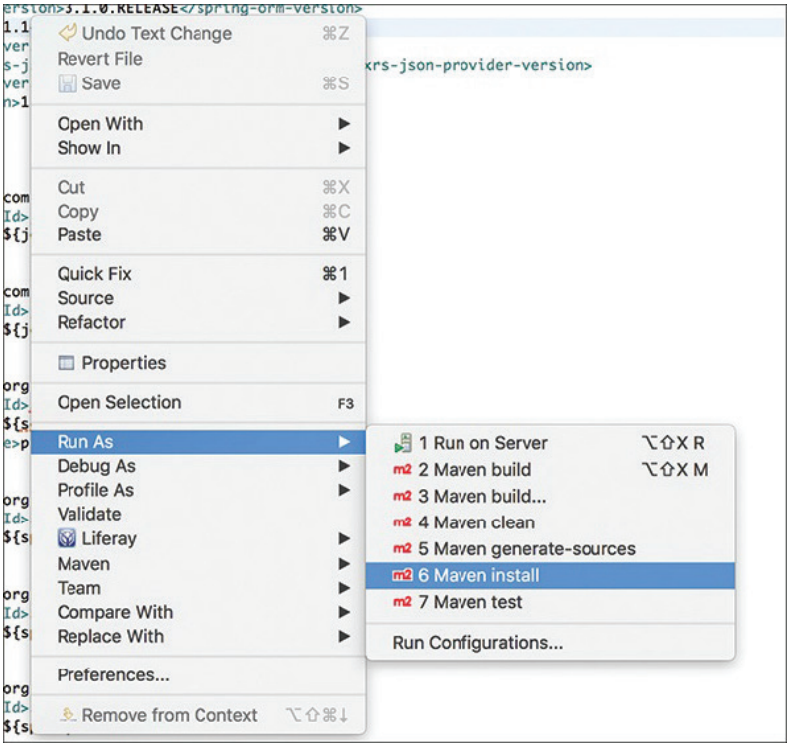


Рис. 12.4 ❖ Выполнение сборки с использованием Maven

Если сборка прошла успешно, должно появиться сообщение, изображенное на рис. 12.5;

```
[INFO] Processing war project
[INFO] Copying webapp resources [/opt/projects/BOOKCODE/catalog-svc/src/main/webapp]
[INFO] Webapp assembled in [1523 msecs]
[INFO] Building war: /opt/projects/BOOKCODE/catalog-svc/target/catalog-svc.war
[INFO] WEB-INF/web.xml already added, skipping
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ catalog-svc ---
[INFO] Installing /opt/projects/BOOKCODE/catalog-svc/target/catalog-svc.war to /Users/anujsin/.m2/repository/org/helpdesk/services/catalog-svc.
[INFO] Installing /opt/projects/BOOKCODE/catalog-svc/pom.xml to /Users/anujsin/.m2/repository/org/helpdesk/services/catalog-svc/1.0.0/catalog-
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 7.783 s
[INFO] Finished at: 2017-09-10T16:24:17-05:00
[INFO] Final Memory: 14M/210M
[INFO] ------------------------------------------------------------------------
```

Рис. 12.5 ❖ Вывод в процессе сборки

5) выполните шаги, перечисленные выше, для остальных микросервисов. Скопируйте все полученные WAR-файлы в каталог *webapp* сервера Tomcat.

Вот как должна выглядеть структура каталогов после развертывания в Linux:

```
search-svc catalog-svc docs helpdesk host-manager
ROOT ticketing-svc.war search-svc.war catalog-svc.war
examples helpdesk.war manager ticketing-svc
```

Как видите, вместе с *helpdesk.war* в том же контейнере Tomcat были развернуты три дополнительных микросервиса.

Новые требования и исправления ошибок

Мы успешно выполнили перенос монолитного приложения на архитектуру микросервисов, стремясь удовлетворить требования бизнеса. Но, как мы знаем, любой бизнес редко останавливается на достигнутом и продолжает развиваться. Давайте рассмотрим возможные варианты наших действий в архитектуре микросервисов в ответ на изменение запросов бизнеса.

Представьте, что нам потребовалось добавить в службу просмотра претензий дополнительный параметр, никак не зависящий от других компонентов. Мы можем изменить запрос претензии, как показано ниже:

```
public TicketResponse createHdTicket(
       @Context HttpHeaders headers,
       TicketRequest ticketRequest)
       throws ServiceInvocationException{
```

Добавить свойство в веб-модель – обычный Java-объект:

```
@Component
private String emailAddress;

@XmlElement
public String getEmailAddress() {
    return emailAddress;
}

public void setEmailAddress(String emailAddress) {
    this.emailAddress = emailAddress;
}
```

И добавить логику в слой DAO, чтобы получить это свойство из базы данных:

```
private String saveToDatabase(TicketRequest ticketRequest){
    // добавить к существующему
    ticket.setEmailAddress(ticketRequest.getEmailAddress());
}
```

Обратите внимание, что это изменение в службе поддержки претензий никак не влияет на другие службы. Поэтому нам остается только протестировать эту службу и развернуть ее. Внесение тех же изменений в монолитное приложение потребовало бы заново собрать все приложение целиком и провести регрессионное тестирование, а это требует времени и сил. Это очень простой пример, но он наглядно иллюстрирует разницу между внесением изменений в монолитное приложение и в приложение на основе микросервисов.

Теперь обратимся к проблемам монолитных приложений, которые были подчеркнуты в главе 11, и посмотрим, действительно ли микросервисы помогли избавиться от них:

- **устранение ошибок.** Исправления касаются только той части приложения, где обнаружена ошибка. Если ошибка присутствует в коде одного микросервиса, достаточно изменить код только этого микросервиса и развернуть

его. Кроме того, поскольку каждый микросервис обычно развертывается за балансировщиком нагрузки, исправления можно вносить последовательно, не влияя на доступность приложения. Если ошибка находится в монолитном коде, придется выполнить полную процедуру внесения исправлений, тестирования и развертывания. Однако обратите внимание на то, что микросервисы при этом не затрагиваются и не требуют тестирования. Благодаря этому сокращается длительность цикла ввода в эксплуатацию новых выпусков;

- **замена компонентов приложения.** Предположим, что, так же как в главе 11, было решено воспользоваться облачными услугами для управления претензиями. В новой архитектуре нам останется только изменить конфигурационные файлы, как было описано выше в этой главе, чтобы сослаться на конечную точку в облаке;
- **замена или добавление нового стека технологий.** Если для разработки имеющейся или новой службы предпочтительнее будет использовать другой стек технологий, например PHP/NOSQL, разработчики вправе поступить именно так;
- **выборочное масштабирование.** Одно из самых больших преимуществ микросервисов – возможность выборочного масштабирования. Как можно заметить на структурной диаграмме новой архитектуры, нагрузка на каждый микросервис может балансироваться отдельно. Если ожидается увеличение трафика в микросервисе управления претензиями, можно запустить дополнительные виртуальные машины или контейнеры для этой службы, не затрагивая других служб и монолитной части приложения. Это поможет сэкономить затраты ресурсов и времени на бессмысленное масштабирование всего приложения. Мы подробнее рассмотрим эту тему в следующей главе;
- **обработка неисправностей.** Проблема или ошибка в одном микросервисе не будет влиять на работоспособность всего приложения, если спроектировать его правильно. В худшем случае неисправность скажется на каком-то конкретном микросервисе, но остальная часть системы будет функционировать как обычно. Представьте сайт электронной коммерции, реализованный в монолитной архитектуре. Допустим, в части приложения, отвечающей за прием рейтинга продукта, возникла аварийная ситуация. В зависимости от того, как написано монолитное приложение, это может привести к аварийному завершению всего приложения, даже при том условии, что в части, реализующей покупательскую корзину или оформление заказов, никаких проблем не возникало. В приложении на основе микросервисов подобная ситуация в худшем случае обрушит микросервис, в результате чего пользователь просто не сможет поставить свою оценку. Но поскольку все остальные службы продолжают нормальное функционирование, пользователи смогут совершать покупки, т. е. неисправность будет иметь весьма ограниченное влияние на бизнес.

Масштабируемость является самой большой проблемой. Когда есть всего несколько микросервисов, эта проблема решается довольно просто. Однако вообще они предназначены для создания огромных систем, состоящих из тысяч микросервисов и имеющих жесткие требования к масштабированию как вверх, так и вниз. Давайте теперь перейдем к следующей главе и посмотрим, как запускать микросервисы в контейнерах, чтобы упростить управление ими и их масштабирование.

Глава 13

Практический пример: контейнеризация приложения Helpdesk

В главе 12 «Практический пример: миграция на архитектуру микросервисов» мы создали три микросервиса, основываясь на потребностях и оценках критериев миграции, с которыми познакомились в этой книге. Теперь перед нами встает другой вопрос: как масштабировать эту модель? В реальном мире крупномасштабные приложения могут иметь сотни и тысячи микросервисов. В этой главе мы воспользуемся нашими знаниями о контейнерах Docker и организуем развертывание и масштабирование микросервисов по требованию.

Монолитная часть приложения продолжит работу в том виде, как есть, но ту часть, что была выделена в микросервисы, включая управление претензиями, каталог продуктов и поиск, мы разместим в контейнерах и внесем соответствующие изменения в монолитную часть.

Контейнеризация микросервисов

В этом разделе мы поместим в контейнер микросервис управления каталогом продуктов, созданную в главе 12. Для этого нужно воспользоваться полученными знаниями и выполнить следующие шаги:

1) написать список зависимостей для каждого микросервиса;
2) собрать двоичные файлы, WAR-файлы и т. д., составляющие микросервис;
3) создать образ Docker, включающий все элементы, выявленные и созданные ранее;
4) использовать созданный образ для запуска одного или нескольких контейнеров.

Список зависимостей

Вот список программного обеспечения (зависимостей), необходимого для запуска микросервиса управления каталогом продуктов:

- **Tomcat** – необходим для выполнения прикладного кода микросервиса;
- **Java** – требуется для выполнения Tomcat;

- **модуль связи с MySQL** – требуется для подключения Tomcat к MySQL;
- **Apache Maven** – устанавливается в системе, где производится сборка микросервиса (см. https://maven.apache.org/install.html).

Сборка двоичных и WAR-файлов

Теперь, определив программное обеспечение, необходимое для работы микросервиса управления каталогом продуктов, нужно собрать сам WAR-файл. Следуйте инструкциям ниже, чтобы собрать и получить WAR-файл микросервиса. Но прежде скопируйте код микросервиса из репозитория GitHub: https://github.com/kocherMSD/Helpdesk_Microservices.git.

Для сборки первого микросервиса воспользуемся преимуществами новейшего набора инструментов вместо Apache Ant (использовавшегося для сборки монолитного приложения). В данном случае используем Apache Maven (более мощный инструмент автоматизации сборки, который способен сам загружать все необходимые зависимости) и с его помощью создадим WAR-файл.

Следующий шаг – проверка файла Apache POM, находящегося в корневом каталоге скопированного кода. В нем перечисляются зависимости, такие как версия библиотеки Java, время выполнения, информация о центральном репозитории Maven и список требуемых JAR-файлов.

После этого следует запустить сборку WAR-файла. В терминале выполните команду `mvn install` в корневом каталоге проекта или щелкните правой кнопкой мыши на файле POM в редакторе Eclipse и выберите пункт **mvn install**. После этого в корневом каталоге должна появиться папка *Target* с готовым WAR-файлом.

Создание образа Docker

Теперь посмотрим, как создать образ Docker для службы управления каталогом продуктов. Процедура создания образов для других микросервисов, таких, например, как служба управления претензиями, будет такой же, только использоваться в ней будут другие двоичные файлы и зависимости.

Как мы узнали в предыдущих главах, чтобы получить образ Docker, нужно создать файл *Dockerfile*, перечисляющий все зависимости, упомянутые выше. В результате сборки с этим файлом *Dockerfile* мы получим образ, необходимый для развертывания микросервиса.

Начнем с файла *Dockerfile*. Обратите внимание, что мы будем править этот файл в несколько этапов, чтобы упростить объяснение его содержимого. Таким образом, в результате шагов, описанных ниже, должен получиться один файл, а не несколько:

```
# Основан на Ubuntu 17.04
FROM ubuntu:17.04

# Переменные окружения для установки Tomcat 7; вы можете изменить
# младший номер версии Tomcat, если необходимо. Если вы решите изменить старший
# номер версии (например, выбрать Tomcat 8), не забудьте изменить также
# переменную TOMCAT_LOCATION.
ENV    TOMCAT_VERSION=7.0.81
```

```
ENV    TOMCAT_FILENAME=apache-tomcat-$TOMCAT_VERSION.tar.gz
ENV    TOMCAT_DIRECTORY=apache-tomcat-$TOMCAT_VERSION
ENV    TOMCAT_LOCATION=http://www-eu.apache.org/dist/tomcat/ \
tomcat-7/v$TOMCAT_VERSION/bin/$TOMCAT_FILENAME
```

Рассмотрим подробнее некоторые из этих строк:

- `FROM ubuntu` – определяет окружение, в котором будет выполняться служба управления каталогом. В данном случае роль окружения будет играть Ubuntu;
- команды `ENV` определяют переменные окружения, которые будут использоваться далее в файле *Dockerfile*.

Следующий шаг – получить и установить все зависимости. Добавьте в файл следующие строки:

```
# Загрузить Tomcat; для установки требуются такие утилиты, как wget и JDK1.8.
# Обновить кеш apt командой "apt-get update".
RUN apt-get update && \
    apt-get install -y wget && \
    apt-get install -y default-jdk && \
    rm -fr /var/lib/apt/lists/* && \
    wget $TOMCAT_LOCATION
```

Далее описывается, что делает этот код:

- `apt-get` – это диспетчер пакетов в Ubuntu, он упрощает цикл управления пакетами (установку/обновление/удаление). Перед установкой всегда желательно выполнять `apt-get update`. Эта команда получает список последних версий пакетов из репозитория Ubuntu;
- команда `apt-get install` установит пакет `wget` – свободно распространяемую утилиту, которая используется для загрузки файлов из интернета. Она необходима для загрузки Tomcat;
- следующая команда `apt-get install` установит Java Development Kit – набор инструментов разработчика на Java, необходимый для Tomcat;
- когда запускается команда `apt-get update`, она загружает список пакетов из репозитория Ubuntu и сохраняет его в каталоге */var/lib/apt/lists*. Этот каталог должен находиться на устройстве, имеющем достаточно большую емкость, потому что образ Docker может получиться довольно большим. После установки этот каталог можно смело очистить, что и делает команда `rm`. Именно так рекомендуется организовывать установку зависимостей в файлах *Dockerfile*;
- `wget` – это утилита, установленная ранее, и здесь она выполняет загрузку Tomcat из интернета.

Обратите внимание, что все эти команды выполняются в одной строке, чтобы уменьшить число слоев в образе Docker. Команда `RUN` требует от Docker выполнить любую указанную команду в заданном окружении (в данном случае в окружении Ubuntu). Если бы в качестве окружения выбрали CentOS (например, `FROM CentOS`), тогда мы должны были бы использовать в команде `RUN` диспетчер пакетов `yum`, потому что именно он применяется в CentOS.

Оформив загрузку Tomcat, добавьте в конец файла следующие строки:

```
# Установить Tomcat в /opt и переименовать каталог установки в "tomcat"
RUN tar -xf $TOMCAT_FILENAME -C /opt && \
    mv /opt/$TOMCAT_DIRECTORY /opt/tomcat
```

Здесь выполняется установка Tomcat в каталог */opt*, и затем каталог установки переименовывается в */opt/tomcat*.

Теперь разверните микросервис:

```
# Развернуть службу каталога продуктов в Tomcat
ADD catalog-svc.war /opt/tomcat/webapps/

# Экспортировать порт в хост-систему
EXPOSE 8080

# Запустить tomcat
CMD ["/opt/tomcat/bin/catalina.sh", "run"]
```

Рассмотрим этот фрагмент поближе:

- команда `ADD` требует от Docker скопировать файл *catalog-svc.war* в каталог *webapps* сервера Tomcat, чтобы служба запускалась сразу после запуска контейнера;
- `EXPOSE` экспортирует порт сервера Tomcat в хост-систему, где будет выполняться контейнер;
- `CMD` определяет команду по умолчанию, которая должна быть выполнена сразу после запуска контейнера. Запуская Tomcat, мы гарантируем автоматический запуск Tomcat и автоматическое развертывание микросервиса управления каталогом продуктов.

Ниже для справки приводится файл целиком:

```
# Основан на Ubuntu 17.04
FROM ubuntu:17.04

# Переменные окружения для установки Tomcat 7; вы можете изменить
# младший номер версии Tomcat, если необходимо. Если вы решите изменить старший
# номер версии (например, выбрать Tomcat 8), не забудьте изменить также
# переменную TOMCAT_LOCATION.
ENV    TOMCAT_VERSION=7.0.81
ENV    TOMCAT_FILENAME=apache-tomcat-$TOMCAT_VERSION.tar.gz
ENV    TOMCAT_DIRECTORY=apache-tomcat-$TOMCAT_VERSION
ENV    TOMCAT_LOCATION=http://www-eu.apache.org/dist/tomcat/ \
tomcat-7/v$TOMCAT_VERSION/bin/$TOMCAT_FILENAME

# Загрузить Tomcat; для установки требуются такие утилиты, как wget и JDK1.8.
# Обновить кеш apt командой "apt-get update".
RUN apt-get update && \
    apt-get install -y wget && \
    apt-get install -y default-jdk && \
    rm -fr /var/lib/apt/lists/* && \
    wget $TOMCAT_LOCATION

# Установить Tomcat в /opt и переименовать каталог установки в "tomcat"
RUN tar -xf $TOMCAT_FILENAME -C /opt && \
    mv /opt/$TOMCAT_DIRECTORY /opt/tomcat

# Развернуть службу каталога продуктов в Tomcat
ADD catalog-svc.war /opt/tomcat/webapps/

# Экспортировать порт в хост-систему
EXPOSE 8080
```

```
# Запустить tomcat
CMD ["/opt/tomcat/bin/catalina.sh", "run"]
```

Теперь используем этот *Dockerfile* для сборки образа Docker со службой управления каталогом продуктов.

Сборка образа Docker

Чтобы задействовать только что созданный файл *Dockerfile*, введите следующую команду:

```
>> docker build -t catalog-svc:1.0.
```

Посмотрим, что она делает:

- `docker build` – это команда для создания образа Docker;
- `-t` – ключ, определяющий имя создаваемого образа (в данном случае `catalog-svc:1.0`), который имеет форму `ImageName:<Tag>`;
- заключительная точка (`.`) определяет каталог (в данном случае – текущий), где находятся необходимые файлы.

Чтобы выполнить эту команду, необходимо:

1) проверить, установлен ли Docker в системе, и если нет – установить;
2) создать каталог, куда поместить только что созданный файл *Dockerfile* и WAR-файл со службой управления каталогом продуктов;
3) выполнить команду `docker build` из каталога, созданного на шаге 2.

Теперь, получив образ Docker со службой управления каталогом продуктов, мы можем использовать его и запускать службу (в контейнере Docker) очень быстро. Но прежде нужно организовать инфраструктуру для работы этой службы. В предыдущих главах мы познакомились с фреймворками Mesos и Marathon, поэтому давайте используем их для развертывания наших микросервисов.

Самый быстрый способ – воспользоваться операционной системой для датацентра (Data Center Operating System, DC/OS), распределенной операционной системой с открытым исходным кодом, основанной на Apache Mesos, которая обеспечивает простой и быстрый способ получить и настроить Mesos, Marathon и Marathon-lb. Мы организуем нашу инфраструктуру внутри Amazon Web Services (AWS). За более подробной информацией о DC/OS обращайтесь по адресу: https://dcos.io.

Настройка кластера DC/OS в AWS

Итак, мы решили, что свои микросервисы будем развертывать в кластере DC/OS, поэтому сначала организуем сам кластер. Есть несколько способов запустить кластер DC/OS. Самый простой вариант – запустить его в AWS. (Для этого вы должны получить учетную запись AWS. Подробное описание можно найти на странице https://docs.mesosphere.com/1.7/administration/installing/ent/cloud/aws/.)

Работая с экземплярами EC2 в Amazon, имейте в виду, что для аутентификации компания Amazon использует ключи Secure Shell (SSH) вместо имен пользователей и паролей. В частности, она использует открытый ключ для шифрования и расшифровывания учетных данных.

Создадим пару ключей, которые понадобятся при создании кластера DC/OS:

1) в консоли AWS (см. рис. 13.1), в разделе **Network & Security** (Сеть и безопасность), щелкните на кнопке **Key Pairs** (Пары ключей);

Рис. 13.1 ❖ Внешний вид консоли AWS

2) введите имя для создаваемой пары ключей;
3) сохраните вновь созданную пару ключей в безопасном месте. Они вскоре потребуются нам, когда мы начнем создавать кластер.

Теперь выполните следующие шаги, чтобы создать кластер DC/OS:

1) запустите шаблон DC/OS по адресу: https://dcos.io/docs/1.7/administration/installing/cloud/aws. Щелкните на ссылке **Launch the DC/OS template** (Запустить шаблон DC/OS), которая указана в шаге 1, в разделе **Install DC/OS** (Установка DC/OS) на странице;
2) выберите тип кластера (с одним или несколькими ведущими узлами). Для целей тестирования достаточно одного ведущего узла. В промышленных системах предпочтительнее иметь несколько ведущих узлов, чтобы избежать появления единственной точки отказа;
3) на следующем экране, как показано на рис. 13.2, оставьте в полях значения по умолчанию и щелкните на кнопке **Next** (Далее);

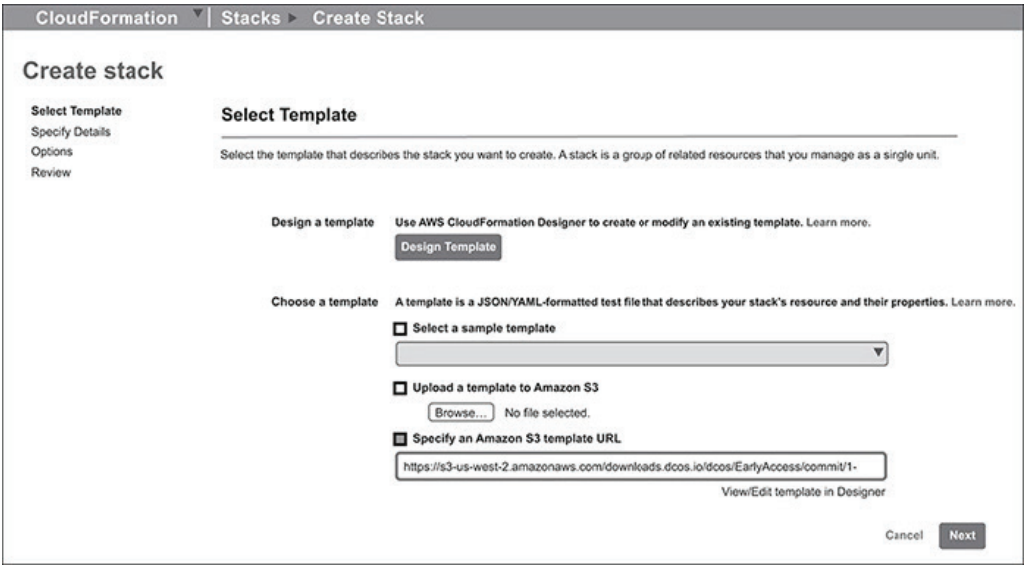


Рис. 13.2 ❖ Выбор шаблона по умолчанию

4) на странице **Create Stack** (Создание стека) выберите раздел **Specify Details** (Указать настройки) слева, как показано на рис. 13.3. Введите имя кластера и в раскрывающемся списке выберите имя пары ключей, созданной ранее;

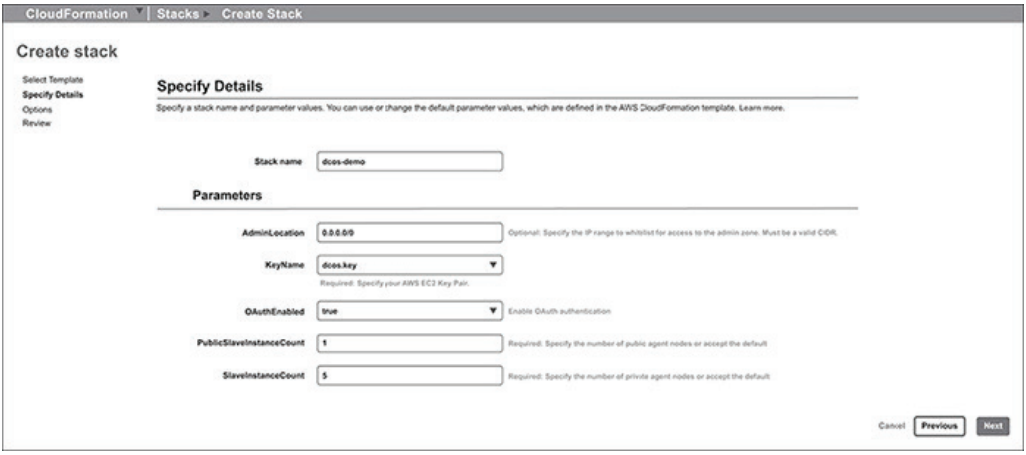


Рис. 13.3 ❖ Настройки параметров стека

5) выберите число общедоступных и приватных узлов. В данном случае оставьте значения по умолчанию;
6) оставьте значения по умолчанию в остальных полях и завершите создание стека.
   Для создания кластера DC/OS потребуется около 10–15 мин DC/OS. Состояние созданного стека можно увидеть, выбрав пункт меню **CloudFormation** → **Stacks** (CloudFormation → Стеки), как показано на рис. 13.4;

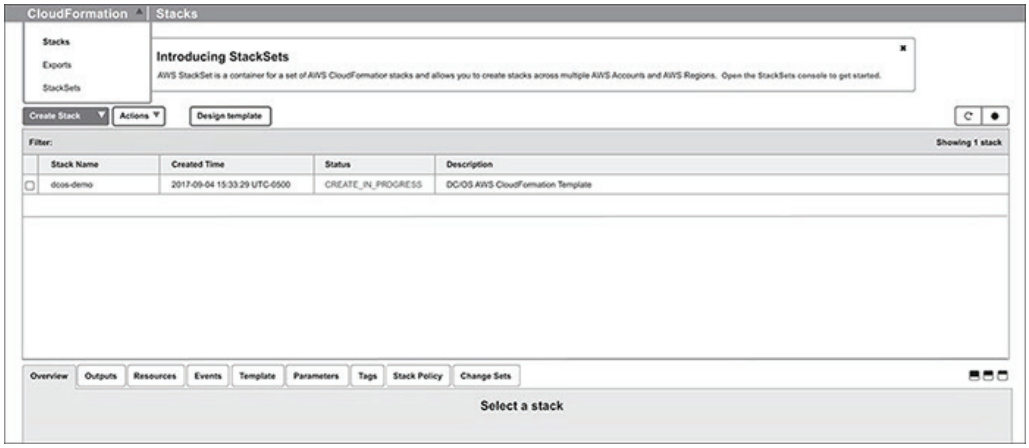


Рис. 13.4 ❖ Текущее состояние стека

7) когда создание стека завершится, перейдите на вкладку **Outputs** (Вывод) и скопируйте URL ведущего узла Mesos в адресную строку браузера. Вы должны увидеть пользовательский интерфейс DC/OS с системными дашбордами, как показано на рис. 13.5;

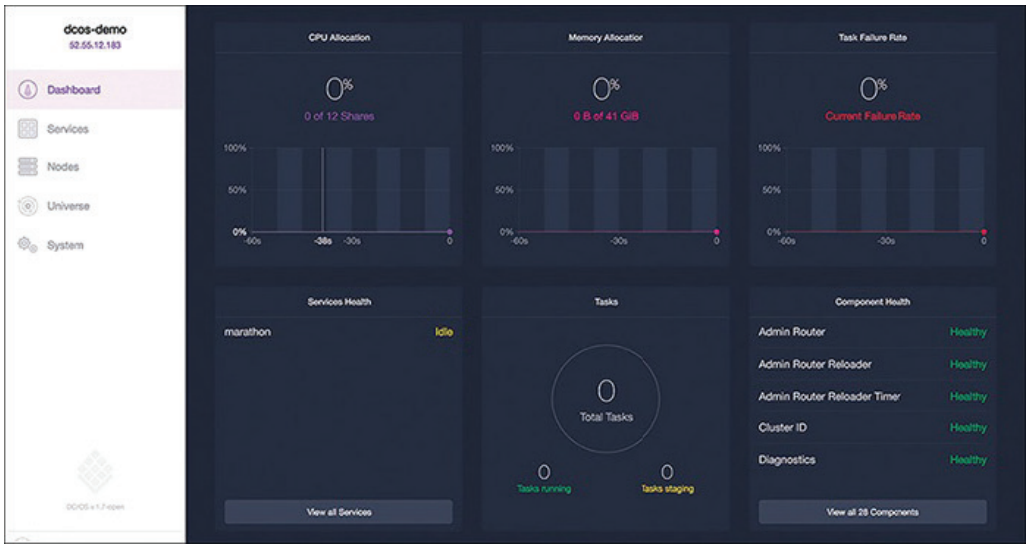


Рис. 13.5 ❖ Системные дашборды

8) на странице пользовательского интерфейса перейдите в раздел **Universe** (Общий обзор), в панели слева, и найдите запись **marathon**. Вы должны увидеть картину, как показано на рис. 13.6. Щелкните на кнопках **Install** (Установить) для Marathon и Marathon-lb.

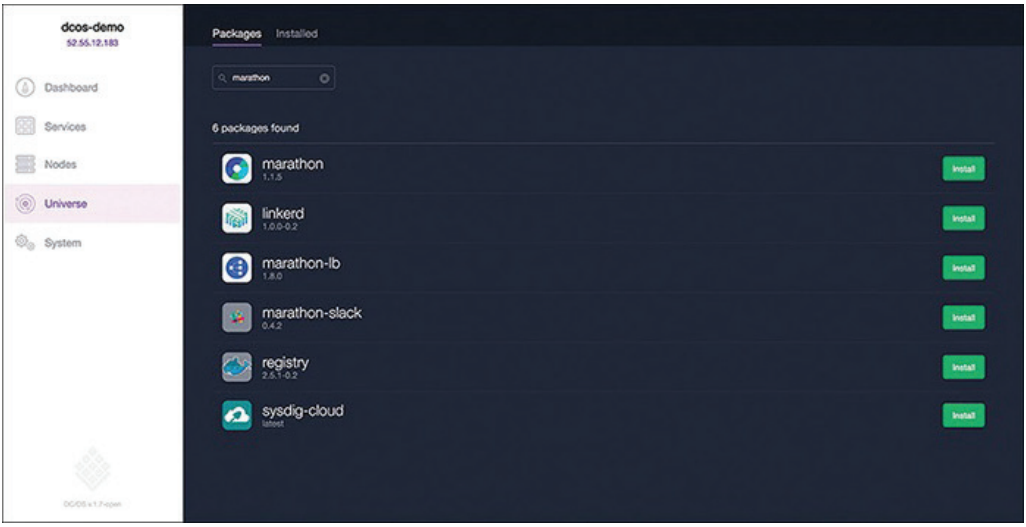


Рис. 13.6 ❖ Установка Marathon и Marathon-lb

На этом установку кластера DC/OS можно считать завершенной. Теперь взгляните на получившуюся у нас логическую структуру приложения, изображенную на рис. 13.7.

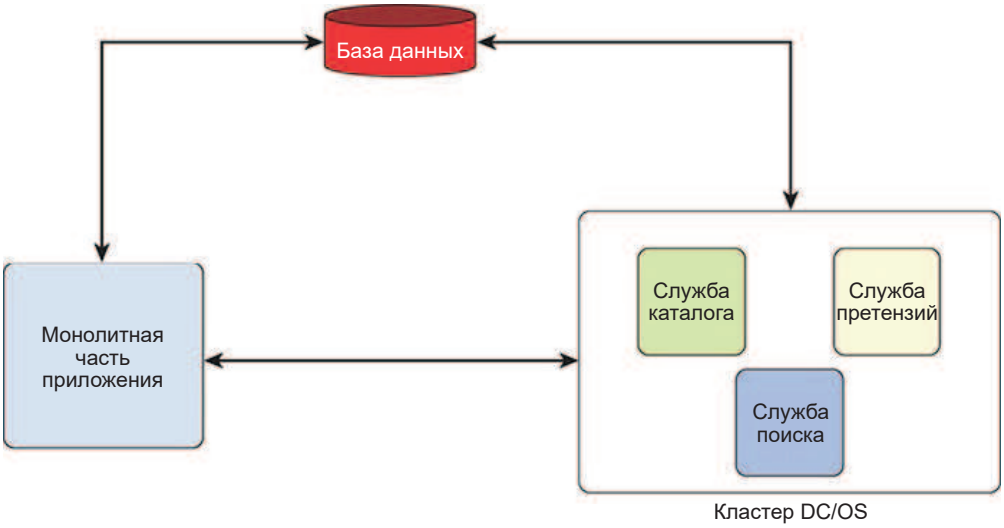


Рис. 13.7 ❖ Логическая структура приложения

Обратите внимание, что службы, выделенные нами и упакованные как микросервисы управления каталогом, претензиями и поиском, будут развернуты в кластере DC/OS, тогда как остальная часть приложения будет работать без изменений. Микросервисы продолжат использовать прежнюю базу данных.

На рис. 13.8 показана логическая организация служб в кластере DC/OS.

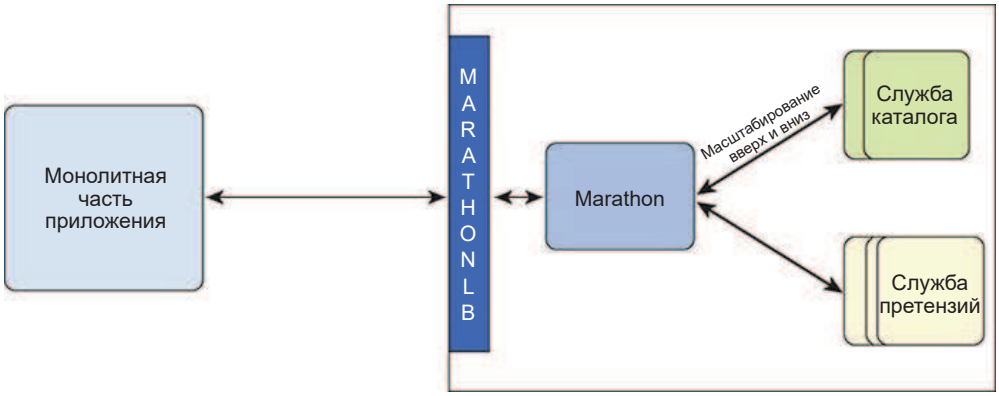


Рис. 13.8 ❖ Логическая организация служб в кластере

Теперь вспомним, что мы узнали в главе 9 «Организация контейнеров» о Mesos и Marathon, чтобы лучше понять, что у нас получилось:

- **Mesos** – это проект компании Apache, реализующий управление ресурсами (такими как вычислительная мощность и память) в кластере. Наши задания, или службы, такие, например, как управление каталогом продуктов и претензиями, будут выполняться в кластере Mesos;
- **Marathon** – еще один фреймворк с открытым исходным кодом, тесно взаимодействующий с ведущим узлом Mesos и планирующий запуск заданий в кластере. Чтобы запустить службу каталога в кластере Mesos, нужно перейти в пользовательский интерфейс Marathon, указать информацию о службе каталога продуктов (например, образ Docker, сетевой порт) и щелкнуть на кнопке **Submit** (Отправить);
- **Marathon-lb** – это балансировщик нагрузки (load balancer, lb), основанный на популярном балансировщике HAProxy. Он действует автоматически и динамически генерирует конфигурацию для HAProxy, как описано ниже:
  - взаимодействует с Marathon посредством API и получает список заданий и служб, которые Marathon запланировал для выполнения в кластере Mesos;
  - по запросу со стороны Marathon отыскивает службы, действующие в кластере, определяет узлы, где они выполняются, использующиеся порты и т. д.;
  - генерирует конфигурацию для HAProxy, которая является простым отражением запроса. Конфигурация содержит такие детали: «Если запрос поступил на адрес конечной точки `/abc`, тогда он может обрабатываться серверами `a`, `b` или `c`», при этом `a`, `b` и `c` – машины, в которых выполняется служба, обрабатывающая запрос `/abc`;
  - для обращения к службам в кластере Mesos внешние приложения всегда используют Marathon-lb.

Теперь наш кластер готов к работе, и мы можем развернуть в нем наши микросервисы. Далее мы развернем только службу каталога продуктов, а две другие службы будет необходимо развернуть самостоятельно.

Развертывание микросервиса каталога

Сначала развернем единственный экземпляр микросервиса каталога продуктов, а потом настроим его масштабирование, учитывая нагрузку.

Чтобы развернуть микросервис в кластере, определим задание с полной информацией о службе и наших потребностях. Затем мы отправим это задание в кластер с помощью Marathon.

Отправка задания в Marathon

Опишем задание для службы каталога продуктов. Отправить задание в Marathon можно двумя способами:

- непосредственно, с помощью простой команды, например `docker run -P -d nginx`. Таким способом можно посылать простые задания, не требующие сложных настроек;
- когда требуется описать задание более подробно, настройки можно поместить в файл JSON. Формат JSON – хорошо известный и стандартный формат файлов, в котором настройки описываются в простом текстовом виде, как пары ключ/значение, что будет показано чуть ниже.

Мы поместим подробное описание службы каталогов в файл JSON, а затем отправим его в Marathon. Вот как выглядит такой файл для микросервиса каталога:

```
{
  "id": "catalog-external",
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "kocher/catalog-svc:1.1",
      "network": "BRIDGE",
      "portMappings": [
        { "hostPort": 0, "containerPort": 8080, "servicePort": 10000 }
      ],
      "forcePullImage":false
    }
  },
  "instances": 1,
  "mem": 1024,
  "healthChecks": [{
    "protocol": "HTTP",
    "path": "/",
    "portIndex": 0,
    "timeoutSeconds": 20,
    "gracePeriodSeconds": 10,
    "intervalSeconds": 10,
    "maxConsecutiveFailures": 10
  }],
```

```
  "labels":{
    "HAPROXY_GROUP":"external",
    "HAPROXY_0_VHOST":"ec2-52-207-255-252.compute-1.amazonaws.com"
  }
}
```

Посмотрим, какие параметры определены в этом описании:

- `id` – идентификатор службы каталогов. Он используется для идентификации служб в кластере;
- раздел `container` описывает контейнер Docker для службы каталога продуктов и включает следующие компоненты:
  - `type` – определяет тип контейнера. По умолчанию используется тип `DOCKER`. Также поддерживается тип `MESOS`. Кроме того, в будущем фреймворк Marathon может поддерживать другие типы контейнеров;
  - `image` – определяет образ Docker для запуска в кластере;
  - `network` – определяет тип сети. Мы выбрали `BRIDGE`. Другие типы сетей перечислены в главе 8 «Поддержка сети в контейнерах»;
  - `portMappings: hostPort` – определяет экспортируемый номер порта хоста. Как можно догадаться по имени, `containerPort` определяет номер порта внутри контейнера. `servicePort` – это порт, на котором служба каталогов будет доступна через балансировщик нагрузки Marathon-lb;
  - `forcePullImage` – если этому параметру присвоить `TRUE`, тогда перед запуском задания Marathon будет извлекать последний образ из реестра Docker. По умолчанию этот параметр получает значение `FALSE`;
- `instances` – определяет, сколько экземпляров службы должно быть запущено в кластере;
- `mem` – определяет объем памяти, выделяемой для службы;
- параметры в блоке `healthChecks` сообщают фреймворку Marathon, как и через какие интервалы времени тот должен проверять работоспособность службы.

Раздел `labels` включает следующие метки:

- `HAPROXY_GROUP` – метка `external` указывает балансировщику нагрузки Marathon, что этот микросервис должен быть доступен извне. Если присвоить этой метке значение `internal`, микросервис будет доступен только внутри кластера DC/OS;
- `HAPROXY_0_VHOST` – указывает балансировщику нагрузки Marathon, что тот должен создать для службы виртуальный хост. Службы с такой меткой будут доступны через `servicePort` и дополнительно через порты 80 и 443.

Теперь перейдите в пользовательский интерфейс Marathon и отправьте файл JSON с описанием нашего первого микросервиса в кластер DC/OS. Для этого в пользовательском интерфейсе DC/OS перейдите в раздел **Services** (Службы) и щелкните на ссылке **Marathon**. В открывшемся интерфейсе Marathon щелкните на кнопке **Open Service** (Открыть службу), как показано на рис. 13.9.

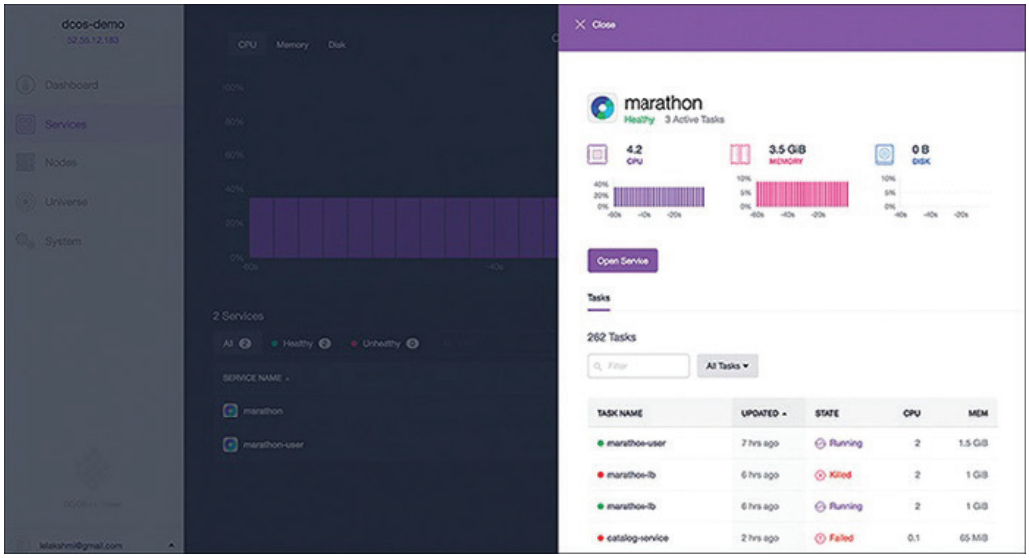


Рис. 13.9 ❖ Текущее состояние стека

Вы должны увидеть наше действующее приложение, как показано на рис. 13.10. Теперь щелкните на кнопке **Create Application** (Создать приложение). Выберите раздел **Ports and Service Discovery** (Обнаружение портов и служб) и щелкните на ссылке **JSON Mode** (Режим JSON), чтобы выгрузить наш файл *catalog.JSON*.

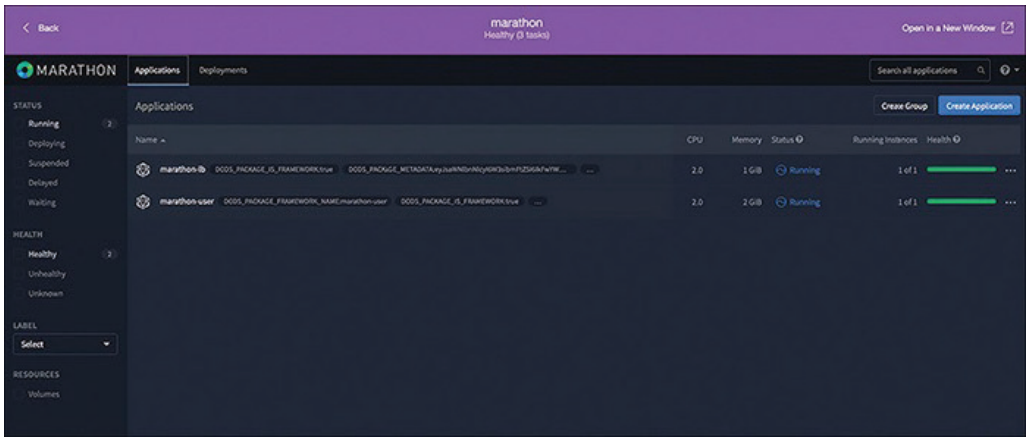


Рис. 13.10 ❖ Создание приложения

На экране должно появиться новое окно **Application** (Приложение), как показано на рис. 13.11.

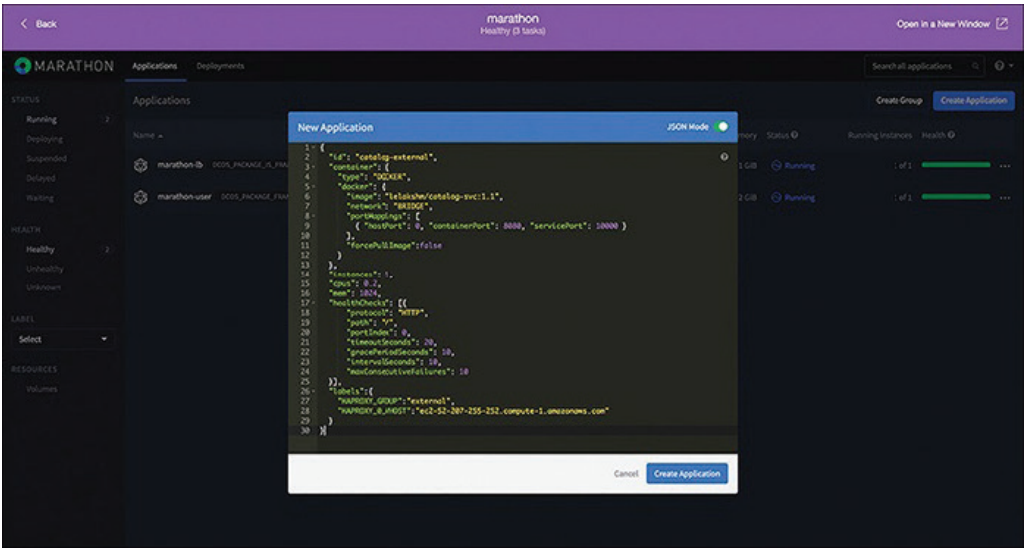


Рис. 13.11 ❖ Новое приложение

Щелкните на кнопке **Create Application** (Создать приложение), и спустя несколько секунд служба каталога продуктов должна запуститься, как показано на рис. 13.12.

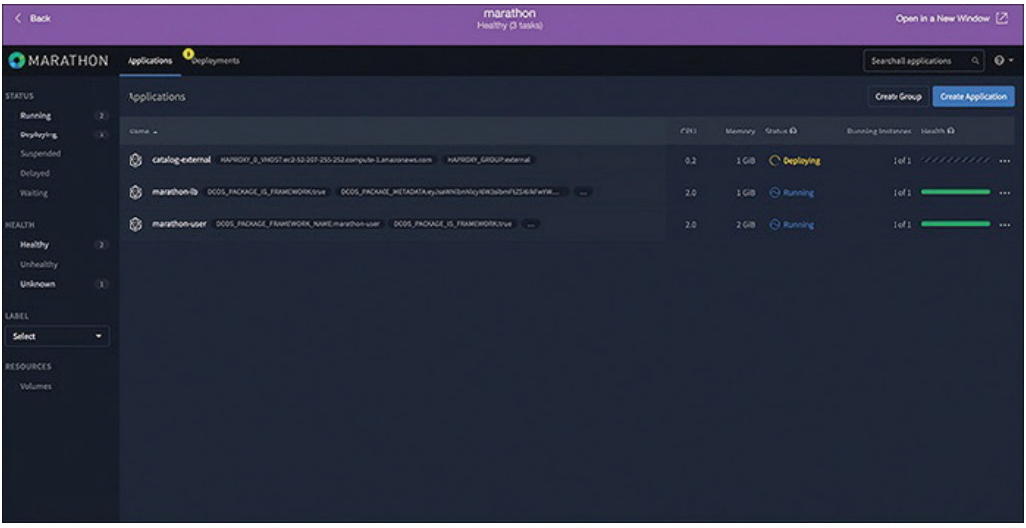


Рис. 13.12 ❖ Запущенные приложения

Проверка и масштабирование службы

Щелкнув на ссылке **catalog-external** в разделе **Applications** (Приложения), можно получить дополнительную информацию об этой службе. Как показано на рис. 13.13, в данный момент выполняется единственный экземпляр микросервиса. Здесь же можно видеть информацию о его состоянии, номер версии и дату последнего обновления.

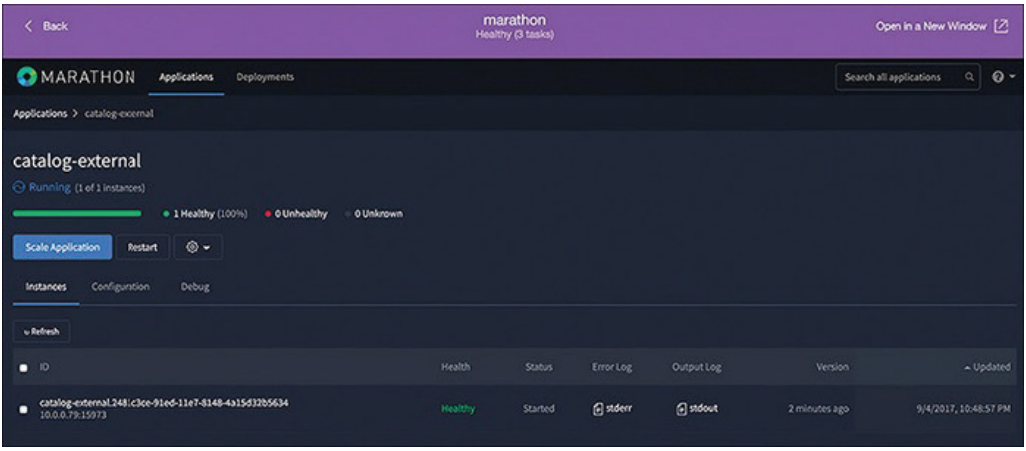


Рис. 13.13 ❖ Информация о внешней службе каталога продуктов

С помощью утилиты `curl` можно быстро получить список продуктов и убедиться, что служба действует нормально:

```
curl http://10.0.0.79:15973/catalog-svc/rest/CatalogService/getCatalog/pkocher
```

```
{
  "productFamilyListList": [
    {
      "productFamily": "Phone",
      "productId": "iPhone5",
      "technologySolution": "N"
    },
    {
      "productFamily": "Phone",
      "productId": "iPhone6",
      "technologySolution": "N"
    }
  ],
  "responseErrorCode": null,
  "responseErrorMessage": null,
  "responseStatus": "SUCCESS"
}
```

Чтобы выполнить масштабирование микросервиса, достаточно просто щелкнуть на кнопке **Scale Application** (Масштабировать приложение) и ввести желаемое число экземпляров. Допустим, мы решили запустить два экземпляра этой служ-

бы. Для этого нужно щелкнуть на кнопке **Scale Application** (Масштабировать приложение) и ввести число **2**. В течение нескольких секунд будет запущен второй экземпляр, как показано на рис. 13.14. В столбце **Running Instances** (Запущено экземпляров) вы увидите текст «2 of 2» (2 из 2), указывающий, что в данный момент в кластере выполняются два экземпляра службы каталога.

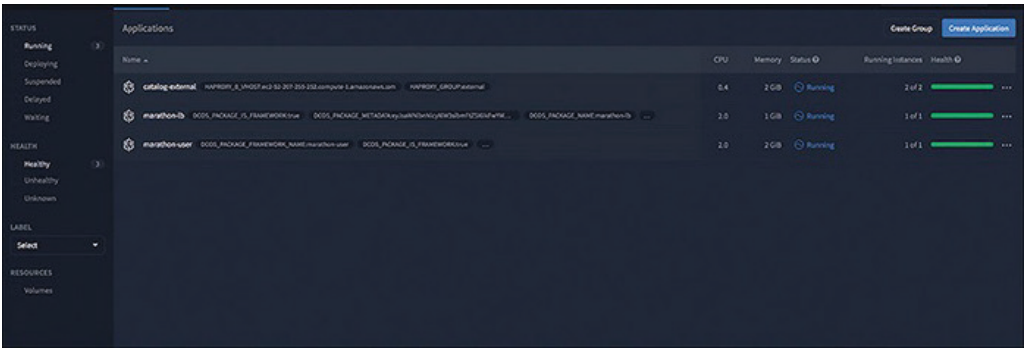


Рис. 13.14 ❖ Масштабирование вверх микросервиса

Таким способом легко масштабировать микросервисы в кластере DC/OS не только вверх, но и вниз. Теперь, когда у нас в кластере действуют два экземпляра микросервиса каталога, мы можем использовать их из монолитной части приложения.

Обращение к службе

Как узнать, где действует необходимая служба? Как рассказывалось в главе 3 «Межпроцессные взаимодействия», это один из самых сложных аспектов архитектуры микросервисов, потому что микросервисы могут запускаться и останавливаться по самым разным причинам, например из-за отказа узла или из-за нехватки ресурсов. Если микросервис остановился по какой-то причине, Marathon обнаружит это и даст команду кластеру Mesos запустить новый экземпляр. Так гарантируется одновременное выполнение в кластере требуемого количества экземпляров.

Балансировщик Marathon-lb, в свою очередь, взаимодействует с Marathon через его API, чтобы обнаружить, какие службы действуют в кластере, на каких машинах и портах и т. д. Обнаружив службу в кластере, для которой определен параметр `servicePort`, балансировщик Marathon-lb присваивает указанный порт себе и обеспечивает доступность службы через него.

В настоящий момент у нас в кластере развернуты два экземпляра службы каталога, и для нее определен параметр `servicePort` со значением 10000. Это означает, что служба каталога доступна по адресу: `http://<имя сервера Marathonlb>:10000`.

Чтобы узнать имя хоста балансировщика Marathon-lb, нужно зайти в пользовательский интерфейс кластера DC/OS, выполняющегося в AWS, выбрать стек и перейти на вкладку **Outputs** (Вывод), как показано на рис. 13.15.

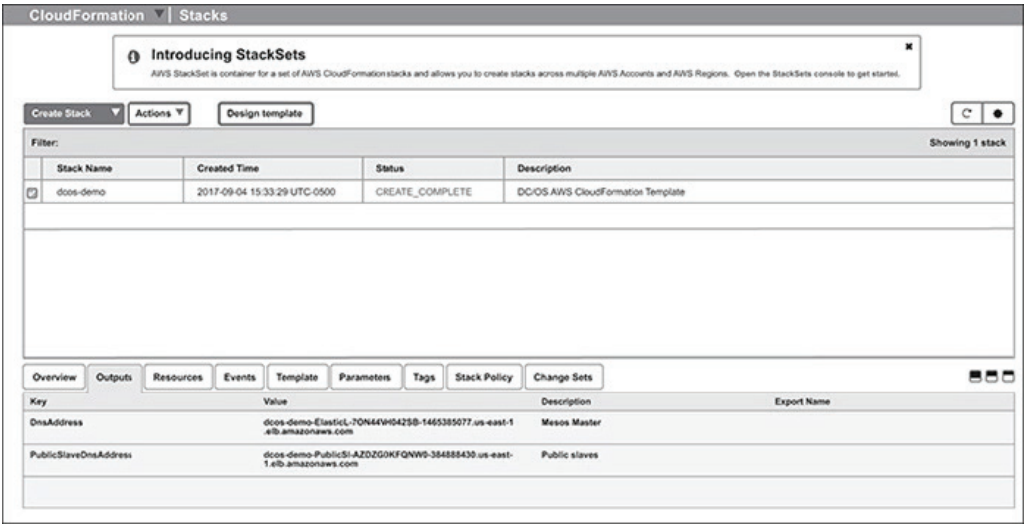


Рис. 13.15 ❖ Определение имени хоста

Искомое имя хоста (`PublicSlaveDnsAddress`), где выполняется Marathon-lb, в данном случае находится во второй строке. Таким образом, чтобы обратиться к службе каталога, запрос следует послать конечной точке:

```
curl http://dcos-demo-PublicSl-1O6EEUP9510VX-628629381.us-east-1.elb.amazonaws.com:10000/
catalog-svc/rest/CatalogService/getCatalog/<userid>
```

Обратите внимание, что в этом URL служба каталога, действующая в кластере DC/OS, представлена портом 10000. Сколько бы экземпляров службы ни было запущено, балансировщик Marathon-lb автоматически обнаружит их все и ассоциирует с портом 10000.

Теперь, когда микросервис каталога запущен и работает в кластере DC/OS, нужно настроить приложение Helpdesk, чтобы оно использовало этот микросервис. Для этого достаточно изменить настройки в файле свойств.

Изменение монолитного приложения

Наше приложение Helpdesk поддерживает список адресов URL всех служб в файле *application.properties*, который находится в каталоге */usr/share/tomcat7/lib*.

Вы должны изменить свойство `endPoints.getCatalog`, присвоив ему URL балансировщика Marathon-lb, как показано ниже:

```
endPoints.getCatalog=http://ec2-52-207-255-252.compute-1.amazonaws.com:10000/catalog-svc/rest/
CatalogService/getCatalog
```

После этого приложение Helpdesk автоматически начнет пользоваться микросервисом. Как мы уже знаем, вовсе не важно, сколько экземпляров службы каталога будет развернуто, ведь конечная точка останется прежней. Балансировщик Marathon-lb сам определит адреса экземпляров службы и автоматически выберет тот или иной экземпляр, стараясь равномерно распределить нагрузку между

ними.

В этой главе мы детально рассмотрели службу каталога продуктов: от ее сборки в виде микросервиса до развертывания в кластере DC/OS и настройки монолитной части приложения для ее использования. Службы управления претензиями и поиска настраиваются аналогично, поэтому я оставляю их развертывание вам в качестве самостоятельного упражнения. Весь необходимый код с инструкциями вы найдете в репозитории GitHub.

Мы рассмотрели и удовлетворили все потребности, о которых говорилось в главе 12, а также увидели, как можно масштабировать наше приложение в будущем. Сочетание микросервисов и контейнеров помогло нам избавиться от многих проблем.

Заключение

В предисловии я отмечал, что данная книга адресована двум группам читателей: опытным прикладным и системным программистам, желающим засучить рукава и опробовать некоторые примеры из реальной жизни, и руководителям (т. е. не программистам), желающим получить общее представление об обсуждаемой теме. Независимо от того, к какой группе относитесь вы (возможно даже, что вы относитесь сразу к обеим группам), я надеюсь, что эта книга оказалась для вас полезной и поучительной.

Каждая из тем, которые мы рассмотрели (обнаружение служб, шлюзы API, Kubernetes, взаимодействие между службами и др.) достойна отдельного тома. (На самом деле некоторым из этих тем посвящено сразу несколько отдельных книг!) В этой книге я стремился обобщить все эти темы и представить их на более высоком уровне, чтобы вы могли сделать выводы для себя и своей карьеры. Микросервисы позволяют организовать масштабирование программных компонентов по мере необходимости, а контейнеры помогают справиться со сложностями виртуализации, предлагая простое и легковесное решение. Они прекрасно дополняют друг друга.

Что такое DevOps?

Во вводных главах мы исследовали возможное влияние внедрения микросервисов и контейнеров на организации, но почти не затронули другой важный аспект их влияния – внедрение модели DevOps. В настоящее время многие компании, занимающиеся разработкой программного обеспечения, переходят на использование модели DevOps, и в этом им очень помогает использование контейнеров и архитектуры микросервисов.

DevOps – это составной термин, объединяющий две области работы с программным обеспечением: разработку (*dev*elopment) и эксплуатацию (*op*erations). Основной упор в этой модели делается на тесное сотрудничество разработчиков и эксплуатационников со следующими целями:

- ускорить выпуск новых версий программного обеспечения;
- повысить качество продукта;
- автоматизировать задачи, такие как сборка кода, тестирование, упаковка, выпуск и развертывание.

Неудивительно, что вся индустрия стремится использовать модель DevOps, чтобы воспользоваться ее потенциальными преимуществами. Так в чем же загвоздка? Какие проблемы мешают завести курицу, несущую золотые яйца? Самая большая из них – необходимость перемен. Для разработки, тестирования и выпуска программного обеспечения в этих организациях используется множество инструментов, которые придется поменять с переходом на новую парадигму. Кроме того, изменения могут затронуть организационную структуру существующих коллективов, что, в свою очередь, может потребовать обучения работников новым навыкам. Эти сложности очень похожи на те, что окружают микросервисы.

Далее, если взглянуть на цели DevOps, можно заметить, что микросервисы точно соответствуют им. В этой архитектуре монолитное приложение разбивается на управляемые фрагменты, каждый из которых реализует только одну функцию. Эти фрагменты можно распределить между несколькими коллективами так, чтобы каждый коллектив сосредоточился на разработке и эксплуатации только одного фрагмента. В результате это сокращает и упрощает цикл разработки, ускоряет развертывание, уменьшает время вывода новых версий на рынок. Эти преимущества, в свою очередь, порождают необходимость более гибкого подхода к эксплуатации и применения автоматизации. Микросервисы нуждаются в такой гибкой культуре и, следовательно, стимулируют переход на модель DevOps.

Учитывая перечисленные преимущества DevOps, может сложиться впечатление, что все разработчики, архитекторы или организации захотят быстрее перейти на парадигму микросервисов. Однако, как уже отмечалось выше, это не всегда так. Микросервисы лучше всего подходят для сложных архитектур, т. е. для программного обеспечения со множеством функций и конечных пользователей, когда необходимы быстрое развертывание и масштабируемость. В ближайшее время большинство компаний, включая мелкие и средние, последует этой тенденции. Почему? Есть пять основных причин:

- **неизбежное усложнение программного обеспечения.** Сети и хранилища, организованные на программном уровне, предоставление программного обеспечения как услуг, платформы для организации сложных взаимодействий между миллионами пользователей и устройствами – вот лишь несколько примеров, которые приходят на ум, когда речь заходит о перспективах программной индустрии. Существует большое количество компаний, которые входят в эти области. По мере движения вперед они неизбежно осознают необходимость внедрения архитектуры микросервисов и более гибкой культуры;
- **новые типы клиентов порождают новые потребности.** Многие инновационные компании разрабатывают решения, поддерживающие все разновидности новых устройств. Каждое семейство устройств имеет разный объем ресурсов. Память, быстродействие и объем хранилища ограничены на одних устройствах и в изобилии присутствуют на других. Когда все эти устройства с разными ограничениями пытаются обратиться к одному и тому же программному обеспечению, оно должно поддерживать их запросы, скрывая сложности от клиентов. Где будут скрываться эти сложности? В самом программном обеспечении! Это означает неизбежное усложнение программного обеспечения и еще более острую потребность в использовании архитектуры микросервисов, поддерживающих взаимодействие с разными клиентами;
- **рост сложности с увеличением числа пользователей.** Предложения от Amazon и Netflix становятся все сложнее благодаря инновациям в направлении упрощения и увеличения удобства для пользователей. Возможно, они смогли бы продолжать использовать монолитную парадигму при сохранении числа пользователей в разумных пределах. Действительно, в первые годы они продолжали идти по этому пути. Но, по мере того как развивающиеся рынки будут догонять развитые, а к сети будут подключаться миллионы (и миллиарды) новых пользователей, программное обеспечение

будет становиться все сложнее в связи с необходимостью удовлетворения требований к масштабируемости, производительности и различных потребностей разных пользователей. Это вынудит все большее число компаний переходить к использованию микросервисов, способных решить все эти проблемы;

- **удовлетворенность работой.** Монолитность подразумевает совместную работу над одной платформой нескольких команд, иногда разрабатывающих разные ее элементы (например, пользовательский интерфейс и серверную часть). Одна из проблем этой модели состоит в том, что за разработку всех служб на стороне сервера, таких как учет платежей, каталог продуктов, покупательская корзина и т. д., может отвечать одна команда инженеров. Когда код становится сложнее, команда начинает делиться на группы, отвечающие за разные компоненты серверной части. Если сложность кода продолжает возрастать, добавляются новые люди и создаются новые команды. Наконец, сложность достигает такого уровня, когда ввод в эксплуатацию даже незначительных изменений и исправлений начинает занимать слишком много времени. В какой-то момент энтузиазм разработчиков начинает угасать. Ошибки при сборке, откаты к предыдущим версиям и утомительная отладка могут из случайных препятствий превратиться в норму. Возрастает недопонимание, и ухудшается взаимодействие между командами. В особенно сложных ситуациях это может привести к конфликтам и даже оттоку талантливых кадров. При правильной реализации методология DevOps и парадигма микросервисов могут способствовать более четкому разделению ролей и сфер ответственности, что, в свою очередь, улучшит взаимопонимание и укрепит сотрудничество между командами. Взаимопонимание повысит производительность труда и положительно скажется на конечном результате. В итоге вы получите удовлетворенность выполняемой работой;
- **преимущества для бизнеса.** Умный бизнес всегда стремится принять на вооружение новые технологии и парадигмы, если они способны улучшить результат и избавить от проблем. Микросервисы как раз предлагают бизнесу одну из таких возможностей – возможность опередить своих конкурентов.

Учитывая перечисленные причины, в ближайшие несколько лет можно ожидать резкого увеличения скорости проникновения микросервисов и контейнеров. Впрочем, время покажет.

Это только начало

Итак, вы дочитали книгу до конца, однако я надеюсь, что эти слова послужат для вас вступлением, а не заключением. Уверяю вас, даже учитывая, что вы только что «окончили» начальный курс изучения микросервисов и Docker, вам многое еще предстоит познать. Надеюсь, этой книгой я смог вызвать у вас интерес! Я призываю вас не останавливаться на достигнутом, продолжать читать другие книги, влиться в ряды сообществ разработчиков микросервисов и пользователей контейнеров и внимательно изучить новые примеры их применения.

Приложение А

Принцип работы приложения Helpdesk

Здесь описывается принцип работы приложения Helpdesk. Считайте его своеобразным руководством пользователя, описывающим возможности, которые предлагаются приложением администратору и клиенту.

В реальном мире большинство приложений технической поддержки интегрируется с другими системами управления заказами и клиентами. Поэтому движение данных во многом осуществляется автоматически. Например, когда в системе управления клиентами создается новая учетная запись, информация о клиенте автоматически попадает в другие системы, в том числе и в приложение технической поддержки. Однако в данном случае мы рассматриваем наше приложение в отрыве от других систем, поэтому вручную добавим все необходимые данные, чтобы объяснить основы его работы.

Вот три основных типа пользователей, поддерживаемых в этом приложении:

- **администратор** – так называемый суперпользователь, который может добавлять и править учетные записи, службы и т. д. Он видит все данные и имеет доступ ко всем внутренним подсистемам, таким как базы данных;
- **клиенты** – пользователи, приобретающие продукты или услуги. Они могут создавать претензии, править их и изменять их состояние. Также клиенты могут видеть отправленные ими претензии;
- **инженер службы поддержки** – пользователь, принимающий и обрабатывающий претензии клиентов. Он может видеть все претензии.

Порядок работы администратора

В этом разделе перечисляются все функции, доступные администратору и связанные с управлением приложением.

Вход

Всякое приложение должно осуществлять аутентификацию, чтобы управлять и пользоваться им могли только уполномоченные пользователи. Для аутентификации приложение принимает имя пользователя и пароль, как показано на рис. А.1. Имена пользователей и пароли хранятся в базе данных. Приложение уже

поддерживает имя пользователя *admin* с паролем *admin*. Вы можете изменить их непосредственно в базе данных.

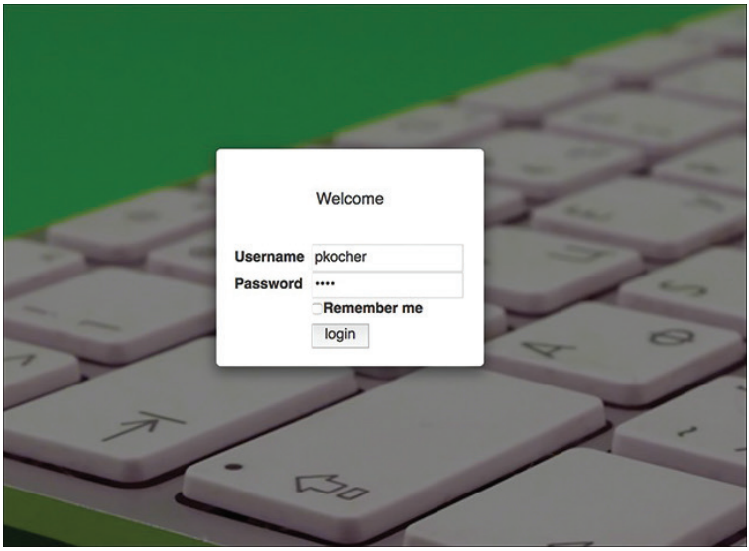


Рис. А.1 ❖ Форма входа в приложение

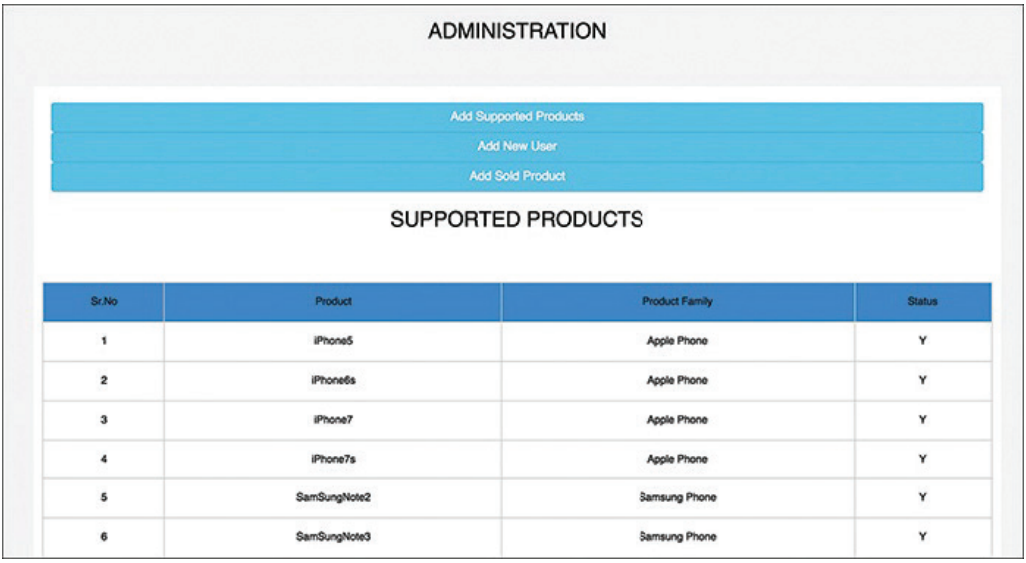
После успешной аутентификации администратор попадает на главную страницу администратора. На этой странице в верхней полосе меню перечислены все доступные модули, как показано на рис. А.2. Давайте рассмотрим их.

Рис. А.2 ❖ Возможности, предлагаемые приложением администратору

Администрирование и поддерживаемые продукты

Администратор может добавлять новых пользователей (клиентов и инженеров службы поддержки) и новые продукты, а также продавать пользователям продукты, перечисленные в каталоге, как показано на рис. А.3.

Новые продукты выпускаются через регулярные интервалы времени, а старые снимаются с поддержки и перестают обслуживаться. Здесь администратор может добавлять новые поддерживаемые продукты или удалять старые, срок поддержки которых уже истек. Например, на рис. А.3 показан список поддерживаемых продуктов, где **Y** означает «Yes» (да, поддерживается), а **N** – «No» (нет, не поддерживается).



Sr.No	Product	Product Family	Status
1	iPhone5	Apple Phone	Y
2	iPhone6s	Apple Phone	Y
3	iPhone7	Apple Phone	Y
4	iPhone7s	Apple Phone	Y
5	SamSungNote2	Samsung Phone	Y
6	SamSungNote3	Samsung Phone	Y

Рис. А.3 ❖ Список поддерживаемых продуктов в панели управления администратора

Добавление нового поддерживаемого продукта

На этой странице администратор может добавить в каталог новый поддерживаемый продукт, как показано на рис. А.4.

Рис. А.4 ❖ Добавление в каталог нового поддерживаемого продукта

Добавление нового пользователя

Как отмечалось выше, подобные действия обычно выполняются автоматически, а данные поступают из вышестоящих систем. Однако для лучшего понимания добавим некоторые данные вручную. В частности, добавим нового пользователя (клиента), которому будет разрешено добавлять новые претензии. Например, создадим для клиента с именем Bob Black учетную запись с именем пользователя `Bblack`, как показано на рис. А.5.

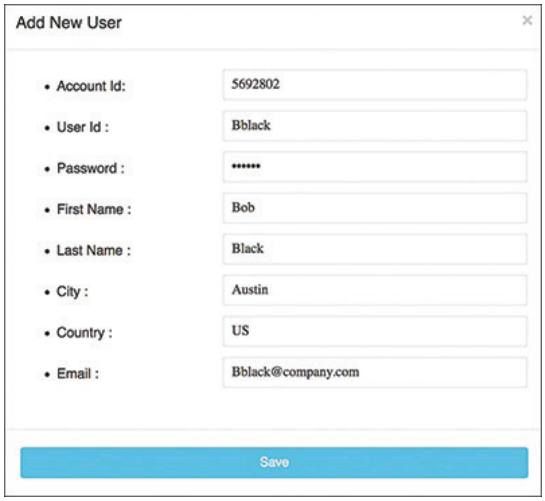


Рис. А.5 ❖ Пример ввода информации при создании учетной записи

Простое создание учетной записи не означает, что Bob сможет добавлять новые претензии. Для этого нужно связать его с приобретенным продуктом. Давайте создадим новую запись в **Add Sold Product** (Добавить проданный продукт).

Добавление проданного продукта

Администратор может вручную добавлять купленные продукты в учетную запись клиента. Допустим, наш клиент купил iPhone 7, как показано на рис. А.6.

Теперь, познакомившись с порядком работы администратора, перейдем к точке зрения клиента.

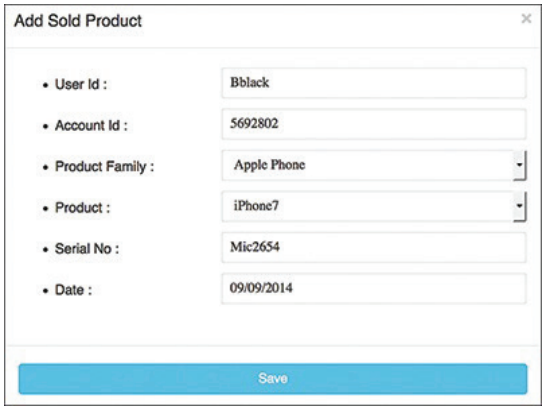


Рис. А.6 ❖ Наш пользователь приобрел iPhone

Порядок работы клиента

После успешного входа с использованием учетных данных, введенных администратором, клиент попадает на главную страницу клиента. Здесь в верхней полосе меню перечислены все доступные ему функции, как показано на рис. А.7.

Рис. А.7 ❖ Функции приложения, доступные клиенту

Рассмотрим их по очереди.

Мои продукты

Щелчок на ссылке **My Products** (Мои продукты) открывает консоль с заголовком «Product Catalog Service» (Служба каталога продуктов). Она используется для просмотра списка поддерживаемых продуктов, перечисленных в учетной записи зарегистрировавшегося пользователя. В данном примере Bob Black имеет право на поддержку iPhone 6 и iPhone 7 (последнюю запись мы только что создали, когда рассматривали порядок работы администратора), как показано на рис. А.8.

Sr.No	Product	Product Family	Support Available
1	iPhone6	Apple Phone	Y
2	iPhone7	Apple Phone	Y

Рис. А.8 ❖ Консоль **My Products** (Мои продукты), также известная как каталог продуктов

Создание претензии

Представим, что этот клиент решил создать претензию в отношении недавно купленного мобильного устройства (iPhone 5). Используя идентификатор пользователя, приложение просмотрит список приобретенных продуктов и разрешит ему создать претензию. Для этого клиенту будет представлена форма, изображенная на рис. А.9. Пользователь должен описать проблему, заполнив обязательные поля, такие как название, серия, модель телефона и категория проблемы, а затем отправить претензию.

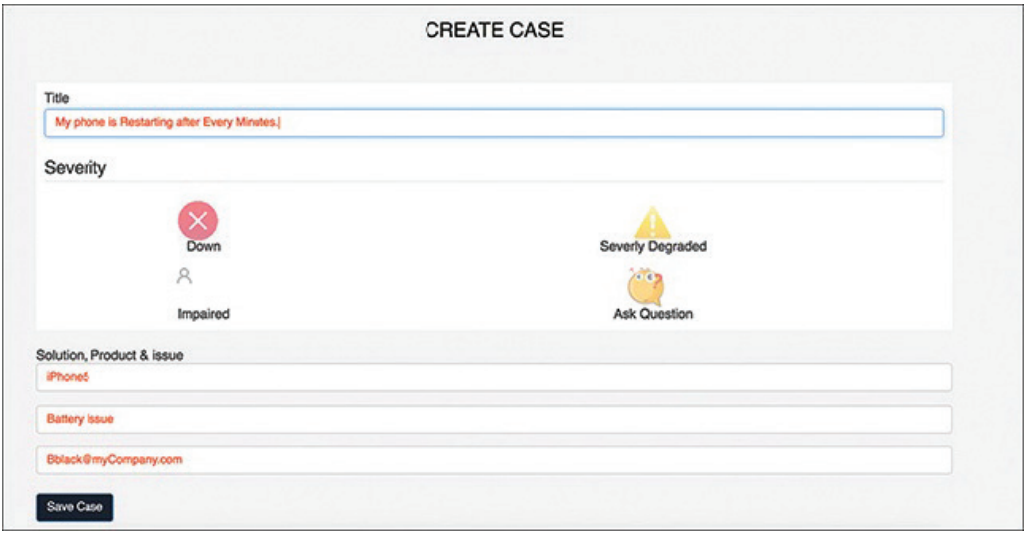


Рис. А.9 ❖ Отправка претензии для продукта

Просмотр претензий

В этой консоли пользователь может видеть список всех претензий, отправленных им ранее, как показано на рис. А.10. Когда инженер службы поддержки обработает претензию, пользователь сможет увидеть результат, щелкнув на номере претензии, как показано на рис. А.11.

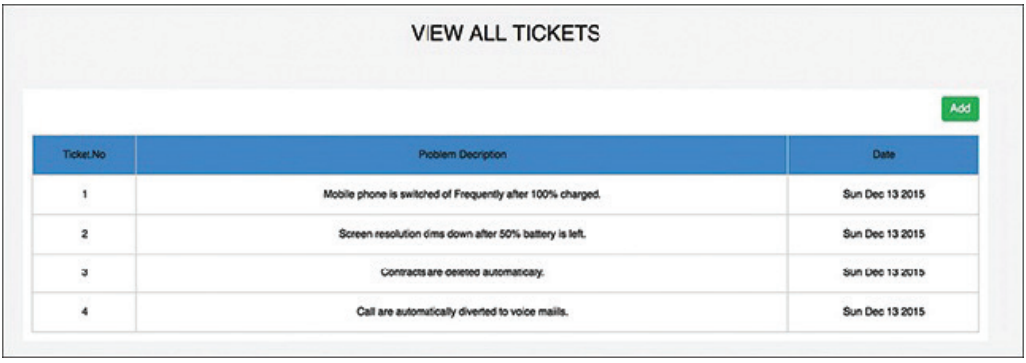


Рис. А.10 ❖ Пользователь может просматривать список своих претензий

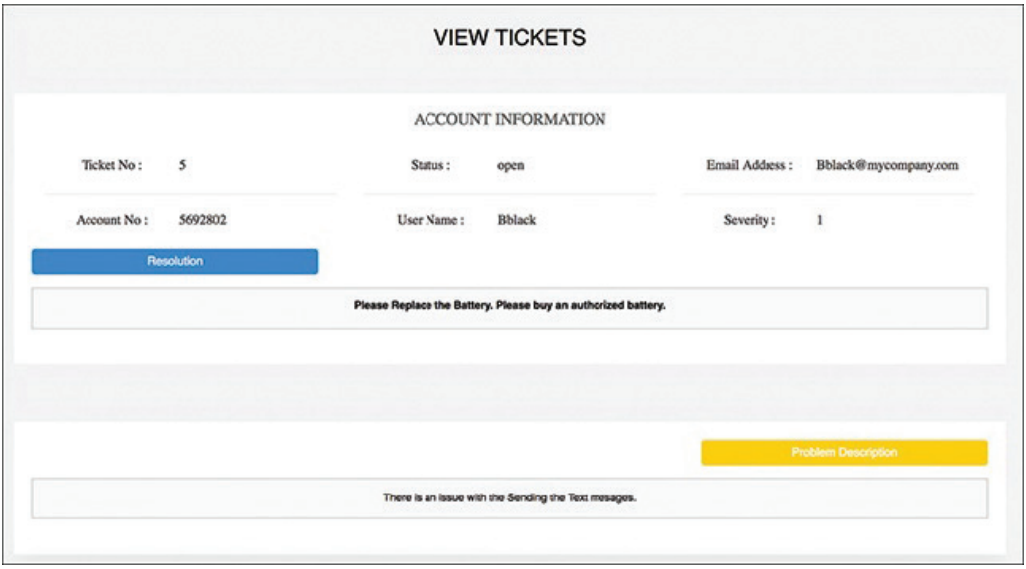


Рис. А.11 ❖ Просмотр результатов обработки претензии

Доска объявлений

Приложение поддерживает очень упрощенную доску объявлений. Благодаря ей пользователи могут оказывать помощь друг другу. Они могут размещать вопросы и оставлять ответы.

В консоли с доской объявлений, изображенной на рис. А.12, выводится полный список сообщений, открытых в настоящий момент.

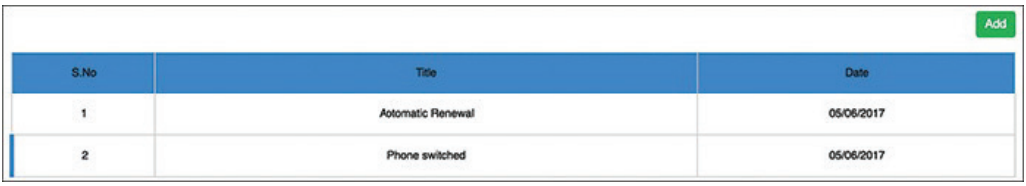


Рис. А.12 ❖ Доска объявлений

Давайте посмотрим, как работать с этой доской объявлений.

Новое сообщение

Щелчок на кнопке **Add** (Добавить) откроет консоль для ввода нового сообщения, открывающего дискуссию, как показано на рис. А.13.

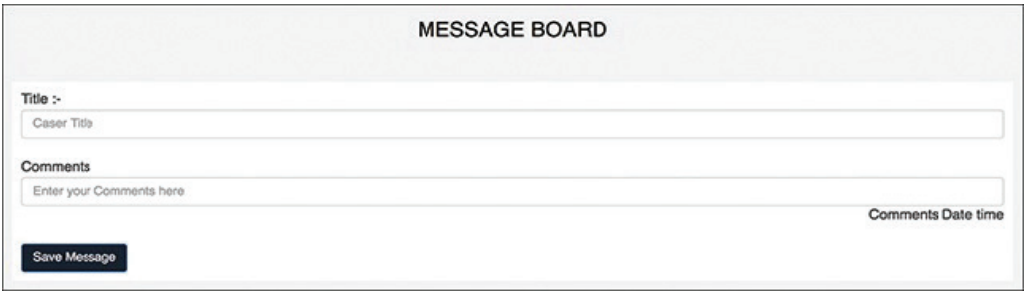


Рис. А.13 ❖ Добавление нового сообщения

Добавление ответа

Щелкнув на заголовке сообщения, как показано на рис. А.14, можно присоединиться к текущему обсуждению и оставить свой комментарий.

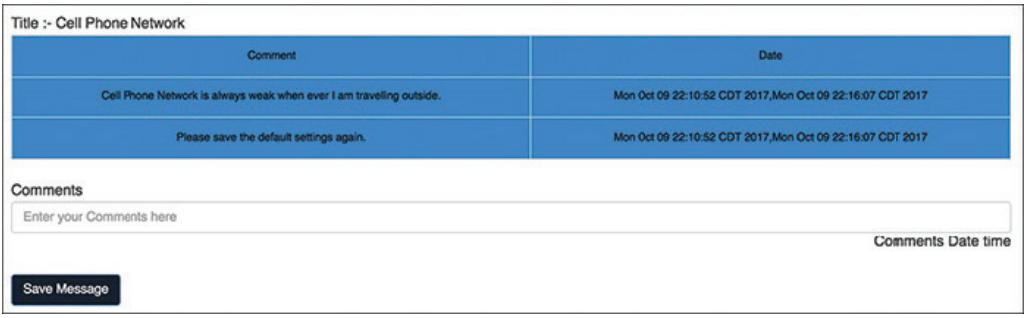


Рис. А.14 ❖ Комментарий к существующему сообщению

Запись на консультацию

В этой консоли пользователи могут записаться на очную консультацию к специалисту службы поддержки, как показано на рис. А.15. Доступные даты и время извлекаются из базы данных. Пользователь может выбрать часовой пояс, дату и время для консультации.

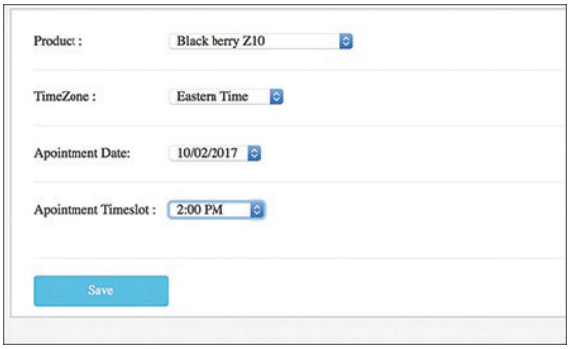


Рис. А.15 ❖ Запись на консультацию

Поиск

Пользователи имеют возможность выполнять поиск решений своих проблем. Приложение поддерживает три варианта поиска, как показано на рис. A.16:

- **Basic Search** (Простой поиск) – выполняет поиск в базе данных по ключевым словам;
- **Wiki Search** (Поиск в вики) – выполняет поиск в вики-данных по ключевым словам;
- **Advance Search** (Улучшенный поиск) – выполняет семантический поиск с помощью механизма Solr.

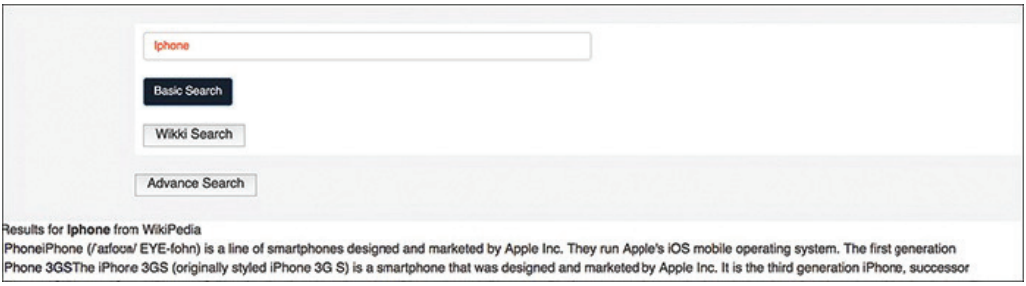


Рис. A.16 ❖ Поиск решений в приложении

Мой профиль

На странице профиля пользователя имеются две вкладки. На первой отображается персональная информация, как показано на рис. A.17, а на второй – учетная информация, как показано на рис. A.18.

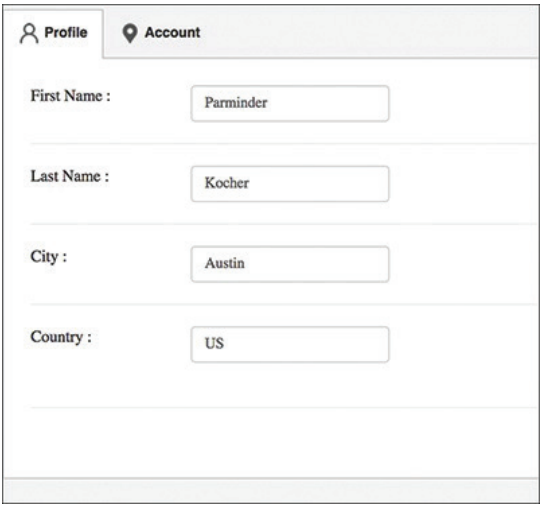


Рис. A.17 ❖ Профиль пользователя

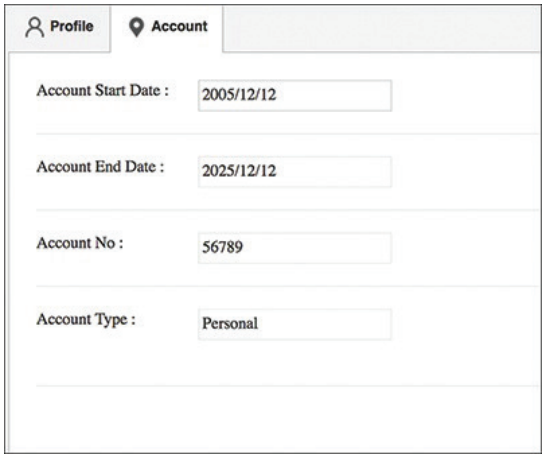


Рис. А.18 ❖ Учетная информация

Порядок работы инженера службы поддержки

Задача инженера службы поддержки состоит в том, чтобы обрабатывать претензии и помогать клиентам решать их проблемы. Инженеру службы поддержки доступны две основные функции: просмотр всех претензий и обработка конкретной претензии.

Просмотр всех претензий

В консоли инженера службы поддержки специалисты могут видеть список всех претензий и выбирать для обработки нужные, щелкая на их номерах, как показано на рис. А.19.

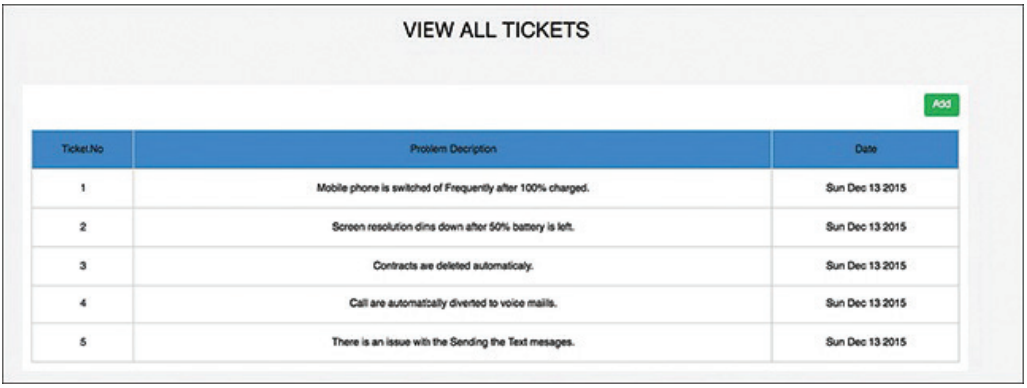


Рис. А.19 ❖ Консоль инженера службы поддержки

Обзор конкретной претензии

После щелчка на номере претензии в консоли инженера службы поддержки открывается форма с содержимым претензии. В этой форме инженер может оставить свой комментарий, изменить статус претензии и т. д., как показано на рис. А.20.

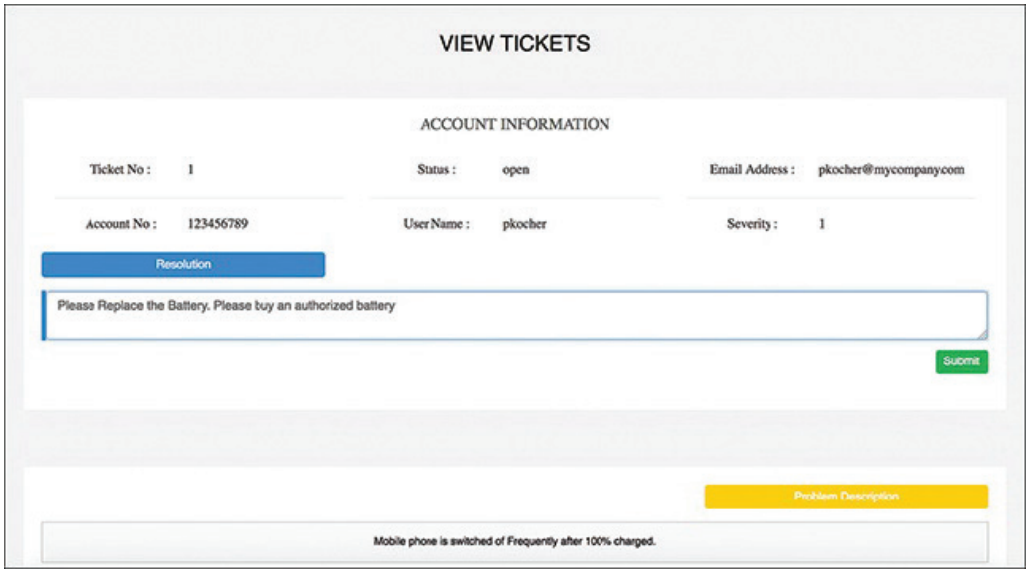


Рис. А.20 ❖ Просмотр и изменение конкретной претензии

На этом мы завершаем общее знакомство с приложением. Напомню, что наша цель состоит не в том, чтобы написать приложение промышленного уровня, а в том, чтобы получить пример для всестороннего исследования в рамках обсуждаемой темы перехода от монолитной архитектуры к архитектуре микросервисов с использованием контейнеров.

Приложение B

Установка механизма поиска Solr

В этом приложении даются пошаговые инструкции по установке и настройке механизма Solr с целью его использования для совершенствования службы поиска в нашем примере приложения, как описывалось в главе 12 «Практический пример: миграция на архитектуру микросервисов». Инструкции писались для дистрибутива Linux CentOS. Желающим узнать больше о Solr советую посетить страницу проекта: http://lucene.apache.org/solr/resources.html.

Требования

- Компьютер или виртуальная машина с установленным дистрибутивом Linux CentOS, имеющая не менее 1 Гбайт ОЗУ.
- Установленный пакет `python-software-properties`.
- Установленная последняя версия Java.

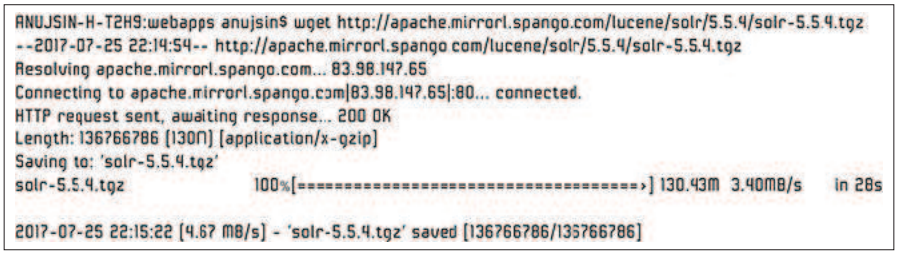
Установка

Установка подразумевает выполнение следующих шагов:

1) загрузите дистрибутив Solr с зеркала. Можно взять самую последнюю версию. Когда писалась эта инструкция, я использовал версию 5.5.

   ```
   wget http://apache.mirror1.spango.com/lucene/solr/5.5.4/solr-5.5.4.tgz
   ```

   Эта команда вызывает утилиту `wget` для загрузки файла дистрибутива, как показано на рис. B.1:



```
ANUJSIN-H-T2H9:webapps anujsin$ wget http://apache.mirror1.spango.com/lucene/solr/5.5.4/solr-5.5.4.tgz
--2017-07-25 22:14:54-- http://apache.mirror1.spango.com/lucene/solr/5.5.4/solr-5.5.4.tgz
Resolving apache.mirror1.spango.com... 83.98.147.65
Connecting to apache.mirror1.spango.com|83.98.147.65|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 136766786 (130M) [application/x-gzip]
Saving to: 'solr-5.5.4.tgz'
solr-5.5.4.tgz          100%[===================================>] 130.43M  3.40MB/s    in 28s

2017-07-25 22:15:22 (4.67 MB/s) - 'solr-5.5.4.tgz' saved [136766786/136766786]
```

Рис. B.1 ❖ Загрузка файла дистрибутива Solr

2) распакуйте архив:

```
tar xzf solr-5.5.4.tgz;
```

3) запустите сценарий установки:

```
solr-5.5.4/bin/install_solr_service.sh
```

Для завершения установки может потребоваться 1 мин. После установки вы сможете открыть в браузере страницу `http://IP_адрес_вашего_сервера:8983/solr`. На рис. B.2 показано, как выглядит веб-интерфейс Solr.

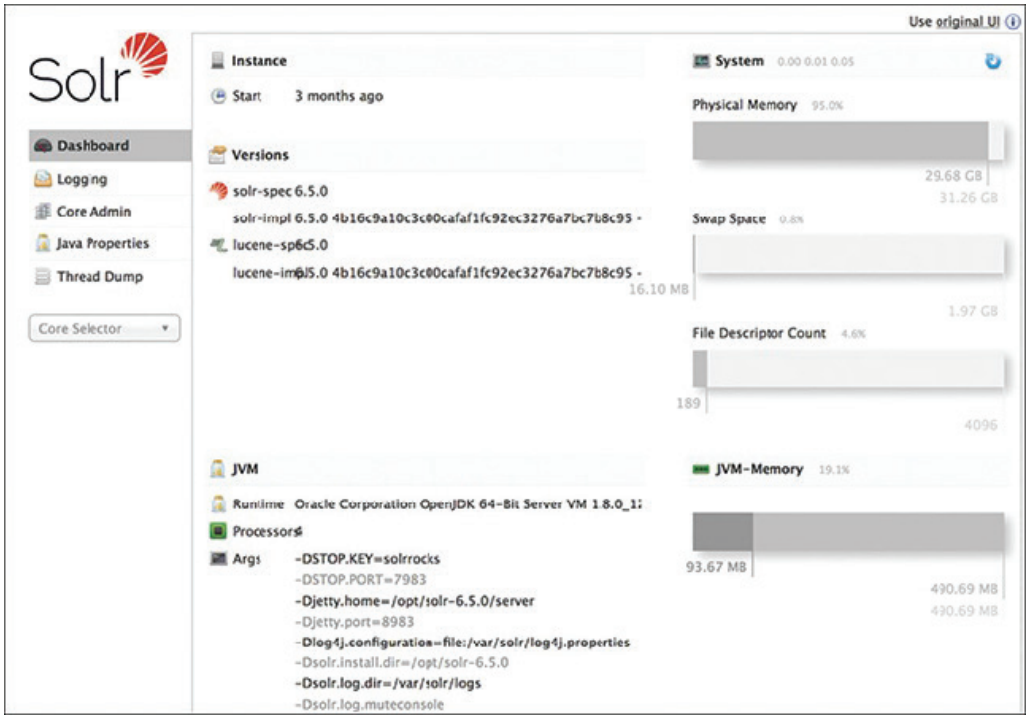


Рис. B.2 ❖ Веб-интерфейс Solr

Для поиска с использованием Solr будет создан отдельный микросервис, но для индексирования в Solr будет использоваться информация из существующей базы данных. Существует несколько разных утилит для извлечения данных из MySQL/PostgreSQL в Solr. (Вам также может понадобиться организовать непрерывную синхронизацию между базой данных приложения и Solr. В этом случае вы можете извлечь данные только один раз.) Мы будем использовать простой механизм импортирования данных и с его помощью импортируем все необходимые таблицы в Solr.

Настройка импорта данных в Solr

Необходимо выполнить операции:

1) добавьте следующие настройки в *solrconfig.xml*. Они определяют путь к файлу с настройками импорта данных. Этот файл устанавливается вместе с Solr. Измените путь `/path/to/my/dbconfigfile.xml`, указав фактический путь к файлу на вашей машине:

```
<requestHandler name="/dataimport"
                class="org.apache.solr.handler.dataimport.DataImportHandler">
  <lst name="defaults">
    <str name="config">/path/to/my/dbconfigfile.xml</str>
  </lst>
</requestHandler>
```

2) добавьте следующие строки в файл *dbconfigfile.xml*. Здесь мы указываем таблицу в базе данных, которую должен проиндексировать Solr. Мы также указали источник данных и запрос для их получения:

```
<dataConfig>
  <dataSource driver="org.hsqldb.jdbcDriver"
              url="jdbc:hsqldb:./example-DIH/hsqldb/ex"
              user="sa" password="secret"/>
  <document>
    <entity name="products" query="select * from products "
            deltaQuery="select id from products
            where updated_date >
            '${dataimporter.last_index_time}'">
    />
  <document>
<dataConfig>
```

3) вернитесь в командную оболочку и выполните следующую команду, чтобы импортировать и индексировать данные:

```
bin/solr -e dih
```

После индексации данных в Solr останется только создать RESTful веб-службу, которая запрашивает данные из Solr и возвращает результаты семантического поиска. Теперь все готово к созданию микросервиса поиска, описанного в главе 12.

Предметный указатель

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.a-planeta.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Парминдер Сингх Кочер

Микросервисы и контейнеры Docker

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Киселев А. Н.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать офсетная.
Усл. печ. л. 19,5. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**