Эммануил Гадзурас

Docker Compose для разработчика

Упростите разработку и оркестрацию многоконтейнерных приложений

ДМК Пресс
издательство

Разработка программного обеспечения становится все сложнее из-за использования различных инструментов. Приложения приходится упаковывать вместе с программными компонентами, чтобы упростить их работу, но это усложняет их запуск. С помощью Docker Compose можно всего одной командой настроить приложение и необходимые зависимости.

Вы познакомитесь с основами томов и сетей Docker, с командами Compose, их назначением и вариантами использования. Настроите базу данных для повседневной работы, доступную через сеть Docker, установите связь между микросервисами. Научитесь с помощью Docker Compose запускать целые стеки локально, моделировать промышленные окружения и расширять задания CI/CD. Кроме того, узнаете, как извлечь выгоду из Docker Compose при создании развертываний в промышленных окружениях, а также подготовите инфраструктуру в общедоступных облаках.

Вы научитесь:

- *создавать многоконтейнерные приложения с помощью Docker Compose;*
- *использовать Docker Compose в повседневной работе;*
- *подключать микросервисы друг к другу, используя сети Docker;*
- *осуществлять мониторинг служб с использованием Prometheus;*
- *развертывать приложения в промышленных окружениях с помощью Docker Compose;*
- *преобразовывать файлы Compose в развертывания Kubernetes.*

Издание предназначено программистам и инженерам DevOps, которые желают научиться настраивать многоконтейнерные приложения Docker. Оно также будет полезно руководителям групп, стремящимся повысить продуктивность команд разработки.

Интернет-магазин:
www.dmkpress.com

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

Packt›

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
www.дмк.рф

ISBN 978-5-93700-203-7

Эммануил Гадзурас

Docker Compose для разработчика

Эммануил Гадзурас

Docker Compose для разработчика

Упростите разработку и оркестрацию многоконтейнерных приложений

ДМК Пресс
издательство
Москва, 2023

УДК 004.451Docker
ББК 32.972.1
Г13

Гадзурас Э.
Г13 Docker Compose для разработчика / пер. с англ. А. Н. Киселева. – М.: ДМК Пресс, 2023. – 220 с.: ил.

ISBN 978-5-93700-203-7

Разработка программного обеспечения становится все сложнее из-за использования различных инструментов. Приложения приходится упаковывать вместе с программными компонентами, чтобы упростить их работу, но это усложняет их запуск. С помощью Docker Compose можно всего одной командой настроить приложение и необходимые зависимости.

Вы познакомитесь с основами томов и сетей Docker, с командами Compose, их назначением и вариантами использования. Настроите базу данных для повседневной работы, доступную через сеть Docker, установите связь между микросервисами. Научитесь с помощью Docker Compose запускать целые стеки локально, моделировать промышленные окружения и расширять задания CI/CD. Кроме того, узнаете, как извлечь выгоду из Docker Compose при создании развертываний в промышленных окружениях, а также подготовите инфраструктуру в общедоступных облаках.

Издание предназначено программистам и инженерам DevOps, которые желают научиться настраивать многоконтейнерные приложения Docker. Оно также будет полезно руководителям групп, стремящимся повысить продуктивность команд разработки.

УДК 004.451Docker
ББК 32.972.1

ISBN 978-1-80323-436-6 (англ.)
ISBN 978-5-93700-203-7 (рус.)

Посвящается
замечательному технологическому сообществу Лондона. Технологическое сообщество Лондона и его экосистема помогают мне оставаться в курсе последних тенденций, воодушевляют и дают возможность общения с прекрасными инженерами. Также посвящается моим коллегам из Yapily

Содержание

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Об авторе

Эммануил Гадзурас (Emmanouil Gkatziouras) начал свой путь в области разработки программного обеспечения, когда поступил на работу в департамент вычислительной техники и информатики в городе Патры (Греция). Затем он работал инженером-программистом в различных компаниях. В 2015 году поступил на работу в Oseven, где начал работать с облачными провайдерами, такими как AWS и Azure, и инструментами оркестрации контейнеров, такими как ECS и Kubernetes. Он занимал самые разные должности и в настоящее время работает как облачный архитектор в команде платформы.

Он любит помогать сообществу разработчиков, активно участвуя в проектах с открытым исходным кодом, таких как InfluxDB, Spring Cloud GCP и Alpakka, а также публикуя в блоге статьи по информатике на различные темы. Старается непрерывно учиться и уже обладает многими сертификатами, такими как CKA, CCDAK, PSM, CKAD и PSO.

Хочу поблагодарить самого себя за то, что собрался с силами и нашел время для написания этой книги, не прекращая при этом выполнять свои повседневные обязанности. Также хочу поблагодарить мою подругу Вив (Viv) за то, что терпела, пока я писал эту книгу, и всю команду редакторов из издательства Packt, которые помогали мне: Роми Диас (Romy Dias), Ашвин Динеш Харва (Ashwin Dinesh Kharwa) и Нираджан Найквади (Niranjan Naikwadi).

О рецензенте

Вернер Дейкерман (Werner Dijkerman) – специалист в облачных технологиях, Kubernetes (сертифицированный специалист) и DevOps. В настоящее время занимается облачными решениями и инструментами, включая AWS, Ansible, Kubernetes и Terraform, а также платформами IaaC (Infrastructure as a Code – инфраструктура как код). Широко использует инструменты мониторинга и автоматизации, такие как Zabbix, Prometheus и ELK Stack, и старается автоматизировать все и вся, чтобы максимально исключить ручную работу.

Большое спасибо, обнимаю и привет
Эрнсту Форстефельду (Ernst Vorsteveld)!

Предисловие

В этой книге описываются основы Docker Compose и демонстрируется его практическое применение. Сначала вы узнаете, какие компоненты входят в состав Docker Compose, какие команды доступны, их назначение и как они используются. Затем мы рассмотрим настройку баз данных, приемы использования сетевых возможностей Docker и организации взаимодействий между микросервисами. Вы также узнаете, как запускать целые стеки локально в Compose, как моделировать промышленные окружения и расширять задания CI/CD с помощью средств, предлагаемых Docker Compose. Наконец, вы познакомитесь с дополнительными темами, такими как использование Docker Compose для развертывания промышленных окружений, подготовки инфраструктуры в общедоступных облаках, таких как AWS и Azure, а также проложите путь для миграции на механизм оркестрации Kubernetes.

Кому адресована эта книга

Эта книга адресована инженерам-программистам, разработчикам и инженерам DevOps, которые желают научиться настраивать многоконтейнерные приложения Docker с помощью Compose без необходимости изучать и настраивать механизм оркестрации Docker. Она также будет полезна руководителям групп, стремящимся повысить продуктивность команд разработчиков за счет оптимизации подготовки сложных окружений разработки с помощью Docker Compose.

О чем рассказывается в книге

Глава 1 «*Введение в Docker Compose*» перечисляет различные особенности и варианты использования Compose. Дает краткое описание формата файла Docker Compose и приводит первый работающий пример на основе Compose.

Глава 2 «*Запуск первого приложения с помощью Compose*» показывает, как создать простое приложение Golang, взаимодействующее с базой данных Redis. К концу вы научитесь запускать многоконтейнерные приложения с помощью Compose.

Глава 3 «*Основы сетей и томов*» посвящена основам организации томов и сетей в Docker. К концу вы научитесь настраивать и использовать сеть для существующего приложения.

Глава 4 «*Выполнение команд Docker Compose*» знакомит с командами Compose, их назначением и вариантами использования.

Глава 5 «*Подключение микросервисов друг к другу*» рассматривает особенности создания новых микросервисов. К концу вы научитесь разрабатывать новые микросервисы, работающие в одной сети, и связывать их друг с другом.

Глава 6 «*Мониторинг служб с помощью Prometheus*» рассказывает, как организовать мониторинг служб с помощью Prometheus.

Глава 7 «*Комбинирование файлов Compose*» рассматривает вопросы модульной организации файлов Compose и разделение их на несколько частей.

Глава 8 «*Локальное моделирование промышленного окружения*» дает обзор сложных конфигураций Compose с целью частичного или полного моделирования промышленного окружения в локальной среде.

Глава 9 «*Создание расширенных заданий CI/CD*» показывает, как создавать сложные задачи CI/CD путем моделирования с помощью Compose.

Глава 10 «*Развертывание Docker Compose на удаленных хостах*» описывает приемы развертывания на удаленных хостах с помощью Compose.

Глава 11 «*Развертывание Docker Compose в AWS*» демонстрирует применение имеющихся знаний о Compose для развертывания в AWS с использованием ECS.

Глава 12 «*Развертывание Docker Compose в Azure*» посвящена еще одному популярному облачному провайдеру – Azure. К концу вы научитесь выполнять развертывание в Azure ACI.

Глава 13 «*Миграция на конфигурацию Kubernetes с помощью Compose*» показывает, как преобразовать файлы Compose в развертывание Kubernetes.

Как получить максимум от этой книги

Предполагается, что читатель понимает основные идеи контейнеризации и имеет базовые знания о Docker. Также желательно иметь навыки работы в командной строке. Лучшим вариантом для изучения книги было бы параллельное использование рабочей станции с UNIX для опробования примеров. Большая часть представленного кода и команд также должна работать на компьютерах с Windows.

Если вы читаете электронную версию этой книги, то мы советуем вводить код самостоятельно или получить его из репозитория книги на GitHub (ссылка приводится в следующем разделе). Это поможет вам избежать возможных ошибок, связанных с копированием и вставкой кода.

Типографские соглашения

В этой книге используется ряд соглашений по оформлению текста.

`Код в тексте`: так оформляются фрагменты программного кода в тексте, имена таблиц баз данных, имена папок, имена файлов, расширения файлов,

пути к файлам, фиктивные URL, ввод пользователя и ссылки Twitter. Например: «Смонтируйте созданный файл конфигурации `nginx.conf` как любой другой файл в вашей системе».

Блоки кода оформляются так:

```
type Task struct {
    Id          string `json:"id"`
    Name        string `json:"name"`
    Description string `json:"description"`
    Timestamp   int64  `json:"timestamp"`
}
```

Фрагменты кода, заслуживающие особого внимания, будут выделяться жирным шрифтом:

```
services:
  redis:
  image: redis
  ports:
    - 6379:6379
```

Любой ввод или вывод в командной строке будет оформляться так:

```
$ curl --location --request POST 'localhost:8080/task/'
$ cat /etc/nginx/nginx.conf
```

Жирным шрифтом будут выделяться новые термины, важные определения или слова, которые видны на экране. Например, надписи в меню или в диалоговых окнах будут выделены жирным шрифтом. Например: «Выберите раздел **System info** (Информация о системе) в панели администратора».

Советы или важные примечания будут оформляться так.

Часть I

ОСНОВЫ DOCKER COMPOSE

Эта часть знакомит с Docker Compose и его работой за кулисами. Мы познакомимся с Compose, разработав и развернув с его помощью набор приложений. Мы также узнаем, как идеи Docker, используемые ежедневно (такие как сети и тома), отражаются в Compose. Наконец, мы перечислим доступные команды Compose и посмотрим, как они выполняются.

Данная часть включает следующие главы:

- главу 1 «Введение в Docker Compose»;
- главу 2 «Запуск первого приложения с помощью Compose»;
- главу 3 «Основы сетей и томов»;
- главу 4 «Выполнение команд Docker Compose».

Глава 1

Введение в Docker Compose

Docker быстро стал неотъемлемой частью разработки и развертывания приложений, поэтому вам часто придется сталкиваться с инструментом **Docker Compose**. Возможно, вы читали о нем, использовали его или даже сталкивались с ним, просматривая официальную документацию Docker.

С развитием программного обеспечения для приложений стало обычным делом взаимодействовать с несколькими программными компонентами. Популярные приложения часто сталкиваются с необходимостью разделения рабочих нагрузок и масштабирования. Разделение логики и ответственности между несколькими программными компонентами неизбежно. Docker предлагает решения, упрощающие контейнеризацию и изоляцию рабочих нагрузок, а также управление ими. А Docker Compose может помочь в разработке современных многоконтейнерных приложений и их развертывании.

Docker Compose – простой и эффективный инструмент. Его применение помогает решать проблемы, возникающие при работе с многоконтейнерными приложениями, и повышать продуктивность разработчиков. Docker Compose можно использовать не только в жизненном цикле разработки, но также в промышленных окружениях, что позволяет устранить разрыв между разработкой в локальной среде и фактическими развертываниями в промышленных окружениях. Эту возможность можно использовать для плавного перехода к механизмам оркестрации, таким как Kubernetes.

В данной главе мы кратко познакомимся с основными возможностями Compose, принципами его работы и типичными вариантами использования. Мы установим Docker Compose и создадим первый файл Compose для запуска программного компонента. Исследуя формат файла Compose, мы также применим некоторые дополнительные настройки и используем один из наших локальных образов.

В этой главе рассматриваются следующие темы:

- знакомство с Docker Compose и особенностями его использования;
- установка Docker Compose;
- знакомство с принципами работы Docker Compose;

- создание первого файла Docker Compose;
- использование образа Docker в Docker Compose.

Технические требования

Примеры кода для этой книги можно найти в репозитории GitHub по адресу https://github.com/PacktPublishing/A-Developer-s-Essential-Guide-to-Docker-Compose. Если мы будем обновлять код, то все обновления будут доступны в этом репозитории.

Знакомство с Docker Compose и особенностями его использования

Docker Compose – это инструмент для определения и запуска многоконтейнерных приложений Docker. Конфигурация определяется с помощью файлов YAML, а с помощью **утилиты командной строки Docker Compose** можно определять и выполнять операции с контейнерами, управляемыми Docker Compose.

Вот список функций, которые предлагает Compose:

- запуск сложных многоконтейнерных приложений на одном хосте;
- изоляция рабочих нагрузок Docker;
- загрузка и распределение сложных приложений;
- создание нескольких окружений;
- сохранение данных об изменении приложения;
- обновление версий приложений;
- конструирование окружения;
- повторное использование конфигураций;
- моделирование сложных промышленных окружений;
- развертывание промышленных приложений.

В этой книге мы подробно рассмотрим все эти функции, увидим, какую пользу можно извлечь из них, и добавим их в наш процесс разработки. В следующем разделе мы установим Docker и Compose на рабочую станцию, используя выбранную вами операционную систему.

Установка Docker Compose

Docker Compose и Compose CLI написаны на языке Go. Compose поддерживает три основные операционные системы: Linux, Windows и macOS. Compose предназначен для управления многоконтейнерными приложениями Docker,

поэтому наличие установленного программного обеспечения Docker является обязательным предварительным условием.

Docker Desktop

В Mac и Windows **Docker Desktop** – это вариант установки Docker Compose. Docker Desktop упрощает настройку Docker на локальном компьютере. Он создает **виртуальную машину Linux** на хосте и поддерживает взаимодействие контейнеров с ОС, например доступ к файловой системе и сети. Установка Docker Desktop **выполняется** в один щелчок и добавляет в систему все необходимые инструменты, такие как Docker CLI. Одним из этих инструментов является Docker Compose. Поэтому установки Docker Desktop достаточно для взаимодействия с Docker Engine с помощью Compose.

Установка Docker

Чтобы установить правильный дистрибутив Docker для выбранной рабочей станции, перейдите в соответствующий раздел на официальной странице Docker:

- Docker Desktop для Mac: https://docs.docker.com/desktop/mac/install/;
- Docker Desktop для Windows: https://docs.docker.com/desktop/windows/install/;
- Docker Engine для Linux: https://docs.docker.com/engine/install/.

В macOS

Apple поставляет рабочие станции с двумя типами процессоров: процессором Intel и процессором Apple. Docker имеет дистрибутивы для обоих. После завершения загрузки, щелкнув на установщике, вы сможете перетащить приложение Docker, как показано на следующем скриншоте (рис. 1.1).

Рис. 1.1 ❖ Установка Docker в Mac

По завершении установки можно проверить работоспособность Docker, выполнив команду `hello world`:

```
$ docker run --rm hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
93288797bd35: Pull complete
Digest: sha256:97a379f4f88575512824f3b352bc03cd75e239179eea
0fecc38e597b2209f49a
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working
correctly.
..
```

Дополнительно можно проверить работоспособность Compose:

```
$ docker compose version
Docker Compose version v2.2.3
```

Теперь посмотрим, как установить Docker Desktop в Windows.

В Windows

В Windows, как и в Mac, установка Docker Desktop не вызывает никаких сложностей.

После загрузки и запуска установочного файла EXE в систему будет установлен Docker вместе со всеми утилитами. Затем необходимо выполнить некоторые дополнительные настройки, чтобы включить виртуализацию в Windows.

Независимо от используемого механизма – WSL 2 или Hyper-V – необходимо настроить BIOS компьютера и включить в нем виртуализацию, как показано на следующем скриншоте (рис. 1.2).

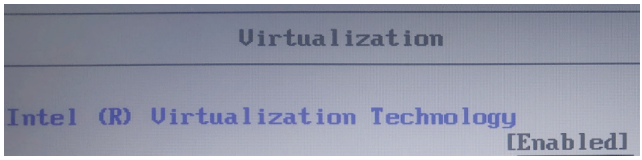

Рис. 1.2 ❖ Включение виртуализации в BIOS

После входа в Windows включите соответствующие функции виртуализации.

Для WSL 2 необходимо включить функцию **Virtual Machine Platform** (платформа виртуальных машин) и **Windows Subsystem for Linux** (подсистема Windows для Linux), как показано на рис. 1.3.

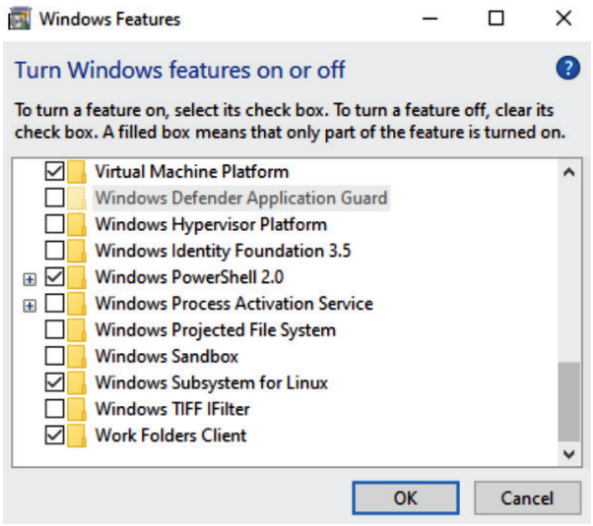

Рис. 1.3 ❖ Включение виртуализации для WSL 2

Для Hyper-V нужно включить **Hyper-V** (рис. 1.4).

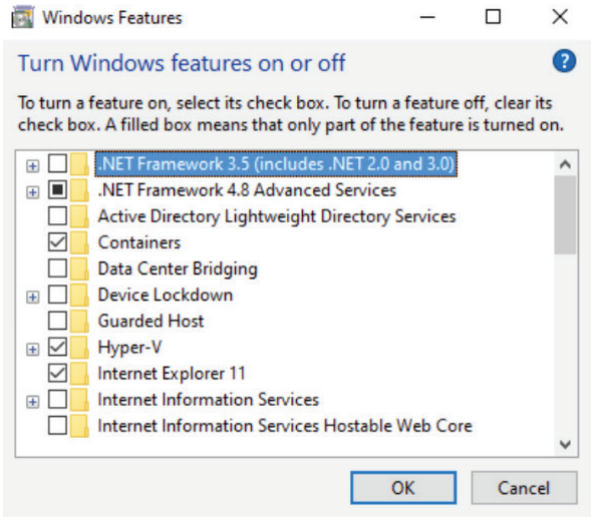

Рис. 1.4 ❖ Включение виртуализации для Hyper-V

Прежде чем продолжить, убедитесь, что ваша учетная запись пользователя добавлена в группу `docker-users`. После этого выйдите из Windows и войдите снова. Запустите Docker и выполните свою первую команду Docker в PowerShell:

```
PS C:\Users\my-user> docker run -d -p 80:80 docker/gettingstarted
Unable to find image 'docker/getting-started:latest' locally
latest: Pulling from docker/getting-started
```

```
59bf1c3509f3: Pull complete
8d6ba530f648: Pull complete
5288d7ad7a7f: Pull complete
39e51c61c033: Pull complete
ee6f71c6f4a8: Pull complete
f2303c6c8865: Pull complete
0645fddcff40: Pull complete
d05ee95f5d2f: Pull complete
Digest: sha256:aa945bdff163395d3293834697fa91fd4c725f47093ec499
f27bc032dc1bdd16
Status: Downloaded newer image for docker/gettingstarted:
latest
852371fcb34fddfe900bddc669af3a7aaab8743f8555fbb9952904bd2516ae7a
PS C:\Users\my-user>
```

Давайте также проверим, установился ли Docker Compose:

```
PS C:\Users\my-user> docker compose version
Docker Compose version v2.2.3
```

Теперь посмотрим, как установить Docker Desktop в Linux.

В Linux

На момент написания этих строк версия Docker Desktop для Linux была недоступна, но находилась в планах, и это всего лишь вопрос времени, когда она появится. Однако для использования Docker Compose достаточно установить **Docker Engine**.

Наиболее распространенный метод установки – добавить репозитории Docker на рабочую станцию Linux, а затем установить Docker Community Edition с помощью диспетчера пакетов.

Если у вас уже установлена старая версия Docker, то ее следует удалить и установить новые версии `docker-ce` и `docker-ce-cli`. Будем считать, что Docker ранее не был установлен у вас.

Поскольку большой популярностью пользуются дистрибутивы Linux на основе Red Hat, мы установим Docker в Fedora – дистрибутив Linux на основе Red Hat.

Сначала установите пакет `dnf-plugins-core`, содержащий инструменты, которые могут помочь в управлении репозиториями `dnf`:

```
$ sudo dnf -y install dnf-plugins-core
```

Затем добавьте репозиторий `docker-ce` для доступа к двоичным файлам Docker:

```
$ sudo dnf config-manager --add-repo https://download.docker.
com/linux/fedora/docker-ce.repo
```

Далее, после настройки репозитория, добавьте пакеты:

```
$ sudo dnf install docker-ce docker-ce-cli containerd.io -y
```

Docker – это демон, который выполняется как служба. Поэтому его запуск производится с помощью команды `systemctl`:

```
$ sudo systemctl start docker
```

Теперь запустите пример `hello-world`:

```
$ sudo docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working
correctly.
...
```

Как видите, почти в каждой команде нам пришлось использовать `sudo`. От этой рутинной обязанности можно избавиться с помощью группы `docker` – пользователи, включенные в нее, получают разрешение на взаимодействие с Docker Engine. Эта группа автоматически создается при установке Docker Engine:

```
$ sudo groupadd docker
$ sudo usermod -aG docker $USER
$ docker run hello-world
```

После установки и настройки Docker можно приступать к установке Compose.

Используем для установки ссылку https://docs.docker.com/compose/install/#install-compose-on-linux-systems:

```
$ sudo curl -L "https://github.com/docker/compose/releases/
download/1.29.2/docker-compose-$(uname -s)-$(uname -m)" -o /
usr/local/bin/docker-compose
$ sudo chmod +x /usr/local/bin/docker-compose
$ docker-compose –version
docker-compose version 1.29.2, build 5becea4c
```

Здесь можно заметить, что по ссылке доступна более старая версия Compose, чем те, что мы видели в предыдущих разделах. К сожалению, пока что не существует стандартного способа установки Compose V2 в Linux, как, например, в Mac и Windows путем установки Docker Desktop. Однако, поскольку Compose V2 все же можно установить в Linux, мы сделаем это, чтобы далее в книге сосредоточиться исключительно на Compose V2.

Последуем рекомендациям в официальной документации, доступной по адресу https://docs.docker.com/compose/cli-command/#install-on-linux:

```
$ mkdir -p ~/.docker/cli-plugins/
$ curl -SL https://github.com/docker/compose/releases/download/
v2.2.3/docker-compose-linux-x86_64 -o ~/.docker/cli-plugins/
docker-compose
$ chmod +x ~/.docker/cli-plugins/docker-compose

$ docker compose version
Docker Compose version v2.2.3
```

docker compose и docker-compose

В разделах, посвященных установке, можно заметить, что в Linux была установлена `docker compose` для Python.

Аналогичную команду можно найти в Windows, но там она имеет имя `docker-compose`:

```
PS C:\Users\my-user> docker-compose-v1.exe version
docker-compose version 1.29.2, build 5becea4c
docker-py version: 5.0.0
CPython version: 3.9.0
OpenSSL version: OpenSSL 1.1.1g 21 Apr 2020
PS C:\Users\my-user>
```

Первоначально Docker Compose создавался на Python; поэтому в инструкциях по установке упоминается установка пакетов `pip`.

Обратите внимание, что в новых версиях Docker Desktop команда `docker-compose` является псевдонимом для `docker compose`.

Первоначальная версия `docker-compose` по-прежнему поддерживается и развивается. Приложениям Compose, созданным и запущенным с помощью `docker-compose`, доступны вспомогательные инструменты, такие как **Compose Switch** (https://docs.docker.com/compose/cli-command/#compose-switch) для упрощения миграции.

При установке Compose Switch старая команда `docker-compose` заменяется командой `compose-switch`.

Compose Switch будет интерпретировать команды, предназначенные для передачи в `docker-compose`, и преобразует их в команды, которые сможет выполнить Compose V2. Затем он вызовет Compose V2 с помощью этой команды.

В нашей книге мы сосредоточимся на версии Compose V2, потому что она является частью `docker-cli`. Эта версия устанавливается по умолчанию в Docker Desktop и поддерживает самые новые функции и дополнительные команды.

Итак, мы установили Docker и Docker Compose и получили некоторое представление о том, как выполнять простые команды. Мы также познакомились с предыдущей версией Compose и узнали, как перейти на последнюю версию. Далее мы погрузимся в исследование особенностей работы Compose и посмотрим, как он взаимодействует с Docker Engine.

Знакомство с принципами работы Docker Compose

Установив Docker и Docker Compose, уделим немного времени Compose и узнаем, что это такое и как он работает за кулисами.

На GitHub можно найти проект (https://github.com/docker/compose) с исходным кодом Docker Compose. Рассматривая исходный код, можно многое узнать о Compose:

- Compose интегрируется с Docker CLI как плагин;
- Compose взаимодействует с Docker Engine через API;
- Compose предоставляет интерфейс командной строки и преобразует команды в вызовы Docker Engine API;
- Compose читает файл Compose в формате YAML и генерирует описанные в нем ресурсы;
- Compose предоставляет слой для преобразования команд `docker-compose` в CLI-совместимые аналоги;
- Compose взаимодействует с объектами Docker и различает их с помощью меток.

Docker CLI предоставляет API для создания и загрузки плагинов. После создания и загрузки плагина при его вызове ему передается команда CLI:

```
func pluginMain() {
    plugin.Run(func(dockerCli command.Cli) *cobra.Command {
        ...
    }
}

func main() {
    if commands.RunningAsStandalone() {
        os.Args = append([]string{"docker"},
                         compatibility.Convert(os.Args[1:])...)
    }
    pluginMain()
}
```

Интерфейс командной строки (CLI) основан на Cobra (https://github.com/spf13/cobra) – популярной библиотеке Go для создания приложений командной строки.

Compose, будучи плагином Docker CLI, использует клиента Docker Engine API, предоставляемого Docker CLI:

```
lazyInit.WithService(compose.NewComposeService(dockerCli.Client(), dockerCli.ConfigFile()))
```

Каждая команда, переданная плагину Docker Compose, инициирует взаимодействие с Docker Engine API на нашем хосте. Например, вот как реализована команда `ls`:

```
func (s *composeService)
List(ctx context.Context, opts api.ListOptions) ([]api.Stack, error) {
    list, err := s.apiClient.ContainerList(ctx, moby.ContainerListOptions{
        Filters: filters.NewArgs(hasProjectLabelFilter()),
        All: opts.All,
    })
    if err != nil {
        return nil, err
    }
    return containersToStacks(list)
}
```

Теперь у вас должно сложиться хорошее понимание, как Compose работает и взаимодействует с Docker Engine. Вы также можете обратиться к исходному коду, чтобы получить дополнительную информацию. Далее мы с вами запустим наше первое приложение Docker Compose.

Ваш первый файл Docker Compose

Представьте, что вам нужно запустить статическую веб-страницу на сервере. Для этой задачи хорошим выбором будет установка сервера NGINX. У нас есть простой HTML-файл `static-site/index.html`:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Hello World</title>
    </head>
    <body>
        <p>Hi! This application should run on docker-compose</p>
    </body>
</html>
```

Используя Docker, мы запустим сервер NGINX, взяв официальный образ, доступный по адресу https://www.docker.com/blog/how-to-use-the-official-nginx-docker-image/:

```
$ docker run --rm -p 8080:80 --name nginx-compose nginx
```

Давайте разберем эту команду:

- Docker Engine запускает Docker-образ сервера NGINX;
- по умолчанию образу назначается порт 80, который отображается в порт 8080 хост-системы, чтобы избежать сложностей с использованием порта из привилегированного диапазона;
- для упрощения взаимодействий с контейнером назначаем ему постоянное имя;
- аргумент `--rm` гарантирует, что после выполнения задачи и остановки контейнер будет удален.

Наш контейнер запущен и работает. Открыв другое окно терминала, мы сможем получить доступ к странице по умолчанию:

```
$ curl 127.0.0.1:8080
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
```

```
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully
installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

У нас получилось запустить сервер NGINX в контейнере. Теперь нам нужно адаптировать команду так, чтобы сервер возвращал нашу HTML-страницу. Для этого достаточно смонтировать файл в контейнер. Завершите выполнение предыдущей команды, нажав комбинацию **Ctrl**+**C**, а затем введите такую команду:

```
docker run --rm -p 8080:80 --name nginx-compose -v $(pwd)/
static-site:/usr/share/nginx/html nginx
```

Как и ожидалось, сейчас по умолчанию возвращается указанная нами страница:

```
$ curl localhost:8080/index.html
<!DOCTYPE html>
<html>
    <head>
        <title>Hello World</title>
    </head>
    <body>
        <p>Hi! This application should run on docker-compose</p>
    </body>
</html>
$
```

Теперь у нас есть все необходимое для переноса этого приложения в Compose. Для этого создадим файл Compose, который будет устанавливать NGINX:

```
services:
  nginx:
    image: nginx
    ports:
      - 8080:80
```

Рассмотрим его подробнее:

- мы указали имя службы `nginx`;
- для запуска службы будет использоваться тот же образ NGINX;

- для взаимодействий с внешним миром настраиваются те же порты, что и прежде.

Это содержимое следует сохранить в файл с именем `docker-compose.yaml`.

Далее выполним команду Compose в терминале:

```
$ docker compose up
[+] Running 2/0
  Network chapter1_default Created
0.0s
  Container chapter1-nginx-1 Created
0.0s

Attaching to chapter1-nginx-1
chapter1-nginx-1 | /docker-entrypoint.sh: /dockerentrypoint.
d/ is not empty, will attempt to perform
configuration
chapter1-nginx-1 | /docker-entrypoint.sh: Looking for shell
scripts in /docker-entrypoint.d/
...
$
```

HTTP-запрос к серверу возвращает тот же результат, что и при запуске контейнера Docker.

Имя файла играет важную роль. Мы использовали команду Compose, чтобы проанализировать файл Compose, но не указали имя этого файла. Так же, как в случае с командой `docker build` и файлом Dockerfile, команда `docker compose` будет искать файл с именем `docker-compose.yaml` в текущем каталоге. Если файл существует, он будет использован как файл Compose по умолчанию. Но имейте в виду, что мы не ограничены единственным именем файла; мы можем использовать любое другое имя файла для наших приложений Compose. В следующих главах вы увидите случаи использования других имен для файлов Compose и будете запускать приложение, используя параметр `-f`.

Далее мы смонтируем свою HTML-страницу в конфигурации Compose:

```
services:
  nginx:
    image: nginx
    ports:
      - 8080:80
    volumes:
      - ./static-site:/usr/share/nginx/html
```

Какой бы простой ни казалась предыдущая команда Docker, за кулисами она создает том Docker, согласно указанному пути в файловой системе, а затем подключает его к контейнеру. То же относится и к Compose. Здесь мы определяем том, указав путь в нашей файловой системе, который затем монтируется в каталог контейнера:

```
$ curl localhost:8080/index.html
<!DOCTYPE html>
<html>
```

```
    <head>
        <title>Hello World</title>
    </head>
    <body>
        <p>Hi! This application should run on docker-compose</p>
    </body>
</html>
```

Как и ожидалось, результат получился тем же, что и в примере с Docker.

Итак, в этом разделе мы запустили экземпляр NGINX с помощью утилиты командной строки Docker, а затем проделали то же самое с помощью Compose, добавив соответствующие разделы YAML, определяющие параметры для команды Docker. Теперь мы можем сделать следующий шаг – создать и запустить образ Docker в Docker Compose.

Использование образа Docker в Docker Compose

Используя Compose, мы запустили образ NGINX и изменили HTML-страницу, отображаемую по умолчанию. Начав применять Compose, мы продолжим работать с этим инструментом и попробуем создать свой образ Docker.

Далее мы создадим образ NGINX, который выводит журнальные записи в формате JSON. Этот формат поддерживается такими инструментами, как **CloudWatch** (https://aws.amazon.com/cloudwatch/), **StackDriver** (https://cloud.google.com/products/operations) и **ELK Stack** (https://www.elastic.co/elastic-stack/), которые предлагают дополнительные возможности просмотра журналов с поиском и фильтрацией записей по полям в элементах JSON.

Для решения этой задачи мы должны определить, как в NGINX задается текущий формат журналирования. Поскольку у нас уже есть готовый контейнер, мы запустим его с помощью Compose и проверим конфигурацию:

```
$ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED
STATUS       PORTS                  NAMES
dc0ca7ebe0cb   nginx     "/docker-entrypoint.…"   7 hours ago
Up 7 hours   0.0.0.0:8080->80/tcp   chapter1-nginx-1
$ docker exec -it chapter1-nginx-1 cat /etc/nginx/nginx.conf

user nginx;
worker_processes auto;

error_log  /var/log/nginx/error.log notice;
pid        /var/run/nginx.pid;

events {
```

```
    worker_connections  1024;
}
http {
    include       /etc/nginx/mime.types;
    default_type  application/octet-stream;

    log_format  main  '$remote_addr - $remote_user [$time_local] "$request" '
                      '$status $body_bytes_sent "$http_referer" '
                      '"$http_user_agent" "$http_x_forwarded_for"';
```

Найдя работающий контейнер с помощью `docker ps` и выполнив команду `cat` в командной оболочке контейнера, мы получили текущую настройку `log_format`, как она определена в конфигурационном файле `/etc/nginx/nginx.conf` в контейнере. Мы изменим этот формат на JSON и создадим свой образ Docker с этой настройкой.

Для этого скопируем файл в локальную систему, чтобы применить изменения:

```
$ docker cp chapter1-nginx-1:/etc/nginx/nginx.conf nginx.conf
```

Отредактируем `nginx.conf`, задав в параметре `log_format` значение `json`:

```
log_format main escape=json '{"remote_addr":"$remote_
addr","remote_user":"$remote_user","time":"[$time_
local]","request":"$request",'
                    '"status":"$status","body_bytes_
sent":"$body_bytes_sent","http_referer":"$http_referer",'
                    '"http_user_agent":"$http_user_
agent","http_x_forwarded_for":"$http_x_forwarded_for"}';
```

Теперь содержимое файла выглядит так:

```
user  nginx;
worker_processes auto;

error_log  /var/log/nginx/error.log notice;
pid        /var/run/nginx.pid;

events {
    worker_connections 1024;
}

http {
    include       /etc/nginx/mime.types;
    default_type  application/octet-stream;

    log_format  main  escape=json '{"remote_addr":"$remote_
addr","remote_user":"$remote_user","time":"[$time_
local]","request":"$request",'
                      '"status":"$status","body_bytes_
sent":"$body_bytes_sent","http_referer":"$http_referer",'
```

```
                    '"http_user_agent":"$http_user_
agent","http_x_forwarded_for":"$http_x_forwarded_for"}';

    access_log  /var/log/nginx/access.log main;
    sendfile        on;
    #tcp_nopush     on;

    keepalive_timeout  65;

    #gzip on;

    include  /etc/nginx/conf.d/*.conf;
}
```

Получив необходимый конфигурационный файл, создадим базовый образ NGINX, который будет использовать эту конфигурацию. Для этого создадим следующий файл Dockerfile:

```
FROM nginx

COPY nginx.conf /etc/nginx/nginx.conf
```

и сгенерируем сам образ:

```
$ docker build -t custom-nginx:0.1 .
```

Давайте продолжим и используем этот образ с недавно созданным файлом `docker-compose.yaml`:

```
services:
  nginx:
    image: custom-nginx:0.1
    ports:
      - 8080:80
    volumes:
      - ./static-site:/usr/share/nginx/html

$ docker compose up
...
chapter1-nginx-1 | 2022/02/10 08:09:27 [notice] 1#1: start
worker process 33
chapter1-nginx-1 | {"remote_addr":"172.19.0.1","remote_
user":"","time":"[10/Feb/2022:08:09:33 +0000]","request":"GET
/ HTTP/1.1","status":"200","body_bytes_sent":"177","http_
referer":"","http_user_agent":"curl/7.77.0","http_x_forwarded_
for":""}
...
```

На данный момент Compose успешно работает с нашим приложением в нашем собственном образе Docker. До сих пор инструмента Compose было достаточно для использования нашего собственного образа с дополнительными изменениями в конфигурации. Результаты получились в точности такими же, как при запуске приложения с помощью команд Docker.

Итоги

В этой главе мы познакомились с Docker Compose и некоторыми его наиболее примечательными функциями. Мы установили Compose в разные операционные системы и определили различия между установками. Затем поговорили о различных версиях Compose, Docker-Compose V1 и Docker Compose V2, а также выбрали версию, которая будет использоваться в этой книге. Заглянув в исходный код Compose, мы сделали еще один шаг и проверили, как Compose работает и взаимодействует с интерфейсом командной строки Docker. Затем мы запустили приложение Docker с помощью команды `docker` и создали его эквивалент в Compose. Следующим шагом мы настроили образ, который использовали в первом примере, и развернули с помощью Compose.

Во второй главе мы с помощью Compose создадим приложение, которое будет работать и взаимодействовать с базой данных Redis.

Глава 2

Запуск первого приложения с помощью Compose

В предыдущей главе мы познакомились с инструментом Docker Compose и узнали, как мы можно извлечь выгоду из его возможностей. Мы выяснили, как он работает и взаимодействует с движком Docker и его интерфейсом командной строки (Docker CLI). Установив Compose на выбранную нами рабочую станцию, мы смогли запустить несколько примеров приложений.

Использование Compose в разработке приложения может упростить процесс и сыграть важную роль в повышении производительности. К концу этой главы вы научитесь упаковывать свои приложения и запускать их с помощью Compose, взаимодействовать с приложением, расширять его функциональные возможности и подключать к базе данных. После развертывания базового многоконтейнерного приложения с помощью Compose мы подробнее исследуем дополнительные возможности Compose, такие как проверка работоспособности, определение меток и переменных окружения, а также переопределение команд.

В этой главе рассматриваются следующие темы:

- создание простого приложения;
- запуск Redis с помощью Compose;
- использование командной оболочки в контейнере, управляемом Compose;
- взаимодействие со службой Docker Compose;
- упаковка приложения с помощью Docker и Compose;
- запуск мультиконтейнерного приложения с помощью Compose.

Технические требования

Примеры кода для этой книги можно найти в репозитории GitHub по адресу https://github.com/PacktPublishing/A-Developer-s-Essential-Guide-to-Docker-Compose. Если мы будем обновлять код, то все обновления будут доступны в этом репозитории.

Создание простого приложения

В этом разделе мы напишем приложение на языке программирования **Go** и будем использовать его на протяжении всей книги для демонстрации возможностей Compose. Go – это язык программирования с открытым исходным кодом. Многие программные приложения, существующие в настоящее время, написаны на этом языке, включая Docker, Kubernetes и Compose. Программирование на Go открывает нам доступ к надежной библиотеке с необходимыми пакетами, реализующими, например, http-серверы, утилиты для тестирования и многое другое.

Установка Go

Установка Go на рабочую станцию выполняется легко и просто. Откройте в браузере страницу загрузки https://go.dev/doc/install, отыщите установочный пакет для своей операционной системы и загрузите его.

REST API в Go с использованием Gin

Реализовать REST API в Go можно с использованием существующих библиотек или стороннего фреймворка. Мы выберем вариант с фреймворком, чтобы сохранить кодовую базу максимально простой и иметь возможность сосредоточить все наше внимание на Compose. В роли фреймворка мы используем **Gin** (https://gin-gonic.com/) – популярный фреймворк для Go, который поможет организовать маршрутизацию, отображение JSON и избежать лишнего кода, который пришлось бы писать, если бы мы использовали только библиотеки Go.

Итак, создадим наш модуль, как показано ниже:

```
$ go mod init task_manager
```

Для начала воспользуемся шаблоном приложения:

```
$ curl https://raw.githubusercontent.com/gin-gonic/examples/
master/basic/main.go > main.go
```

Затем загрузим фреймворк Gin:

```
$ go get github.com/gin-gonic/gin
```

И создадим модуль с помощью `go build`:

```
$ go build
```

Теперь можно запустить приложение командой `go run`:

```
$ go run main.go
```

После запуска проверим, работает ли приложение, воспользовавшись утилитой `curl`:

```
$ curl http://localhost:8080/user/John
{"status":"no value","user":"John"}
```

Мы успешно реализовали наш первый REST API на языке Go. Мы воспользовались некоторыми вспомогательными инструментами, чтобы начать разработку приложения. На следующем шаге мы создадим наше базовое приложение, взяв за основу только что созданный шаблон.

Приложение

Приложение будет называться **Task Manager** (Диспетчер задач) и обслуживать интерфейс REST API. Пользователь приложения будет посылать ему задачи с помощью `POST`-запросов, получать задачи с помощью `GET`-запросов, а также удалять их с помощью `DELETE`-запроса. Таким образом мы должны сформировать следующие конечные точки:

- `GET /task`: для получения всех задач;
- `POST /task`: для добавления задачи;
- `GET /task/{id}`: для получения задачи по идентификатору;
- `DELETE /task/{id}`: для удаления задачи по идентификатору.

Добавив некоторые изменения в методе `main.go`, мы преобразуем класс в сервер задач. Текущая реализация будет хранить задачи в памяти. Код приложения может находиться в одном файле, как показано на странице https://raw.githubusercontent.com/PacktPublishing/A-Developer-s-Essential-Guide-to-Docker-Compose/main/Chapter2/example-task-manager/main.go.

Структура данных, описывающая задачу, будет содержать идентификатор, имя задачи, описание и отметку времени:

```
type Task struct {
    Id          string `json:"id"`
    Name        string `json:"name"`
    Description string `json:"description"`
    Timestamp   int64  `json:"timestamp"`
}
```

Далее определяются контроллеры, реализующие упомянутые действия:

```
// Получение задач
r.GET("/task", func(c *gin.Context) {
    tasks := []Task{}
    for _, v := range taskMap {
        tasks = append(tasks, v)
    }
    c.JSON(http.StatusOK, gin.H{"tasks": tasks})
})

// Получение задачи по идентификатору
r.GET("/task/:id", func(c *gin.Context) {
    id := c.Params.ByName("id")
    task, ok := taskMap[id]
    if ok {
        c.JSON(http.StatusOK, gin.H{"task": task})
    } else {
        c.JSON(http.StatusNotFound, gin.H{"id": id,
            "message": "not found"})
    }
})

// Добавление задачи
r.POST("/task", func(c *gin.Context) {
    var task Task
    if err := c.BindJSON(&task); err != nil {
        c.JSON(http.StatusOK, gin.H{"task": task,
            "created": false, "message": err.Error()})
    } else {
        taskMap[task.Id] = task
        c.JSON(http.StatusCreated, gin.H{"task": task,
            "created": true, "message": "Task Created Successfully"})
    }
})

// Удаление задачи
r.DELETE("/task/:id", func(c *gin.Context) {
    id := c.Params.ByName("id")
    delete(taskMap, id)
    c.JSON(http.StatusOK, gin.H{"id": id, "message":
        "deleted"})
})
```

Запустим наш REST API и попробуем выполнить несколько вызовов с помощью `curl`. Чтобы отправить запрос на *создание задачи*, выполним следующую команду:

```
$ curl --location --request POST 'localhost:8080/task/' \
--header 'Content-Type: application/json' \
--data-raw '{
    "id": "8b171ce0-6f7b-4c22-aa6f-8b110c19f83a",
    "name": "A task",
```

```
    "description": "A task that need to be executed at the
timestamp specified",
    "timestamp": 1645275972000
}'
{"created":true,"message":"Task Created Successfully","tas
k":{"id":"8b171ce0-6f7b-4c22-aa6f-8b110c19f83a","name":"A
task","description":"A task that need to be executed at the
timestamp specified","timestamp":1645275972000}}
```

Чтобы отправить запрос на *получение списка всех задач*, выполним следующую команду:

```
$ curl --location --request GET 'localhost:8080/task'
{"tasks":[{"id":"8b171ce0-6f7b-4c22-aa6f-
8b110c19f83a","name":"A task","description":"A
task that need to be executed at the timestamp
specified","timestamp":1645275972000}]}
```

Чтобы получить конкретную задачу с заданным идентификатором, выполним следующую команду:

```
$ curl --location --request GET 'localhost:8080/task/8b171ce0-
6f7b-4c22-aa6f-8b110c19f83a'
{"task":{"id":"8b171ce0-6f7b-4c22-aa6f-8b110c19f83a","name":"A
task","description":"A task that need to be executed at the
timestamp specified","timestamp":1645275972000}}
```

Для простоты и переносимости наших примеров мы использовали утилиту `curl`. Однако есть еще один популярный инструмент для взаимодействия с REST API – Postman. Коллекция доступна по адресу https://github.com/PacktPublishing/A-Developer-s-Essential-Guide-to-Docker-Compose/blob/main/Chapter2/Task-Manager.postman_collection.json.

На данный момент мы создали приложение диспетчера задач, поддерживающее REST API, а затем протестировали его работу с помощью `curl`. Далее мы расширим наше приложение для использования сервера Redis в качестве хранилища.

Запуск Redis с помощью Compose

Наше базовое приложение готово. Для простоты вместо хранения задач в базе данных мы использовали карту в памяти. Это хорошо работает для прототипирования; однако наши данные и задачи остаются только в одном процессе. В случае запуска двух экземпляров нашего приложения каждый экземпляр будет содержать разные задачи.

Чтобы решить эту проблему, мы будем хранить данные в базе данных. Таким образом, данные будут храниться в одном месте и не будет различий в задачах, обслуживаемых разными экземплярами. Redis будет нашим выбором для хранения данных.

Redis – популярное хранилище структур данных в памяти. Оно широко применяется в качестве кеша, и многие популярные облачные провайдеры используют его в качестве решения кеширования. Такие компоненты, как **ElastiCache** в Amazon Web Services (https://aws.amazon.com/elasticache) или **Memorystore** в Google Cloud Platform (https://cloud.google.com/memorystore), могут привлекать Redis. Кроме того, Redis можно использовать для работы с базами данных. Также этот инструмент включает поддержку *брокеров* и *потоков*. Redis и его универсальные возможности будут помогать нам в этой книге подчеркивать возможности Docker Compose.

Запускать Redis мы будем с помощью Docker Compose, поэтому создадим файл Compose с описанием службы Redis:

```
services:
  redis:
    image: redis
    ports:
      - 6379:6379
```

Служба будет называться `redis`. По умолчанию Redis использует порт `6379`, поэтому привяжем его к локальному `6379`. Запустив Redis командой `compose up`, мы сможем взаимодействовать с базой данных `redis`:

```
$ docker compose up
...
chapter2-redis-1  | 1:M 19 Feb 2022 14:18:20.302 # Server
initialized
chapter2-redis-1  | 1:M 19 Feb 2022 14:18:20.303 * Ready to
accept connections
```

Теперь база данных Redis запущена и готова принимать запросы. В следующем разделе мы исследуем контейнер Redis, поближе познакомимся с ним и выполним некоторые команды.

Использование командной оболочки в контейнере, управляемом Compose

Мы успешно запустили базу данных Redis с помощью Compose и теперь попробуем выполнить несколько команд в этом экземпляре Redis, чтобы поближе познакомиться с базой данных. Как и во многих контейнерных дистрибутивах баз данных, образ может содержать не только саму базу данных, но и инструменты, помогающие взаимодействовать с ней в административных целях.

Интерфейс командной строки Redis можно применять для отправки команд в службу Redis. Образ Redis Docker содержит утилиты командной строки для работы с Redis, и мы можем использовать их для исследования работающей базы данных.

Давайте найдем наш работающий образ Redis:

```
$ docker ps --format "{{.Names}}"
chapter2-redis-1
$
```

Войдем в командную оболочку в этом образе:

```
$ docker exec -it chapter2-redis-1 bash
root@d189b089bcf6:/data#
```

Мы только что успешно вошли в командную оболочку в контейнере, управляемом Compose. Теперь посмотрим, что имеется в нашем распоряжении:

```
$ root@d189b089bcf6:/data# ls
dump.rdb
root@d189b089bcf6:/data# ls /usr/local/bin
docker-entrypoint.sh  gosu  redis-benchmark  redis-check-aof
redis-check-rdb  redis-cli  redis-sentinel  redis-server
```

Каталог по умолчанию содержит файл `dump.rdb`. Кроме того, как показывает вывод команды `ls`, в этом каталоге имеются двоичные файлы, такие как `redis-server`, `redis-sentinel` и `redis-cli`. Поскольку интерфейс командной строки Redis уже присутствует в образе, мы можем выполнить некоторые команды на работающем сервере Redis:

```
$ root@d189b089bcf6:/data# redis-cli
127.0.0.1:6379>
```

Redis будет служить основным хранилищем для задач, поэтому добавим идентификатор задачи с помощью команды `ZADD`, а затем проверим результат:

```
127.0.0.1:6379> ZADD tasks 1645275972000 "8b171ce0-6f7b-4c22-
aa6f-8b110c19f83a"
(integer) 1

127.0.0.1:6379> ZRANGE tasks 0 -1 WITHSCORES
1) "8b171ce0-6f7b-4c22-aa6f-8b110c19f83a"
2) "1645275972000"
```

После успешного запуска сервера Redis с помощью Compose мы вошли в командную оболочку в контейнере Redis, выяснили, какие инструменты командной строки имеются в нашем распоряжении, и использовали команду `redis-cli` для взаимодействия с сервером. В следующем разделе мы добавим в наше приложение возможность применения сервера Redis для хранения задач.

Взаимодействие со службой Docker Compose

Мы запустили Redis с помощью Compose, *вошли в командную оболочку* в этом экземпляре, попробовали выполнить некоторые команды и доба-

вили некоторые данные. Но, как вы понимаете, взаимодействовать с этим экземпляром можно не только посредством командной оболочки. Экземпляр настроен на прием запросов через порт `6379`, связанный с локальным портом `6379`.

Например, мы можем взаимодействовать с этим экземпляром с помощью клиента `redis-cli`, имеющего доступ к хосту `localhost`. В следующем примере запускается еще один образ Redis Docker, который обращается к нашей базе данных Redis, управляемой Compose:

```
$ docker run --rm -it --entrypoint bash redis -c 'redis-cli -h
host.docker.internal -p 6379'
host.docker.internal:6379> ZRANGE tasks 0 -1 WITHSCORES
1) "8b171ce0-6f7b-4c22-aa6f-8b110c19f83a"
2) "1645275972000"
host.docker.internal:6379>
```

Как видите, запись, сохраненная ранее, никуда не исчезла и осталась в базе данных. Мы благополучно подключились к **службе Compose** извне и теперь приступим к адаптации нашего приложения для использования Redis вместо словаря в памяти. Мы будем использовать структуру данных, которая называется **отсортированное множество**. Отсортированное множество подобно обычному множеству, элементы которого сопровождаются числовым значением – **оценкой**. Элементы в множестве сортируются по величине оценки. Если они имеют одинаковую оценку, то сортировка производится в лексикографическом порядке значений. Если внимательнее рассмотреть команду `ZADD`, которую мы запускали ранее, то можно заметить, что выбранная нами оценка на самом деле является отметкой времени. Итак, наши задачи будут отсортированы по времени.

Также нам нужно иметь возможность получать задачи по идентификатору (`id`). **Хеши** – хороший вариант, потому что они идеально подходят для представления объектов. Используя **хеш**, можно быстро отыскать и получить значение структуры по ее ключу:

```
host.docker.internal:6379> HMSET task:8b171ce0-6f7b-4c22-
aa6f-8b110c19f83 Id 8b171ce0-6f7b-4c22-aa6f-8b110c19f83a Name
"A task" Description "A task that need to be executed at the
timestamp specified" Timestamp 1645275972000
OK

host.docker.internal:6379> HGETALL task:8b171ce0-6f7b-4c22-
aa6f-8b110c19f83
1) "Id"
2) "8b171ce0-6f7b-4c22-aa6f-8b110c19f83a"
3) "Name"
4) "A task"
5) "Description"
6) "A task that need to be executed at the timestamp specified"
7) "Timestamp"
8) "1645275972000"
host.docker.internal:6379>
```

Давайте выйдем из сеанса и завершим работу контейнера, нажав комбинацию **Ctrl**+**C**.

Теперь адаптируем приложение для использования Redis в роли базы данных. Существует несколько реализаций клиентов, поэтому выберем ту, что пользуется наибольшей популярностью в настоящее время, – клиента `go-redis/redis`.

Импортируем `go-redis/redis` в проект:

```
$ go get github.com/go-redis/redis/v8
```

После некоторых изменений наше приложение должно обслуживать запросы, извлекая и сохраняя данные в Redis. Реализацию операций сохранения в Redis можно увидеть в исходном коде на GitHub: https://github.com/PacktPublishing/A-Developer-s-Essential-Guide-to-Docker-Compose/blob/main/Chapter2/main.go#L91.

Соответствующим образом необходимо адаптировать методы контроллера, как можно видеть в файле https://github.com/PacktPublishing/A-Developer-s-Essential-Guide-to-Docker-Compose/blob/main/Chapter2/main.go#L39.

Для каждой задачи будет создан эквивалентный хеш. Каждый хеш будет иметь ключ, начинающийся со строки `task`. Хеш будет хранить всю информацию, необходимую задаче. Чтобы организовать некоторую упорядоченность по дате задачи, используем отсортированное множество.

Таким образом, каждая конечная точка должна выполнять следующие операции с Redis:

- `GET /task`: получить идентификаторы задач, используя отсортированное множество, и извлечь соответствующие хеши;
- `POST /task`: сформировать хеш с именем `task:{id}` для задачи и добавить его в отсортированное множество;
- `GET /task/{id}`: получить хеш задачи с помощью ключа `task:{id}`;
- `DELETE /task/{id}`: удалить хеш с именем `task:{id}` и удалить соответствующий элемент из отсортированного множества.

Запустим наше обновленное приложение:

```
$ go run main.go
```

Запустив сервер Redis с помощью Compose, мы получили возможность задействовать Redis в нашем базовом приложении, которую успешно реализовали и тем самым заменили хранилище в памяти приложения соответствующими структурами данных в Redis. В следующем разделе мы упакуем приложение с помощью Docker и запустим его с помощью Compose.

Упаковка приложения с помощью Docker и Compose

Пришло время упаковать и развернуть приложение с помощью Compose. Для этого нужно создать образ Docker из нашего приложения. Но прежде следует

адаптировать наш код, чтобы он мог работать в разных окружениях без необходимости изменять его и генерировать другой образ.

Передача конфигурации через окружение

Исследуя код, можно заметить, что некоторые настройки может понадобиться изменить. Конфигурация Redis должна быть гибкой, потому что сервер Redis может находиться где угодно. По этой причине желательно, чтобы клиент Redis получал конфигурацию через переменные окружения. Однако если конфигурация не была определена явно, то следует вернуться к настройкам по умолчанию.

В этом нам поможет следующая функция. Если переменная окружения не определена, она будет возвращать значение по умолчанию:

```
func getIntEnv(key string, defaultvaule int) int {
    if value := os.Getenv(key); len(value) == 0 {
        return defaultvaule
    } else {
        if i, err := strconv.Atoi(value); err == nil {
            return i
        } else {
            return defaultvaule
        }
    }
}
func getStrEnv(key string, defaultValue string) string {
    if value := os.Getenv(key); len(value) == 0 {
return defaultValue
    } else {
        return value
    }
}
```

Далее организуем настройку конфигурации клиента Redis с использованием переменных окружения:

```
var (
    client = redis.NewClient(&redis.Options{
            Addr:     getStrEnv("REDIS_HOST", "localhost:6379"),
            Password: getStrEnv("REDIS_PASSWORD", ""),
            DB:       getIntEnv("REDIS_DB", 0),
    })
    taskMap = make(map[string]Task)
)
```

И наконец, настроим имя хоста и номер порта нашего сервера:

```
func main() {
     r := setupRouter()
     r.Run(getStrEnv("TASK_MANAGER_HOST", ":8080"))
}
```

Теперь наш существующий код будет получать настройки из переменных окружения. Это делает наше решение переносимым в различные среды. Если дополнительная конфигурация не задана, то приложение будет использовать настройки по умолчанию. Теперь можно приступать к созданию образа Docker.

Создание образа Docker

Благодаря предыдущим изменениям наш код теперь можно заключить в контейнер и запускать в различных окружениях. Вот как будет выглядеть файл Dockerfile для приложения:

```
# syntax=docker/dockerfile:1
FROM golang:1.17-alpine
WORKDIR /app
COPY go.mod ./
COPY go.sum ./
RUN go mod download
COPY *.go ./
RUN go build -o /task_manager
EXPOSE 8080
CMD [ "/task_manager" ]
```

Разберем его содержимое:

1. Определяется версия синтаксиса Dockerfile.
2. Используется образ на основе `golang alpine`.
3. `/app` – это каталог, где находится приложение и его зависимости.
4. Копирование файлов `mod` и `sum`.
5. Запуск команды, которая загрузит зависимости.
6. Копирование файлов нашего приложения.
7. Сборка приложения.
8. Открывается порт, на котором должно работать приложение, для доступа извне.
9. Определяется команда для запуска приложения после развертывания образа.

Теперь создадим образ:

```
$ docker build . -t task-manager:0.1
```

Создав файл Dockerfile, мы можем упаковать наше приложение в образ Docker и запустить его с помощью Docker CLI или Compose.

Запуск образа

После создания образа его можно запустить:

```
$ docker run --rm -p 8080:8080 --env REDIS_HOST=host.docker.
internal:6379 task-manager:0.1
```

Приложение будет работать в точности как ожидалось, и, как видите, при запуске мы указали ему имя хоста, на котором работает сервер Redis. Теперь давайте вместо хоста Redis попробуем использовать контейнер Redis, работающий в Compose. Поскольку контейнер Redis у нас уже имеется, его следует подключить к сети, созданной Compose. Отыскать сеть можно, перечислив сети, присутствующие в Docker:

```
$ docker network ls
NETWORK ID     NAME               DRIVER    SCOPE
1392d8a87032   bridge             bridge    local
...
453bafcfd97a   chapter2_default   bridge    local
4a6d6a3e8475   host               host      local
...
```

Мы подключим контейнер к сети `chapter2_default`:

```
$ docker run --rm -p 8080:8080 --env REDIS_HOST=redis:6379
--network=chapter2_default task-manager:0.1
```

Обратите внимание, что мы сменили определение переменной окружения `REDIS_HOST`. Поскольку контейнер работает в той же сети, доступ к экземпляру Redis можно получить, просто указав имя службы, которое будет преобразовано в IP-адрес внутри сети.

Теперь, получив образы, мы можем добавить их в файл Compose, чтобы не запускать по отдельности:

```
services:
  task-manager:
    image: task-manager:0.1
    ports:
      - 8080:8080
    environment:
      - REDIS_HOST=redis:6379
  redis:
    image: redis
    ports:
      - 6379:6379
```

Посмотрим, как это работает:

- Compose запускает обе службы;
- сеть не указана явно, поэтому обе службы используют сеть по умолчанию;
- находясь в одной сети, они могут обращаться друг к другу по именам служб;
- все необходимые переменные окружения определяются в разделе `environment`.

Пересоздадим приложение, чтобы изменения вступили в силу:

```
$ docker compose up --force-recreate
```

Наше многоконтейнерное приложение теперь работает под полным управлением Compose.

Создание образа с помощью Compose

К настоящему моменту мы упаковали наше приложение с помощью Compose. Но, как вы наверняка заметили, на протяжении всей этой главы код нашего приложения неоднократно изменялся. Из диспетчера задач, хранящего данные в памяти, оно превратилось в приложение баз данных, поддерживающее возможность настройки некоторых параметров через переменные окружения. Очевидно, это были не последние изменения в коде. В будущем код продолжит изменяться, и поэтому важно организовать автоматическую перестройку и запуск приложения.

Вместо сборки образа и перехода на новую версию путем изменения содержимого раздела `image` в файле Compose вручную мы должны как-то сообщить Compose, что настал момент пересобрать образ и запустить его. Для этого нужно настроить Compose для использования Dockerfile с целью сборки и использования образа:

```
services:
  task-manager:
    build: .
    ports:
      - 8080:8080
    environment:
      - REDIS_HOST=redis:6379
  redis:
    image: redis
    ports:
      - 6379:6379
```

Образ содержит файлы Dockerfile и `docker-compose.yaml` в одном и том же каталоге, поэтому в `build:` мы указываем один и тот же каталог – «точку» (`.`), соответствующую текущему каталогу. Если код находится в другом образе, то можно указать путь и запустить Compose с несколькими каталогами и даже разными проектами.

Сборка и определение имени образа

До сих пор мы использовали приложение, указывая *имя образа*. Теперь попробуем сделать то же самое в Compose, определив `build` и `image`. При одновременном использовании `build` и `image` Compose создаст образ, а затем пометит его, применяя имя, заданное в параметре `image`:

```
services:
  task-manager:
```

```
    build: .
    image: task-manager:0.1
```

Начав с создания образа Docker вручную в командной строке, мы перешли к автоматизации процесса сборки с помощью Compose. И теперь, заложив основы для использования Compose с нашим приложением, мы можем перейти к более продвинутым концепциям.

Запуск многоконтейнерного приложения с помощью Compose

Научившись запускать многоконтейнерное приложение в Compose, можно переходить к более конкретным концепциям и выяснить, как получить максимальную отдачу от нашего приложения при локальной разработке.

Проверка работоспособности

Наше приложение имеет дополнительную конечную точку, которая не упоминалась выше. Она известна как **конечная точка ping**. Как можно заметить в коде, эта конечная точка отвечает на вызовы простым неизменным сообщением:

```
// Проверка работоспособности
r.GET("/ping", func(c *gin.Context) {
    c.String(http.StatusOK, "pong")
})
```

Эта конечная точка будет использоваться для **проверки работоспособности**. Если приложение работоспособно, оно обязательно вернет ответ при вызове данной конечной точки. Если с приложением что-то не так, то конечная точка не ответит и будет помечена как *неработоспособная*.

Как действует проверка работоспособности

Добавляя проверку работоспособности с помощью Compose, мы указываем команду для запуска в контейнере. В зависимости от успешности выполнения команды и временного порога контейнер будет помечен как *работоспособный* или *неработоспособный*. Указанная команда выполняется внутри контейнера. Использование несуществующей команды приведет к ошибке при проверке работоспособности.

В команде можно сослаться на любой выполняемый файл в контейнере. Это может быть существующая команда в контейнере или даже ваш собственный сценарий со сложным алгоритмом проверки работоспособности. Compose будет проверять код завершения сценария, выполняя его через указанные интервалы времени.

Поскольку наша конечная точка проверки работоспособности реализована на http-сервере, нам нужна команда, посылающая http-запрос к ней. Для

этого прекрасно подойдет утилита **cURL**; поэтому добавим ссылку на нее в Dockerfile и включим ее в образ:

```
# syntax=docker/dockerfile:1
FROM golang:1.17-alpine
RUN apk add curl
...
```

А затем создадим образ командой `compose build`:

```
$ docker compose build
```

Добавление проверки работоспособности в Compose

Мы установили в наш образ утилиту `curl` и теперь можем определить команду и параметры проверки работоспособности в разделе `healthcheck`:

```
services:
  task-manager:
    build: .
    image: task-manager:0.1
    ports:
      - 8080:8080
    environment:
      - REDIS_HOST=redis:6379
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8080/ping"]
      interval: 20s
      timeout: 10s
      retries: 5
      start_period: 5s
```

Каждые 20 секунд после начального периода в 5 секунд приложению `task-manager` будет посылаться запрос с помощью `curl` с тайм-аутом 10 секунд. В случае сбоя будет предпринято пять попыток повторить запрос, и только после этого экземпляр будет помечен как *неработоспособный*. При запуске `curl` с флагом `-f` эта команда будет сообщать о неуспехе при получении таких HTTP-кодов, как *5XX*.

Если теперь запустить приложение, то в журнале можно наблюдать, как производится проверка работоспособности:

```
$ docker compose up
...
chapter2-task-manager-1  | [GIN] 2022/02/20 - 17:23:07 | 200 |
8.584µs |       127.0.0.1 | GET      "/ping"
chapter2-task-manager-1  | [GIN] 2022/02/20 - 17:23:27 | 200
|      34.459µs |       127.0.0.1 | GET      "/ping"
chapter2-task-manager-1  | [GIN] 2022/02/20 - 17:23:47 | 200
|      42.458µs |       127.0.0.1 | GET      "/ping"
```

Запустим еще одно окно терминала и посмотрим, какие контейнеры запущены:

```
$ docker ps
CONTAINER ID   IMAGE              COMMAND
CREATED         STATUS                              PORTS
          NAMES
20c6d1cb557b   task-manager:0.1   "/task_manager"          4
seconds ago   Up 3 seconds (health: starting)   0.0.0.0:8080-
>8080/tcp   chapter2-task-manager-1
$ docker ps
20c6d1cb557b   task-manager:0.1   "/task_manager"          2
minutes ago   Up 44 seconds (healthy)   0.0.0.0:8080->8080/tcp
   chapter2-task-manager-1
```

Можно пойти еще дальше и настроить конечную точку как неработоспособную, чтобы можно было посмотреть, как помечается неработоспособная служба:

```
// Проверка работоспособности
r.GET("/ping", func(c *gin.Context) {
     c.String(http.StatusInternalServerError, "pong")
})
```

Через некоторое время контейнер действительно получает отметку `unhealthy` (неработоспособный):

```
$ docker ps
CONTAINER ID   IMAGE              COMMAND
CREATED         STATUS                      PORTS
NAMES
20c6d1cb557b   task-manager:0.1   "/task_manager"          2
minutes ago   Up 2 minutes (unhealthy)   0.0.0.0:8080->8080/tcp
chapter2-task-manager-1
```

Мы познакомились с возможностью проверки работоспособности в Compose и сумели воспроизвести успешные и неудачные сценарии проверки работоспособности.

Зависимости от служб

Служба диспетчера задач `task-manager` зависит от Redis. Если служба Redis не запущена, то диспетчер задач не сможет работать. В подобных случаях можно потребовать не запускать службу, пока другая служба, от которой зависит эта, не будет полностью запущена:

```
services:
  task-manager:
    build: .
    image: task-manager:0.1
    ports:
      - 8080:8080
    environment:
      - REDIS_HOST=redis:6379
```

```
    depends_on:
      - redis
```

Точка входа, аргументы и переменные окружения

Предположим, что перед запуском службы `task-manager` также желательно загрузить некоторые данные в базу данных Redis. Реализовать это можно разными способами. Для нас было бы предпочтительнее запустить другой контейнер `redis` с помощью **интерфейса командной строки Redis**, а для этого нужно переопределить *точку входа* и указать некоторые аргументы.

Начнем с первоначального определения службы:

```
redis-populate:
  image: redis
  depends_on:
    - redis
```

Оно пока не работает, но мы постепенно доработаем его.

Файлы с переменными окружения

Мы уже использовали переменные окружения, определяя их в разделе `environment` в виде пар ключ-значение, теперь попробуем определить их в файле `env`. Файл с переменными окружения будет определять хост и порт, к которому должен подключиться интерфейс командной строки Redis:

```
HOST=redis
PORT=6379
```

Следующий шаг – монтирование файла `env` в службу `redis-populate`:

```
redis-populate:
  image: redis
  depends_on:
    - redis
  env_file:
    - ./env.redis-populate
```

Сценарий

Выполнив несколько дополнительных настроек, можно обеспечить возможность использования сценария, выполняющего команды интерфейса командной строки Redis. Допустим, нам нужно добавить задачу перед запуском приложения; для этого можно поместить необходимые команды в текстовый файл:

```
HMSET task:a3a597d1-26f8-43e5-be05-81373f2c0dc3 Id a3a597d1-
26f8-43e5-be05-81373f2c0dc3 Name "Existing Task" Description "A
task that was here before" Timestamp 1645393434000
ZADD tasks 1645393434000 "a3a597d1-26f8-43e5-be05-81373f2c0dc3"
```

И передать его содержимое по конвейеру интерфейсу командной строки Redis. С помощью переменных окружения также можно организовать выполнение команд на разных серверах. Сценарий, выполняющий команды Redis из текстового файла, может выглядеть примерно так:

```
#!/bin/sh

cat $1| redis-cli -h $HOST -p $PORT
```

Эти команды можно заключить в выполняемый сценарий и смонтировать в контейнер `redis-populate`:

```
redis-populate:
  image: redis
  depends_on:
    - redis
  env_file:
    - ./env.redis-populate
  volumes:
    - ./redis-populate.txt:/redis-populate.txt
    - ./redis-populate.sh:/redis-populate.sh
```

Настройка точки входа

Чтобы запустить сценарий и заполнить базу данных, нужно изменить точку входа в Docker:

```
redis-populate:
  image: redis
  entrypoint: ["/redis-populate.sh","/redis-populate.txt"]
  depends_on:
    - redis
  env_file:
    - ./env.redis-populate
  volumes:
    - ./redis-populate.txt:/redis-populate.txt
    - ./redis-populate.sh:/redis-populate.sh
```

Определив точку входа `ENTRYPOINT` и переменные окружения, мы получаем возможность предварительно загрузить некоторые данные в приложение.

Метки

Метки – это одна из возможностей Docker, которые можно использовать в Compose. С помощью меток можно добавлять метаданные к объектам Docker, таким как образы, контейнеры, сети и тома. Это упрощает организацию приложений, например логическую группировку или поиск ресурсов с помощью фильтров меток.

Образы

Создавая образы, их можно снабжать метками:

```
build:
  context: .
  labels:
    - com.packtpub.compose.app=task-manager
```

А затем отфильтровать требуемые образы, используя эту метку:

```
$ docker images --filter "label=com.packtpub.compose.app=task-
manager"

REPOSITORY     TAG       IMAGE ID       CREATED         SIZE
task-manager   0.1       de4b8f7c9fff   15 hours ago    529MB
```

Контейнеры

В разделе `services` можно указать метку для контейнера:

```
redis:
  image: redis
  ports:
    - 6379:6379
  labels:
    - com.packtpub.compose.app=task-manager
```

И фильтровать контейнеры, используя эти метки:

```
docker images --filter "label=com.packtpub.compose.app=task-
manager"

$ docker ps --filter="label=com.packtpub.compose.app=task-
manager"
CONTAINER ID   IMAGE              COMMAND
CREATED          STATUS                    PORTS
NAMES
98c45777bdcd   redis              "docker-entrypoint.s…"   58
seconds ago   Up 57 seconds             0.0.0.0:6379->6379/tcp
chapter2-redis-1
```

Итак, мы упаковали наше приложение с помощью Compose, создали образ Docker и снабдили его меткой. Это помогло организовать запуск приложения в Compose от начала до конца, от реализации кода до запуска в контейнере Docker.

Итоги

В этой главе мы создали многоконтейнерное приложение, работающее под управлением Docker Compose, и использовали такие возможности Compose, как создание образа, добавление к нему метки, взаимодействие между контейнерами с использованием одной и той же сети, проверка работоспособности и настройка точки входа. Эти возможности упрощают переход от ручного управления на основе Docker к автоматизированному управлению на основе Compose. Разобравшись с возможностью упаковки сложного приложения, мы заложили основу для развертывания и распространения нашего многоконтейнерного приложения с помощью Compose.

Следующая глава посвящена управлению сетями и томами. В ней рассказывается, как идея томов и сетей в Docker используется в Compose.

Глава 3

Основы сетей и томов

В предыдущей главе мы написали наше базовое приложение на **Go** и организовали хранение данных с помощью сервера Redis. Затем мы исследовали более продвинутые концепции Docker, такие как проверка работоспособности, создание образов, добавление меток и логическая группировка с их использованием. На протяжении всей главы также использовались **тома** и **сети**, однако мы не акцентировали на них своего внимания. В этой главе, напротив, все наше внимание будет сосредоточено на сетях и томах, а также на их настройке в приложениях на основе Compose.

Первая часть будет посвящена томам, как они используются в Compose и как работают за кулисами. Тома играют важную роль в Docker. Они позволяют подключать внешнюю документацию и файлы, необходимые приложениям. Тома могут быть общими и использоваться для упрощения работы приложения.

Вторая часть будет посвящена сетям. Docker реализует простую и понятную схему поддержки сетей. В особых случаях можно использовать различные варианты, обеспечивающие связь между несколькими хостами, разные уровни сетевых взаимодействий между приложениями и лучшую изоляцию между контейнерами.

В этой главе рассматриваются следующие темы:

- основы томов Docker;
- подключение тома Docker к контейнеру;
- драйверы томов Docker;
- объявление томов Docker в файлах Compose;
- подключение тома Docker к существующему приложению;
- создание конфигурационного файла;
- монтирование файла с использованием тома;
- основы сетей Docker;
- определение сетей в конфигурации Compose;
- добавление дополнительной сети в текущее приложение.

Технические требования

Примеры кода для этой книги можно найти в репозитории GitHub по адресу https://github.com/PacktPublishing/A-Developer-s-Essential-Guide-to-Docker-Compose. Если мы будем обновлять код, то все обновления будут доступны в этом репозитории.

Основы томов Docker

Для многих приложений важно иметь возможность взаимодействий с файловой системой. Эти взаимодействия могут выполняться для сохранения данных или, в некоторых случаях, для чтения конфигурации. Примером может служить приложение базы данных, использующее файловую систему для обработки и сохранения данных. То же относится и к конфигурациям, которые применяются к приложениям. Давайте представим JEE-приложение, сильно зависящее от конфигурации *XML*. Оно должно иметь возможность читать и хранить данные в файловой системе. Но проблема дисковых операций в контейнерах Docker заключается в том, что после закрытия контейнера все изменения теряются. Чтобы этого не происходило, Docker предоставляет тома. Тома являются предпочтительным механизмом хранения данных, которые создаются и используются контейнером.

Вот некоторые характеристики томов:

- тома и данные в них сохраняются после закрытия контейнера;
- том можно подключить к другому контейнеру;
- тома могут использоваться несколькими контейнерами одновременно;
- тома абстрактны;
- тома могут быть локальными или удаленными.

Теперь, познакомившись с томами в общих чертах, давайте посмотрим, как подключать тома к контейнерам и взаимодействовать с ними.

Подключение тома Docker к контейнеру

Создадим наш первый том Docker:

```
$ docker volume create example-volume
example-volume
```

И проверим его:

```
$ docker volume ls --filter="name=example-volume"
--format='{{json .}}'
{"Driver":"local","Labels":"","Links":"N/A","Mountpoint":"/var/
lib/docker/volumes/example-volume/_data","Name":"example-volume
","Scope":"local","Size":"N/A"}
```

Как видите, том использует локальный драйвер. Локальный драйвер – это драйвер по умолчанию, используемый при создании томов. Том со всеми своими данными будет находиться на хосте, где работает Docker Engine. Также в примере выше можно видеть точку монтирования в локальной файловой системе. Именно здесь физически находятся данные из этого тома.

Поскольку мы уже создали том, используем его в контейнере. Для этого подключим том к контейнеру с помощью параметра `--mount` и запишем в него некоторые данные:

```
$ docker run -it --rm --name example-volume-container --mount
source=example-volume,target=/storage bash
bash-5.1# echo some-text > /storage/data-file.txt
bash-5.1# cat /storage/data-file.txt
some-text
bash-5.1# exit
```

Рассмотрим происходящее здесь подробнее:

1. Запускается контейнер с bash в интерактивном режиме.
2. После выхода из контейнера он автоматически будет остановлен и удален.
3. В процессе запуска в контейнер монтируется том в каталог `/storage`.
4. Создается файл в томе.
5. После выхода файл должен остаться в томе.

Теперь создадим еще один контейнер, смонтируем в него этот же том и проверим, сохранился ли наш файл:

```
$ docker run -it --rm --name second-volume-container --mount
source=example-volume,target=/storage bash
bash-5.1# cat /storage/data-file.txt
some-text
```

Как видите, файл никуда не делся.

В другом окне терминала проверим контейнер и раздел `Mounts`:

```
$  docker inspect second-volume-container --format '{{json
.Mounts}}'
[{"Type":"volume","Name":"example-volume","Source":"/var/
lib/docker/volumes/example-volume/_data","Destination":"/
storage","Driver":"local","Mode":"z","RW":true,
"Propagation":""}]
```

Теперь закроем оба терминала. Как вы могли видеть, том подключается и хранит информацию. Для подключения тома достаточно указать его фактическое местоположение и точку монтирования в контейнере. После успешного создания тома и его подключения к разным контейнерам перейдем к сценарию совместного использования тома.

Общие тома

Раньше мы использовали один том и подключали его к одному контейнеру. Остановив контейнер и подключив том к другому контейнеру, мы обнаружили, что сохраненные данные никуда не пропали. Но это еще не все: том можно подключить сразу к нескольким контейнерам. Если запустить следующие команды одновременно, то они запустят два контейнера, одновременно записывающие данные в один и тот же том:

```
$ docker run -it --rm  --name container-2 --mount
source=example-volume,target=/storage bash -c "for i in \$(seq
1 1 100000); do echo \$HOSTNAME \$i >> /storage/\$HOSTNAME.txt
; done"
$ docker run -it --rm  --name container-1 --mount
source=example-volume,target=/storage bash -c "for i in \$(seq
1 1 100000); do echo \$HOSTNAME \$i >> /storage/\$HOSTNAME.txt
; done"
```

Если теперь подключить том и перейти в каталог `/storage`, то можно увидеть два разных файла, созданных двумя контейнерами:

```
$ docker run -it --rm  --name container-1 --mount
source=example-volume,target=/storage bash
bash-5.1# ls /storage
770bdfc4dc94.txt  b9b09390cf17.txt data-file.txt
```

Оба контейнера благополучно сохранили свои файлы в томе.

Тома только для чтения

У нас получилось использовать оригинальный том во втором контейнере. Однако в некоторых сценариях возможность записывать данные должен иметь лишь один контейнер, а остальные могут *только читать* их.

Как показано в следующем примере, монтирование тома в режиме *только для чтения* не позволит записывать в него какие-либо файлы:

```
$ docker run -it --rm  --name read-only-volume-container
--mount source=example-volume,target=/storage,readonly bash
bash-5.1# cat /storage/data-file.txt
some-text
bash-5.1# touch /storage/test.txt
touch: /storage/test.txt: Read-only file system
bash-5.1# exit
```

Итак, мы познакомились с общими характеристиками томов и узнали, как их монтировать в контейнеры. Мы также успешно смонтировали том в режиме *только для чтения*. В следующем разделе мы поговорим о драйверах томов и о том, какие возможности они могут предложить.

Драйверы томов Docker

Мы создали тома Docker и использовали их в наших контейнерах. Изучив том, как показано в предыдущем примере, можно получить ценную информацию:

```
$ docker volume inspect example-volume
[
    {
        "CreatedAt": "2022-03-08T07:01:29Z",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/example-volume/_data",
        "Name": "example-volume",
        "Options": {},
        "Scope": "local"
    }
]
```

Здесь параметру `"Driver"` присвоено значение `"local"`. Это означает, что данные, записываемые в том, будут храниться локально на рабочей станции, а сам драйвер обеспечивает поддержку дисковых операций из контейнера Docker в пути в виртуальной машине.

Это дает нам возможность узнать физическое местоположение файла. Следующий пример показывает, как в macOS подключиться к виртуальной машине, где размещен Docker Engine, используя `netcat`:

```
$ nc -U ~/Library/Containers/com.docker.docker/Data/debug-
shell.sock
/ # cat /var/lib/docker/volumes/example-volume/_data/data-file.txt
cat /var/lib/docker/volumes/example-volume/_data/data-file.txt
some-text
/ # exit
exit
```

Давайте разберем этот пример подробнее. В macOS и Windows Docker Engine работает в виртуальной машине:

1. Нам нужно подключиться к этой виртуальной машине и исследовать содержимое каталога.
2. Оказавшись внутри виртуальной машины, мы можем убедиться, что данные продолжают там храниться.

Как вы наверняка понимаете, том может находиться где угодно, поэтому драйвер упрощает использование тома.

Драйверы томов могут обеспечивать взаимодействия с удаленными файловыми системами и сохранять данные там. В числе доступных драйверов можно назвать Azure File Storage, NFS и AWS EFS.

Помимо взаимодействия с томами на удаленных хостах, драйверы томов можно использовать для добавления дополнительных функций, вызываемых при сохранении данных, например для шифрования при передаче или сохранении.

Использование драйвера тома вместо локального монтирования

Как можно заключить из предыдущего раздела, существуют различные варианты, позволяющие организовать взаимодействие с удаленной файловой системой. Один такой вариант, действующий в обход драйвера, – монтирование файловой системы на хосте. В таком случае контейнер может получить доступ к удаленной файловой системе через хост, который ее уже смонтировал. Вы можете выбрать любой удобный для вас вариант, в зависимости от желания уменьшить накладные расходы на поддержку или получить расширенные возможности обработки, предлагаемые драйверами томов.

Выше мы создали несколько томов с помощью командной строки, посмотрели, как работают тома за кулисами, а также перечислили некоторые варианты организации взаимодействий с удаленными файловыми системами с помощью драйверов. В следующем разделе мы сосредоточимся на использовании томов в приложении Compose.

Объявление томов Docker в файлах Compose

Мы создали том Docker, использовали его и познакомились с некоторыми возможностями. На следующем шаге мы создадим и используем тома с помощью Compose. Тома в Compose определяются в разделах `volumes`:

```
services:
  nginx:
    image: nginx
volumes:
  example-compose-volume:
```

Тома, определяемые в разделах `volumes`, можно *снабжать метками*:

```
services:
  nginx:
    image: nginx
volumes:
  example-compose-volume:
    labels:
      - com.packtpub.compose.app=volume-example
```

Также можно определять более сложные конфигурации, например использовать драйвер `nfs`:

```
services:
volumes:
  nfsvol:
    driver_opts:
      type: "nfs"
```

```
      o: "addr=127.0.0.1,nolock,rw"
      device: ":/data"
```

Итак, мы создали и настроили тома с помощью Compose. Теперь можно вернуться к нашему приложению и адаптировать его для использования томов.

Подключение томов Docker к существующему приложению

Предыдущая глава была посвящена созданию приложения диспетчера задач, использующего базу данных Redis, которая, как и всякая другая база данных, должна периодически создавать резервные копии и выполнять другие дисковые операции.

Вот как выглядела наша предыдущая конфигурация Redis Compose:

```
services:
  redis:
    image: redis
    ports:
      - 6379:6379
    labels:
      - com.packtpub.compose.app=task-manager
```

Redis поддерживает разные варианты сохранения данных (https://redis.io/topics/persistence), от моментальных снимков – компактного представления данных *на определенный момент времени* – до *файлов, доступных только для добавления*. Мы сосредоточимся на варианте с моментальными снимками и будем использовать том для хранения этих снимков.

По умолчанию резервные копии `.rdb` в Docker-образе Redis хранятся в каталоге `/data`. Мы смонтируем том в этот каталог.

Сначала создадим том:

```
volumes:
  redis-data:
    labels:
      - com.packtpub.compose.app=task-manager
```

Затем подключим том к контейнеру:

```
services:
  redis:
    image: redis
    ports:
      - 6379:6379
    volumes:
      - redis-data:/data
```

После подключения тома к базе данных резервные копии будут сохраняться в нем и к ним сможет получить доступ другой контейнер. В следующем разделе мы определим свою конфигурацию для службы Redis, чтобы увеличить частоту резервного копирования.

Создание конфигурационного файла

Основываясь на документации, настроим Redis так, чтобы моментальный снимок создавался каждые 60 секунд, если изменился хотя бы один ключ. Вот как выглядит соответствующая настройка Redis:

```
dbfilename taskmanager.rdb
save 60 1
```

Здесь мы изменили имя резервной копии на `taskmanager.rdb` и указали, что моментальный снимок должен делаться каждые 60 секунд, если изменится хотя бы один ключ. Теперь, когда конфигурация готова, нужно подключить этот файл к базе данных, созданной выше с помощью Compose.

Монтирование файла с использованием тома

Созданный конфигурационный файл будет монтироваться с помощью поддержки **монтирования файлов в контейнер** (bind mount). Эта поддержка позволяет смонтировать файл, хранящийся на рабочей станции, где работает Docker Engine, в контейнер Docker. При использовании Compose монтируемые файлы определяются в разделах `volumes`. Нахождение в одном разделе с томами может сбивать с толку, и все же, когда определяется единственный файл, – это монтирование файла в контейнер.

Смонтируем конфигурационный файл Redis в контейнер и изменим точку входа:

```
services:
  redis:
    image: redis
    ports:
      - 6379:6379
    entrypoint: ["redis-server","/usr/local/etc/redis/redis.conf"]
    labels:
      - com.packtpub.compose.app=task-manager
    volumes:
      - ./redis.conf:/usr/local/etc/redis/redis.conf
      - redis-data:/data
```

Проверив контейнер, можно убедиться, что файл действительно был смонтирован:

```
$ docker inspect chapter3-redis-1 --format '{{json .Mounts }}'
...
          {
               "Type": "bind",
               "Source": "/path/to/repo/A-Developer-s-
Essential-Guide-to-Docker-Compose/Chapter3/redis.conf",
               "Destination": "/usr/local/etc/redis/redis.conf",
               "Mode": "rw",
               "RW": true,
               "Propagation": "rprivate"
           }
...
```

Мы настроили в Redis создание моментальных снимков в выбранном нами томе, а теперь продолжим создание резервных копий этих моментальных снимков, используя другой процесс, имеющий доступ к исходному тому только для чтения.

Монтирование томов только для чтения

Теперь нам нужно сохранить создаваемые моментальные снимки в файл. В таком случае мы сможем восстановить базу данных на определенный момент времени. Однако резервные копии мы будем сохранять в другом томе, единственной целью которого будет хранение резервных копий:

```
...
    volumes:
      - ./redis.conf:/usr/local/etc/redis/redis.conf
      - redis-data:/data
volumes:
  redis-data:
  backup:
```

Добавим еще один контейнер. Он будет отвечать за копирование резервных копий Redis и их перемещение в папку `backup`. Контейнер не будет содержать ничего сложного – только *сценарий* на bash, запускаемый каждые несколько секунд.

Вот этот сценарий на bash:

```
#!/bin/sh

while true; do
  cp /data/taskmanager.rdb /backup/$(date +%s).rdb;
  sleep $BACKUP_PERIOD;
done
```

Этот сценарий будет запускаться при запуске контейнера. Поскольку у нас есть все необходимое, настроим контейнер.

В новый контейнер должны быть смонтированы оба тома. Том с моментальными снимками Redis будет доступен *только для чтения*. Процесс ре-

зервного копирования будет копировать текущий файл из каталога `data` в контейнере Redis и добавлять к его имени отметку времени:

```
services:
  ...
  redis-backup:
    image: bash
    entrypoint: ["/snapshot-backup.sh"]
    depends_on:
      - redis
    environment:
      - BACKUP_PERIOD=10
    volumes:
      - ./snapshot-backup.sh:/snapshot-backup.sh
      - redis-data:/data:ro
      - backup:/backup
volumes:
  redis-data:
  backup:
```

Обратите внимание на параметр `ro` (*read only* – только для чтения) монтирования тома с папкой `data`. Установить режим *только для чтения* можно другим способом:

```
    volumes:
      - type: volume
        source: redis-data
        target: /data
        read_only: true
```

Наша база данных Redis подключает процесс резервного копирования моментальных снимков следующим образом:

```
services:
  ...
  redis-backup:
    image: bash
    entrypoint: ["/snapshot-backup.sh"]
    depends_on:
      - redis
    environment:
      - BACKUP_PERIOD=10
    volumes:
      - ./snapshot-backup.sh:/snapshot-backup.sh
      - redis-data:/data:ro
      - backup:/backup
    labels:
      - com.packtpub.compose.app=task-manager
volumes:
  redis-data:
    labels:
      - com.packtpub.compose.app=task-manager
```

```
  backup:
    labels:
      - com.packtpub.compose.app=task-manager
```

Давайте запустим предыдущее приложение Compose:

```
$ docker compose up
```

У нас должна сохраниться возможность выполнять те же запросы cURL, которые выполнялись в главе 2 «Запуск первого приложения с помощью Compose».

К настоящему моменту мы настроили нашу службу Redis для использования тома, добавили еще одну службу для запуска процесса резервного копирования, подключив к ней в режиме *только для чтения* том с моментальными снимками для резервного копирования и поместив ее в отдельный контейнер, *специально предназначенный для резервного копирования*. В следующем разделе мы перейдем к знакомству с сетями Docker и узнаем, как настраивать и использовать их в Compose.

Основы сетей Docker

Важнейшей особенностью Docker является его поддержка сетей. Контейнеры, работающие в Docker, могут беспрепятственно обмениваться данными между собой. Благодаря поддержке сетей Docker предоставляет различные сетевые возможности:

- приватные сети;
- внутреннюю службу DNS;
- связь между контейнерами;
- контейнеры, играющие роль мостов между сетями.

Давайте посмотрим, какие сети доступны в настоящий момент в нашей среде:

```
$ docker network ls
NETWORK ID     NAME               DRIVER    SCOPE
6a149c758fc2   bridge             bridge    local
9ec7758d7050   chapter1_default   bridge    local
7c95e79497b6   chapter2_default   bridge    local
16fda3c38e57   chapter3_default   bridge    local
4a6d6a3e8475   host               host      local
be28a5cd8a16   none               null      local
```

Docker поддерживает сети разных видов. К наиболее часто используемым относятся:

- мостовая сеть;
- сеть хоста;
- оверлейная сеть.

Рассмотрим каждую из разновидностей в следующих разделах.

Мостовая сеть

Мост – сетевой драйвер, используемый в Docker по умолчанию. Мостовая сеть в Docker – это программный уровень, обеспечивающий связь между контейнерами, подключенными к одной мостовой сети. Кроме того, эти контейнеры изолированы от контейнеров, не подключенных к этой сети.

В примере выше, где мы получили список имеющихся сетей, можно увидеть сеть с именем `bridge`. Это сеть по умолчанию, к которой должны подключаться контейнеры.

Давайте запустим контейнер `nginx` и проверим сеть, к которой он подключится:

```
$ docker run --rm -p 8080:80 --name nginx-compose nginx
```

Теперь, запустив контейнер, в другом окне терминала проверим сеть:

```
$ docker inspect --format '{{json .NetworkSettings.Networks }}'
nginx-compose
{"bridge":{
  "IPAMConfig":null,
  "Links":null,
  "Aliases":null,
  "NetworkID":"6a149c758fc27c0c719b7c3c26b74c8f03c866ae1406ac2902a1c45af720a79d",
  "EndpointID":"4323e8f8bbceafcb74491e1595073adc063fb424f87259ad1dd02078f59eeb13",
  "Gateway":"172.17.0.1",
  "IPAddress":"172.17.0.2",
  "IPPrefixLen":16,
  "IPv6Gateway":"",
  "GlobalIPv6Address":"",
  "GlobalIPv6PrefixLen":0,
  "MacAddress":"02:42:ac:11:00:02",
  "DriverOpts":null}
}
```

Как видите, контейнер был подключен к мостовой сети по умолчанию:

```
$ NETWORK_ID=$(docker inspect --format '{{json .NetworkSettings
.Networks.bridge.NetworkID }}' nginx-compose|sed 's/"//g')
$  docker network ls --filter ID=$NETWORK_ID
NETWORK ID     NAME      DRIVER    SCOPE
6a149c758fc2   bridge    bridge    local
```

Теперь можно выйти из сеанса в контейнере NGINX, нажав комбинацию **Ctrl**+**C**.

Мостовая сеть, определяемая пользователем

Мостовая сеть, *определяемая пользователем*, выглядит предпочтительнее сети по умолчанию. Контейнеры определяют сетевые адреса друг друга по именам или псевдонимам:

- контейнеры подключаются к сети, привязанной к их приложению;
- мостовую сеть можно настроить под нужды приложения.

Сеть хоста

При использовании сети хоста контейнер Docker не изолируется от хоста, на котором выполняется. Контейнер не имеет собственного IP-адреса, и доступ к нему осуществляется с использованием IP-адреса хоста.

Запустим экземпляр `nginx`, используя сеть хоста:

```
$ docker run --rm --network host --name host-nginx nginx
```

Давайте убедимся, что сервер работает, используя другой контейнер для доступа к `nginx` через `localhost`.

Для этого в другом окне терминала выполните следующую команду:

```
$ docker run -it --network host --rm --name nginx-wget bash
wget -O- localhost:80
Connecting to localhost:80 ([::1]:80)
writing to stdout
<!DOCTYPE html>
<html>
<head>
...
```

Как видите, в этом примере контейнеры действительно были подключены к сети хоста. Как и ожидалось, контейнеру не был назначен отдельный IP-адрес; поэтому любые запросы к `localhost` отображаются в хост. Это позволяет контейнеру `nginx-wget` получить доступ к контейнеру `host-nginx`.

Оверлейная сеть

Сети, которые мы исследовали выше, предназначены для использования в Docker Engine на одном хосте. Для применения Docker в промышленном окружении с механизмом оркестрации, таким как **Swarm**, необходимо несколько хостов. *Оверлейная* сеть создает распределенную сеть, охватывающую все эти хосты. Оверлейная сеть прозрачно соединяет сети хостов и создает единую сеть поверх них. Теперь, познакомившись с сетями Docker и некоторыми их особенностями, посмотрим, как настраиваются сети в Compose.

Определение сетей в конфигурации Compose

До сих пор наши контейнеры могли беспрепятственно взаимодействовать друг с другом благодаря созданию мостовой сети в момент запуска приложения Compose. Как показывают предыдущие примеры, контейнеры могут

взаимодействовать с другими службами Compose и обмениваться данными через псевдонимы:

```
$ docker network ls --filter name=chapter1_default
NETWORK ID     NAME               DRIVER    SCOPE
9ec7758d7050   chapter1_default   bridge    local
```

На первых порах этого может быть достаточно, однако в какой-то момент неизбежно появится желание дать сети другое имя.

Сделать это в Compose очень просто:

```
networks:
  task-manager-pubic-network:
    labels:
      - com.packtpub.compose.app=task-manager
```

Однако недостаточно просто создать сеть. К ней еще нужно подключить контейнеры. Давайте вернемся к нашему приложению и подключим службы к этой сети. После внесения некоторых изменений файл Compose должен выглядеть так:

```
// Chapter3/docker-compose.yaml
services:
  task-manager:
    build:
      context: ../Chapter2/.
      labels:
        - com.packtpub.compose.app=task-manager
    image: task-manager:0.1
    ...
    networks:
      - task-manager-pubic-network
    labels:
      - com.packtpub.compose.app=task-manager
  redis:
    image: redis
    ...
    networks:
      - task-manager-pubic-network
    labels:
      - com.packtpub.compose.app=task-manager
  redis-backup:
    image: bash
    ...
    networks:
      - task-manager-pubic-network
    labels:
      - com.packtpub.compose.app=task-manager
...
networks:
  task-manager-pubic-network:
    labels:
      - com.packtpub.compose.app=task-manager
```

Итак, мы создали сеть для приложения Compose и подключили к ней наши службы.

Теперь запустим приложение:

```
$ docker compose up
```

Поскольку наша служба подключена к сети, мы можем проверить, какие сети Docker существуют:

```
$ docker network ls
NETWORK ID     NAME                                  DRIVER
SCOPE
6a149c758fc2   bridge                                bridge
local
ce5aa144f0f9   chapter3_task-manager-pubic-network   bridge
local
4a6d6a3e8475   host                                  host
local
be28a5cd8a16   none                                  null
local
```

Все наши контейнеры были подключены к только что созданной сети, поэтому они все находятся в одной сети. Сеть по умолчанию, которая ранее создавалась в Compose, теперь отсутствует.

Добавление дополнительной сети в текущее приложение

К настоящему моменту мы определили мостовую сеть для использования службами, запустили наши контейнеры, и теперь они могут беспрепятственно взаимодействовать друг с другом. Также мы заметили, что мостовая сеть, создававшаяся ранее по умолчанию, теперь отсутствует. В предыдущих примерах мы видели и использовали только одну сеть, поэтому давайте попробуем добавить в Compose несколько сетей.

Цель этого упражнения – подключить нашу базу данных Redis. В реальной жизни базы данных часто располагаются в другой сети, отличной от сети, где находится приложение:

```
networks:
  redis-network:
    labels:
      - com.packtpub.compose.app=task-manager
```

Мы подключим базу данных Redis к этой сети:

```
 redis:
    image: redis
    ports:
```

```
      - 6379:6379
    entrypoint: ["redis-server","/usr/local/etc/redis/redis.conf"]
    volumes:
      - ./redis.conf:/usr/local/etc/redis/redis.conf
      - redis-data:/data
      - backup:/backup
    networks:
      - redis-network
    labels:
      - com.packtpub.compose.app=task-manager
```

Если теперь, снедаемые любопытством, вы попробуете запустить приложение, то получите ошибку:

```
{ "id": "8b171ce0-6f7b-4c22-aa6f-8b110c19f838",
  "name": "A task",
  "description": "A task that need to be executed at the timestamp specified",
  "timestamp": 1645275972000 }
```

Как и ожидалось, приложение на Go не имеет доступа к сети, к которой подключена служба Redis. Мы исправим эту проблему, подключив приложение `task-manager` к сети Redis:

```
  task-manager:
    build:
      context: ../Chapter2/.
      labels:
        - com.packtpub.compose.app=task-manager
    image: task-manager:0.1
    ports:
      - 8080:8080
    environment:
      - REDIS_HOST=redis:6379
    depends_on:
      - redis
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8080/ping"]
      interval: 10s
      timeout: 5s
      retries: 5
      start_period: 5s
    networks:
      - task-manager-pubic-network
      - redis-network
```

Итак, контейнер можно подключить к двум сетям. Благодаря этому у нас получилось связать контейнеры с приложением и с базой данных Redis.

Если выполнить GET-запрос, то в ответ должен вернуться положительный результат, как показано ниже:

```
curl --location --request GET 'localhost:8080/task/8b171ce0-
6f7b-4c22-aa6f-8b110c19f83a'
```

```
{"task":{
  "id":"8b171ce0-6f7b-4c22-aa6f-8b110c19f83a",
  "name":"A task",
  "description":"A task that need to be executed at the timestamp specified",
  "timestamp":1645275972000}
}
```

Итоги

В этой главе мы подготовили тома, подключили их к службам Compose и попробовали использовать одновременно в нескольких службах. Мы выяснили, как определяются сети в Docker, как их можно настраивать в Compose, как они работают и как подключить приложение к нескольким сетям.

Следующая глава будет посвящена командам Docker Compose. Мы углубимся в доступные команды, их назначение и поговорим о том, как они могут помочь в подготовке приложений Compose и управлении ими.

Глава 4

Выполнение команд Docker Compose

В предыдущей главе мы познакомились с особенностями использования сетей и томов Docker в приложении Compose. Настроив сети, мы смогли установить связь между компонентами приложения Compose, а настроив тома, мы упростили ввод-вывод и обеспечили переносимость и сохранность данных.

До сих пор для подготовки приложений и взаимодействий с компонентами Compose мы использовали различные команды Docker Compose. Поэтому в этой главе мы подробнее остановимся на командах Compose и их параметрах. Для начала мы посмотрим, какие команды доступны. Затем углубимся в команды подготовки образов и взаимодействия с контейнерами Compose. После этого перейдем к командам освобождения ресурсов. Наконец, после знакомства с командами освобождения ресурсов, помогающими в разработке приложений, мы погрузимся в команды мониторинга.

В этой главе рассматриваются следующие темы:

- введение в команды Compose;
- интерфейс командной строки Docker и команды Compose;
- команды инициализации;
- команды управления контейнерами;
- команды освобождения ресурсов;
- команды управления образами;
- команды мониторинга;
- другие команды.

Технические требования

Примеры кода для этой книги можно найти в репозитории GitHub по адресу https://github.com/PacktPublishing/A-Developer-s-Essential-Guide-to-Docker-Compose. Если мы будем обновлять код, то все обновления будут доступны в этом репозитории.

Введение в команды Compose

Compose имеет множество команд для взаимодействий с контейнерами, мониторинга приложений, а также для управления образами.

В этой главе мы рассмотрим команды, поддерживаемые Compose, попробуем применить их к целевому приложению, разберем их особенности и исследуем их внутреннюю работу.

Вот команды, на которых мы сосредоточимся:

- `build`;
- `create`;
- `up`;
- `images`;
- `pull`;
- `push`;
- `down`;
- `rm`;
- `logs`;
- `top`;
- `ps`;
- `events`.

Мы исследуем эти команды, посмотрим, какую выгоду можно извлечь из них, и используем их в приложении Compose.

Интерфейс командной строки Docker и команды Compose

Большинство этих команд покажутся знакомыми тем, кому доводилось пользоваться интерфейсом командной строки Docker. Команды Compose отличаются от команд Docker CLI тем, что последние работают со всеми компонентами Docker в системе, тогда как команды Compose ограничены контекстом конкретного приложения, указанного в файле `docker-compose.yaml`.

За кулисами команды Compose анализируют файл `docker-compose.yaml` и получают информацию о приложении. Вызовы API, отправляемые в Docker Engine, будут включать фильтры с учетом информации, полученной из файла, и взаимодействовать только со службами и ресурсами приложения. Например, команда `ps` вернет список не всех запущенных контейнеров, как это произошло бы при использовании интерфейса командной строки Docker, а только тех, которые должны запускаться в приложении Compose.

Настройка целевого приложения

Для целей демонстрации команд мы объединим все, что было сделано в предыдущих главах, и соберем приложение Compose. Приложение будет исполь-

зовать исходный код диспетчера задач, разработанного в главе 2 «Запуск первого приложения с помощью Compose», а также базу данных Redis.

В определении службы Compose используется код из предыдущей главы:

```
services:
  task-manager:
    build:
      context: ../Chapter2/.
    image: docker.io/library/chapter4_task-manager
    ports:
      - 8080:8080
    environment:
      - REDIS_HOST=redis:6379
  redis:
    image: redis
    ports:
      - 6379:6379
```

Итак, у нас есть список команд, на которых мы сосредоточимся, а также приложение, на примере которого мы будем исследовать работу этих команд. В следующем разделе мы сосредоточимся на командах, подготавливающих и выделяющих ресурсы для приложения.

Команды подготовки

Настроив файл приложения Compose, можно укомплектовать его необходимыми ресурсами. В этом разделе мы рассмотрим следующие команды:

- `build`;
- `create`;
- `up`.

Рассмотрим их поближе.

build

Как было показано выше, Compose избавляет нас от необходимости вручную создавать образ Docker и добавлять к нему метку. Compose может создать образ автоматически.

Вот как выполняется команда `build`, создающая образ:

```
$ docker compose build
=> => exporting layers
0.0s
...
=> => naming to docker.io/library/chapter4_task
manager        0.0s
$ docker images
```

```
REPOSITORY                      TAG       IMAGE ID
CREATED         SIZE
chapter4_task-manager           latest    58247e548811   35
hours ago    529MB
```

Команда `build` создает образ приложения диспетчера задач и добавляет в него соответствующую метку. Теперь мы перейдем к запуску приложения.

create

Команда `create` устарела, и в настоящее время вместо нее можно использовать команду `up`. Цель `create` – создать контейнеры, тома и сети, но без запуска контейнеров. Эти же действия выполняет команда `up --no-start`:

```
$ docker compose create
[+] Running 3/0
 ⠿ Network chapter4_default           Created
0.0s
 ⠿ Container chapter4-redis-1        Created
0.0s
 ⠿ Container chapter4-task-manager-1  Created
```

up

Команда `up` имеет множество применений и параметров. Вот, например, как можно заменить предыдущую команду `create` командой `up`:

```
$ docker compose up --no-start
[+] Running 3/3
 ⠿ Network chapter4_default          Created
0.0s
 ⠿ Container chapter4-task-manager-1  Created
0.0s
 ⠿ Container chapter4-redis-1         Created
$ docker ps -a
CONTAINER ID   IMAGE                                COMMAND
CREATED         STATUS                     PORTS      NAMES
3d1ad2f353d3   redis                                "docker-
entrypoint.s…"   3 seconds ago   Created
chapter4-redis-1
b6d1a5f16313   chapter4_task-manager                "/
task_manager"          3 seconds ago   Created
             chapter4-task-manager-1
```

Она создает контейнеры и сети, но не запускает их.

После создания ресурсов можно запустить приложение. Одним из полезных параметров команды `up` является параметр `-d` или `--detach`, активизиру-

ющий автономный режим. Автономный режим запускает приложение Compose и отсоединяет его от терминала:

```
$ docker compose up --detach
[+] Running 2/2
⠿ Container chapter4-task-manager-1  Started
0.3s
⠿ Container chapter4-redis-1         Started
```

Поскольку образ уже сконструирован и все необходимые ресурсы созданы, потребуем от команды не создавать образ, добавив параметр `--no-build`:

```
$ docker compose up --no-build --detach
[+] Running 3/3
⠿ Network chapter4_default           Created
0.0s
⠿ Container chapter4-redis-1         Started
0.3s
⠿ Container chapter4-task-manager-1  Started
```

При необходимости обновить службу можно обновить и удалить ресурсы. Например, в случае развертывания `task-manager-2` было бы желательно удалить старые версии контейнера.

Предположим, что мы решили переименовать службу:

```
services:
  task-manager-2:
...
```

Если теперь выполнить команду `up`, она выведет предупреждение о наличии устаревшего контейнера:

```
$ docker compose up
WARN[0000] Found orphan containers ([chapter4-task-
manager-2-1]) for this project. If you removed or renamed this
service in your compose file, you can run this command with the
--remove-orphans flag to clean it up.
```

Избавиться от него можно, добавив параметр `--remove-orphans`:

```
$ docker compose up --remove-orphans
[+] Running 3/0
⠿ Container chapter4-task-manager-1    Removed
          0.0s
⠿ Container chapter4-redis-1           Created
```

После запуска приложения с помощью команд подготовки, показанных выше, можно использовать команды Compose для взаимодействия с контейнерами, в которых выполняются службы Compose; эти команды сосредоточены исключительно на взаимодействиях с контейнерами.

Команды управления контейнерами

Команды управления контейнерами позволяют запускать, останавливать, перезапускать и уничтожать контейнеры.

exec

На протяжении всей книги мы выполняли команды с помощью Docker-команды `exec`. Docker Compose тоже предоставляет аналогичную команду. Разница в том, что команда `exec` в версии Compose запускает не контейнер, а службу. За кулисами она передает соответствующую команду в контейнер службы:

```
$ docker compose exec task-manager ls
go.mod    go.sum    main.go
```

Если проверить существующие контейнеры, мы не увидим других подготовленных контейнеров:

```
$ docker ps -a
CONTAINER ID   IMAGE                                  COMMAND
             CREATED          STATUS                       PORTS
NAMES
8e559f9bf6d1   chapter4_task-manager                  "/
task_manager"          18 seconds ago   Up 17 seconds
0.0.0.0:8080->8080/tcp   chapter4-task-manager-1
46ca9a239557   redis                                  "docker-
entrypoint.s…"   18 seconds ago   Up 17 seconds
0.0.0.0:6379->6379/tcp   chapter4-redis-1
```

run

Команда `run` может показаться похожей на `exec`, но, в отличие от последней, она запускает новый контейнер. Это позволяет запускать контейнер в окружении, предоставленном файлом Compose. Этот контейнер получит доступ к созданным нами томам и сетям:

```
$ docker compose run task-manager sh
/app #
```

Запустив другой терминал, можно проверить, был ли запущен контейнер:

```
$ docker ps -a
CONTAINER ID   IMAGE
COMMAND                  CREATED          STATUS
PORTS                    NAMES
b7920b959c98   host.docker.internal:5000/task-manager:0.1
"sh"                     23 minutes ago   Up 23 minutes
```

```
8080/tcp                chapter4_task-manager_run_7ad8bcc01122
...
b575f8bcd7d3   host.docker.internal:5000/task-manager:0.1
"/task_manager"          2 hours ago      Up 42 minutes
0.0.0.0:8080->8080/tcp   chapter4-task-manager-1
```

Как видите, команда создала новый контейнер с именем `Chapter4_task-manager_run_7ad8bcc01122`. При выходе контейнер будет остановлен.

pause

Команда `pause` в Docker приостанавливает процессы в указанном контейнере. Compose-команда `pause` приостанавливает контейнеры указанной службы:

```
$ docker compose pause task-manager
[+] Running 1/0
⠿ Container chapter4-task-manager-1  Paused
docker ps
CONTAINER ID
IMAGE                   COMMAND                  CREATED
STATUS                  PORTS                    NAMES
8e559f9bf6d1   chapter4_task-manager   "/task_
manager"          7 minutes ago   Up 7 minutes (Paused)
0.0.0.0:8080->8080/tcp   chapter4-task-manager-1
46ca9a239557   redis                   "docker-entrypoint.s…"
7 minutes ago   Up 7 minutes            0.0.0.0:6379->6379/tcp
chapter4-redis-1
```

unpause

Выше мы приостановили контейнер `task-manager`. Возобновить его работу можно с помощью команды `unpause`:

```
$ docker compose unpause task-manager
[+] Running 1/0
⠿ Container chapter4-task-manager-1  Unpaused
          0.0s
$ docker ps
CONTAINER ID   IMAGE
COMMAND                  CREATED         STATUS
PORTS                    NAMES
8e559f9bf6d1   chapter4_task-manager   "/task_
manager"          8 minutes ago   Up 8 minutes   0.0.0.0:8080-
>8080/tcp   chapter4-task-manager-1
46ca9a239557   redis                   "docker-entrypoint.s…"
8 minutes ago   Up 8 minutes   0.0.0.0:6379->6379/tcp
chapter4-redis-1
```

Как и ожидалось, контейнер возобновил выполнение.

start и stop

С помощью команды `stop` можно остановить работающие контейнеры. Команда `start` используется для запуска существующих контейнеров. Остановленный контейнер не удаляется из памяти. Если войти в командную оболочку контейнера, создать файл, а затем остановить контейнер командой `stop`, то после запуска контейнера командой `start` контейнер сможет использовать этот файл:

```
$ docker compose ps
NAME                        COMMAND             SERVICE
STATUS              PORTS
chapter4-task-manager-1   "/task_manager"     task-
manager        running             0.0.0.0:8080->8080/tcp
$ docker compose exec task-manager sh -c "echo test > text.txt"
$ docker compose stop task-manager
[+] Running 2/0
⠿ Container chapter4-task-manager-1                 Stopped
0.1s
```

Контейнер остановлен. Ожидается, что после запуска контейнера наш файл окажется на месте:

```
$ docker compose start task-manager
[+] Running 1/1
Container chapter4-task-manager-1  Started
$ docker compose exec task-manager sh -c "cat text.txt"
test
```

restart

Команда `restart` перезапустит службу Compose. Даже если служба остановлена, она запустится:

```
$ docker compose restart task-manager
[+] Running 1/1
 Container chapter4-task-manager-1  Started
```

Давайте остановим службу, перезапустим ее и посмотрим, что из этого получится:

```
$ docker compose stop task-manager
[+] Running 1/0
Container chapter4-task-manager-1  Stopped
$ docker compose restart task-manager
[+] Running 1/1
Container chapter4-task-manager-1  Started
0.2s
```

Независимо от предыдущего состояния приложение запускается.

kill

Команду `kill` можно использовать для уничтожения контейнеров служб:

```
$ docker compose kill task-manager
```

ps

Команда `compose ps` в Compose похожа на команду `docker ps`. Ключевое отличие состоит в том, что цель команды `compose ps` фильтруется с учетом настроек в файле Compose:

```
$ docker compose ps
NAME                        COMMAND             SERVICE
STATUS              PORTS
chapter4-task-manager-1   "/task_manager"     task-
manager        running             0.0.0.0:8080->8080/tcp
```

Добавив параметр `--services`, можно вывести только запущенные службы:

```
$ docker compose ps --services
task-manager
```

Добавив параметр `--quiet`, можно отобразить только идентификатор контейнера:

```
$ docker compose ps --quiet
9b612d9ed210ab96d2b340ed8840781c80288097e1af6a96ed208d6b6d1
fb42a
```

При желании можно даже вывести остановленные контейнеры, если один раз запустить команду в службе диспетчера задач:

```
$ docker compose run task-manager sh
/app # exit
```

С помощью `ps -a` можно показать остановленный контейнер, созданный командой `run`:

```
$ docker compose ps -a
NAME                                  COMMAND
SERVICE             STATUS              PORTS
chapter4-task-manager-1                "/task_manager"
task-manager        running             0.0.0.0:8080->8080/tcp
chapter4_task-manager_run_91e7eec48f5d   "sh"
task-manager        exited (0)
```

В этом разделе мы создали наше базовое приложение Compose. Используя команды Compose, создали образы, необходимые приложению. Исследовали дополнительные параметры команд запуска. Затем опробовали команды

Compose взаимодействий с контейнерами, развернутыми при запуске приложения. Мы применили команды к существующим контейнерам, а также развернули контейнеры, используя существующие ресурсы окружения. В следующем разделе мы рассмотрим команды, освобождающие ресурсы, полученные с помощью Compose.

Команды освобождения ресурсов

Часто, работая с Docker, мы выделяем различные ресурсы и используем их в процессе разработки. Удаление образов, контейнеров, а также сетей и томов может стать довольно утомительной задачей. Compose предлагает более простой способ работы со всеми этими ресурсами.

down

Запустим приложение в подключенном режиме:

```
$ docker compose up
[+] Running 2/0
 ⠿ Container chapter4-task-manager-1 Running              0.0s
 ⠿ Container chapter4-redis-1 Running                     0.0s
```

Нажав комбинацию **Ctrl**+**D**, мы выйдем из Compose, сообщения перестанут отображаться, и службы перестанут работать:

```
Gracefully stopping... (press Ctrl+C again to force)
[+] Running 2/2
 ⠿ Container chapter4-task-manager-1  Stopped
0.1s
 ⠿ Container chapter4-redis-1         Stopped
0.1s
canceled
```

Судя по сообщениям в терминале, приложение остановилось. Но ресурсы, выделенные ему, все еще остаются занятыми. В этом легко убедиться с помощью команд `images`, которая выводит созданные образы, и `ps`, которая выводит список запущенных контейнеров:

```
$ docker compose images
Container                Repository              Tag
Image Id            Size
chapter4-redis-1         redis
latest              f16c30136ff3        107MB
chapter4-task-manager-1   chapter4_task-manager
latest              58247e548811        529MB
$ docker compose ps
NAME                     COMMAND                  SERVICE
```

```
STATUS              PORTS
chapter4-redis-1          "docker-entrypoint.s…"   redis
exited (0)
chapter4-task-manager-1   "/task_manager"          task-
manager        exited (2)
$ docker network ls
NETWORK ID
NAME                                  DRIVER    SCOPE
...
ed28fb3286ab   chapter4_
default                       bridge    local
```

При повторном запуске команды `docker compose up` она будет использовать те же ресурсы.

В какой-то момент может появиться желание удалить все эти артефакты из нашей системы и освободить место для новых разработок. В таком случае можно использовать команду `docker compose down`. Она освободит ресурсы, занятые приложением Compose:

```
$ docker compose down
[+] Running 3/2
⠿ Container chapter4-task-manager-1  Removed
0.1s
⠿ Container chapter4-redis-1         Removed
0.1s
⠿ Network chapter4_default           Removed
0.0s
$ docker compose ps
NAME                COMMAND             SERVICE
STATUS              PORTS
$  docker network ls|grep chapter4
```

Глядя на запущенные команды, можно подумать, что все ресурсы были освобождены. Однако остается еще кое-что, а именно образы, созданные ранее с помощью `docker compose build`.

Команда `down` позволяет удалить образ, созданный при запуске приложения:

```
$ docker compose down --rmi local
[+] Running 4/2
⠿ Container chapter4-redis-1         Removed
0.2s
⠿ Container chapter4-task-manager-1  Removed
0.2s
⠿ Image chapter4_task-manager        Removed
0.0s
⠿ Network chapter4_default           Removed
0.0s
```

Параметр `local` относится к образам, созданным с учетом информации из файла Compose и не имеющим имен. В нашем случае, поскольку у нас есть тег `image: Chapter4_task-manager:latest`, мы должны использовать параметр `all`:

```
$ docker compose down --rmi all
[+] Running 5/0
 ⠿ Container chapter4-task-manager-1              Removed
0.0s
 ⠿ Container chapter4-redis-1                     Removed
0.0s
 ⠿ Image redis                                    Removed
0.0s
 ⠿ Image docker.io/library/chapter4_task-manager  Removed
0.0s
 ⠿ Network chapter4_default                       Removed
0.0s
```

Имейте в виду, что параметр `all` также удалит локально кешированные образы, как можно видеть в предыдущем примере.

rm

Команда `down`, представленная в предыдущем разделе, удалила все контейнеры. Используя `rm`, можно удалять контейнеры по отдельности. Представьте, что у нас есть плохо работающий контейнер:

```
services:
  failed-manager:
    build:
      context: ../Chapter2/.
    entrypoint: ["/no-such-command"]
  task-manager:
    build:
      context: ../Chapter2/.
    image: chapter4_task-manager:latest
    ports:
      - 8080:8080
    environment:
      - REDIS_HOST=redis:6379
  redis:
    image: redis
    ports:
      - 6379:6379
```

При попытке запустить службу `failed-manager` она завершится ошибкой:

```
$ docker compose up -d
[+] Running ¾
 ⠿ Network chapter4_default             Created
 0.0s
 ⠿ Container chapter4-failed-manager-1  Starting
                              0.5s
 ⠿ Container chapter4-redis-1           Started
 0.4s
```

```
⠿ Container chapter4-task-manager-1    Started
                          0.4s
Error response from daemon: OCI runtime create failed:
container_linux.go:380: starting container process caused:
exec: "/no-such-command": stat /no-such-command: no such file
or directory: unknown
```

Здесь можно видеть сообщение об ошибке. Выполнив команду `ps`, можно увидеть, что один контейнер запущен, а другой только лишь создан:

```
$ docker compose ps
NAME                          COMMAND                  SERVICE
STATUS              PORTS
chapter4-failed-manager-1     "/no-such-command"       failed-
manager      created             8080/tcp
chapter4-redis-1              "docker-entrypoint.s…"   redis
running             0.0.0.0:6379->6379/tcp
chapter4-task-manager-1       "/task_manager"          task-
manager        running             0.0.0.0:8080->8080/tcp
```

Удалим плохой контейнер:

```
$ docker compose rm failed-manager
? Going to remove chapter4-failed-manager-1 Yes
[+] Running 1/0
⠿ Container chapter4-failed-manager-1  Removed
```

Принудительно удалить контейнер можно, используя флаг `-f` или `--force`:

```
$ docker compose rm failed-manager --force
Going to remove chapter4-failed-manager-1
[+] Running 1/0
⠿ Container chapter4-failed-manager-1  Removed
```

Параметр `force` предотвращает вывод запроса на подтверждение.

После запуска приложения и взаимодействия с созданными контейнерами мы остановили приложение, а также освободили созданные ресурсы. Одним из таких ресурсов, созданных в процессе сборки, был образ приложения. Теперь мы углубимся в управление образами с помощью Compose.

Команды управления образами

Compose предоставляет различные варианты эффективного использования образов, избавляя от необходимости создавать образы по отдельности и использовать их в приложении. Создание образов, добавление меток к ним, а также отправка их в репозиторий – все это можно сделать с помощью Compose. Благодаря этому все усилия по разработке можно сосредоточить в одном месте.

Список образов

Наше приложение пока имеет всего две службы, выполняемые в контейнерах. Получить список образов, используемых контейнерами, можно с помощью команды `docker compose images`:

```
$ docker compose images
Container                 Repository                Tag
Image Id            Size
chapter4-redis-1          redis                     6.2.6
23d787aaa419        107MB
chapter4-task-manager-1   chapter4_failed-manager
latest              58247e548811        529MB
```

Вывод можно сделать менее подробным, если использовать параметр `--quiet`:

```
$ docker compose images --quiet
58247e548811b8812d48467436bc07ed40b4a7d4cd8328e57234d465ef18914a
23d787aaa419ab884dd8682dca3153506f4dd00aba2ca9cd5953e94cae36bd7d
```

Извлечение образов

С помощью команды `images` мы смогли получить список образов, используемых приложением Compose. А что, если мы поменяем образ? В таком случае нам нужно извлечь образы перед запуском. Для этого Compose предоставляет команду `pull`, которая извлекает указанный образ.

Итак, давайте укажем другой образ `redis`:

```
services:
  redis:
    image: redis:6.2.6
```

Вместо использования команды `up` и извлечения образа по умолчанию воспользуемся командой `pull`:

```
$ docker compose pull --ignore-pull-failures --parallel
[+] Running 2/3
 ⠿ failed-manager Skipped                          0.0s
 ⠿ redis Pulled                                    1.2s
 ⠿ task-manager Error                              1.5s
Pulling task-manager: Error response from daemon: pull access
denied for chapter4_task-manager, repository does not exist or
may require 'docker login': denied: requested access to the
resource is denied
$ echo $?
0
```

Как видите, выполнив команду `pull`, мы извлекли указанные старые образы Redis. Также из-за отсутствия диспетчера задач в виде образа в реестре, который создан локально, его не удалось извлечь. Однако эта неудача

не заблокировала операцию, поэтому команда вернула `0`. Такое поведение обеспечил параметр `--ignore-pull-failures`. Кроме того, параметр `--parallel` обеспечил параллельное извлечение образов.

Отправка образов

С помощью Compose также можно выполнить отправку образов в реестр Docker. Для демонстрации настроим реестр с помощью Compose.

Локальный реестр Docker в Compose

Чтобы отправить образ Docker, необходим реестр. Мы развернем реестр с помощью Compose и настроим наше приложение Compose для использования этого реестра.

Наш файл определения реестра Compose выглядит следующим образом:

```
services:
  registry:
    image: registry:2
    ports:
      - 5000:5000
```

Реестр будет работать на порту 5000, поэтому нужно убедиться, что этот порт доступен в системе. Имейте в виду, что это небезопасный реестр, поэтому нужно отредактировать конфигурацию демона Docker, чтобы явно указать незащищенный реестр.

Местоположение конфигурационного файла `daemon.json` зависит от операционной системы:

- **Linux**: `/etc/docker/daemon.json`;
- **Windows**: `C:\ProgramData\docker\config\daemon.json`;
- **macOS**: `~/.docker/daemon.json`.

Отыскав файл, добавим в него следующую запись:

```
  "insecure-registries" : [
    "host.docker.internal:5000"
   ],
```

После редактирования файл должен выглядеть так:

```
{
...
  "insecure-registries" : [
    "host.docker.internal:5000"
   ],
  "builder" : {
...
  }
}
```

Отправка в локальный реестр

Теперь можно отправить образ `task-manager` в локальный реестр, хотя точно так же можно использовать любой доступный реестр. Поскольку реестр является локальным и управляется движком Docker, мы сошлемся на него с помощью DNS-записи `host.docker.internal`:

```
services:
  task-manager:
    build:
      context: ../Chapter2/.
    image: host.docker.internal:5000/task-manager:0.1
    ports:
      - 8080:8080
    environment:
      - REDIS_HOST=redis:6379
  redis:
    image: redis
    ports:
      - 6379:6379
```

Теперь можно собрать образ:

```
$ docker compose build
[+] Building 2.0s (17/17) FINISHED

=> [internal] load build definition from Dockerfile
...
=> => naming to host.docker.internal:5000/task-manager:0.1
...
```

и отправить его в реестр:

```
$ docker compose push
[+] Running 1/13
 redis Skipped
0.0s
 Pushing task-manager: 98ca4aa0fc4c Pushed
15.8s
...
 Pushing task-manager: 590efbee44c0 Pushed
15.8s
```

Мы собрали образы с помощью Compose и отправили их в созданный нами реестр. Мы получили список используемых образов и извлекли новые образы, указанные в файле Compose. Следующий раздел будет посвящен мониторингу приложения Compose.

Команды мониторинга

Часто в повседневной разработке необходим мониторинг приложения, чтобы убедиться в его правильной работе. В приложении могут наблюдаться самые разные проблемы – от использования слишком большого количества ресурсов до неуловимых ошибок или даже перезапусков приложения. В этих случаях решающую роль играют команды мониторинга.

logs

Журналы дают возможность просматривать сообщения, выводимые приложениями Compose, работающими в нашей системе. Если запустить команду `up` в автономном режиме, мы не сможем увидеть никаких сообщений:

```
$ docker compose up -d
[+] Running 2/2
 ⠿ Network chapter4_default           Created
0.0s
 ⠿ Container chapter4-task-manager-1  Started
```

Тем не менее приложения продолжают генерировать сообщения, и мы можем получить их:

```
$ docker compose logs
chapter4-task-manager-1  | [GIN-debug] [WARNING] Creating an
Engine instance with the Logger and Recovery middleware already
attached.
...
chapter4-task-manager-1  | [GIN-debug] Listening and serving
HTTP on :8080
```

Команда `logs` отображает все сообщения, сгенерированные приложением и отправленные в терминал. Эта команда имеет также дополнительный параметр `--follow`. Благодаря этому мы можем наблюдать за генерируемыми сообщениями и выводить их на экран:

```
$ docker compose logs -f
chapter4-task-manager-1  | [GIN-debug] [WARNING] Creating an
Engine instance with the Logger and Recovery middleware already
attached.
...
chapter4-task-manager-1  | [GIN-debug] Listening and serving
HTTP on :8080
```

Кроме того, команда `logs` имеет такие параметры, как `--no-color`, отключающий вывод сообщений в цвете; `--timestamps`, отображающий время появления сообщения; и `--tail`, ограничивающий количество выводимых сообщений:

```
$ docker compose logs -f --timestamps --tail="3" --no-color
chapter4-task-manager-1  | 2022-03-26T07:50:13.846624301Z
[GIN-debug] [WARNING] You trusted all proxies, this is NOT
safe. We recommend you to set a value.
chapter4-task-manager-1  | 2022-03-26T07:50:13.846625967Z
Please check https://pkg.go.dev/github.com/gin-gonic/
gin#readme-don-t-trust-all-proxies for details.
chapter4-task-manager-1  | 2022-03-26T07:50:13.846627551Z
[GIN-debug] Listening and serving HTTP on :8080
```

top

`logs` – действительно полезная команда, но она выводит только сообщения, генерируемые приложением. С помощью команды `top` можно отобразить список запущенных процессов Compose:

```
$ docker compose top
chapter4-task-manager-1
UID    PID    PPID   C    STIME   TTY   TIME       CMD
root   7913   7887   0    07:50   ?     00:00:00   /task_manager
```

Команда `top` в Docker ограничена указанным контейнером. Команда `top` в Compose следит за контейнерами, перечисленными в файле `docker-compose.yaml`.

events

`events` – это команда Compose, действующая подобно команде `events` в Docker. Команда `events` отслеживает события, происходящие в приложении Compose:

```
$ docker compose events
...
```

Остановим контейнер:

```
$ docker stop chapter4-task-manager-1
chapter4-task-manager-1
```

На вкладке `events` мы должны увидеть сообщение:

```
$ docker compose events
...
2022-03-26 10:05:45.233256 container stop
2cda5390558dc9d6b12bae95d6c65b23a6d95b36a5279677834f81af1d62
69f2 (name=chapter4-task-manager-1, com.packtpub.compose.app
=task-manager, image=host.docker.internal:5000/task-
manager:0.1)
```

Каждая операция, созданная командой `events` в Docker и влияющая на компоненты приложения, будет выведена на экран.

Мы только что рассмотрели команды мониторинга. С их помощью можно отслеживать события, происходящие в приложении, определять используемые ресурсы, а также устранять проблемы с помощью журналов.

Другие команды

Существуют команды, которые могут помочь вывести имеющуюся информацию.

help

`help` выводит инструкции по использованию доступных команд.

version

Как было показано в предыдущих главах, `version` отображает версию Compose:

```
$ docker compose version
Docker Compose version v2.2.3
```

port

`port` выводит номер порта, привязанного к приложению:

```
$ docker compose port task-manager 8080
0.0.0.0:8080
```

config

`config` используется для проверки конфигурации Compose и выбора информации для вывода:

```
$ docker-compose config
services:
  redis:
    image: redis
    ports:
    - published: 6379
      target: 6379
  task-manager:
    build:
      context: /path/to/Chapter2
```

```
    environment:
      REDIS_HOST: redis:6379
    image: host.docker.internal:5000/task-manager:0.1
    ports:
    - published: 8080
      target: 8080
version: '3.9'
```

Ее также можно использовать для отображения только служб:

```
$ docker-compose config --services
task-manager
redis
```

В качестве альтернативы она может отобразить список томов:

```
$ docker-compose config --volumes
```

Познакомившись с предыдущими командами, мы теперь имеем полный набор команд Compose, полезных в повседневной разработке.

Итоги

В этой главе мы рассмотрели доступные команды Compose и их параметры. Мы использовали команды, которые помогли нам подготовить наше многоконтейнерное приложение, а также команды для взаимодействия с контейнерами приложения. Мы рассмотрели функциональные возможности для работы с образами, позволяющие извлекать, создавать и развертывать образы из реестра Docker без выполнения посторонних команд, не относящихся к Compose. Запустив приложения и оценив существующие параметры команд, мы затем рассмотрели возможность мониторинга приложения путем отслеживания журналов или перехвата событий Docker, а также понаблюдали за активностью процессов Compose с помощью `top`.

В следующих главах мы перейдем к более конкретным концепциям Compose, способным помочь нам в повседневной разработке, и займемся созданием приложения на основе микросервисов с помощью Compose, понаблюдаем за его работой, разобьем его на модули и развернем с помощью CI/CD.

Часть II

ПОВСЕДНЕВНАЯ РАЗРАБОТКА С ПОМОЩЬЮ DOCKER COMPOSE

Эта часть знакомит с более продвинутыми концепциями Docker Compose. Здесь с помощью Compose мы настроим базы данных для повседневного использования. Затем с помощью Docker организуем сетевые взаимодействия между микросервисами. Мы будем запускать полные стеки технологий локально и моделировать промышленные окружения и, наконец, расширим задания CI/CD с помощью Docker Compose.

Данная часть включает следующие главы:

- главу 5 «Подключение микросервисов друг к другу»;
- главу 6 «Мониторинг служб с помощью Prometheus»;
- главу 7 «Комбинирование файлов Compose»;
- главу 8 «Локальное моделирование промышленного окружения»
- главу 9 «Создание расширенных заданий CI/CD».

Глава 5

Подключение микросервисов друг к другу

Предыдущая часть была посвящена началу работы с Docker. Установив Docker на рабочую станцию, мы начали ближе знакомиться с Docker Compose и приемами его использования. Мы научились комбинировать образы Docker с помощью Compose и запускать многоконтейнерные приложения. Успешно освоив запуск многоконтейнерных приложений, мы перешли к изучению более сложных концепций, таких как тома и сети Docker. Тома позволяют организовать хранение и совместное использование данных, а сети помогают изолировать определенные приложения и получать к ним доступ только через определенную сеть. В процессе знакомства мы постепенно перешли от использования команд Docker CLI к командам Docker Compose. Начав использовать команды Compose, мы переключили наше внимание на приложение Compose и получили возможность взаимодействовать с контейнерами, наблюдать и управлять ими с помощью команд администрирования, а также сосредоточить наши действия на ресурсах Compose.

Поскольку мы уже научились создавать многоконтейнерные приложения, теперь мы можем перейти к более сложным сценариям использования Compose. Современные приложения становятся все сложнее и сложнее. Это вынуждает разбивать приложение на несколько частей для лучшего масштабирования или для разработки несколькими командами. Микросервисы – это новая нормальность. Разделив задачу на более мелкие части, команды могут извлечь выгоду из ускорения их решения. Кроме того, микросервисы могут помочь управлять производительностью и масштабированием наиболее важных частей приложения.

Концепция микросервисов существовала задолго до появления Docker. Она сыграла решающую роль в массовом внедрении микросервисов. Способы изоляции и развертывания служб, упакованных необходимыми инструментами, позволяют сократить затраты, связанные с развертыванием микросервисов на виртуальной или физической машине.

С помощью микросервисов приложение разбивается на несколько частей. Связь между службами имеет решающее значение. Существуют общедоступные микросервисы – точки входа, и микросервисы, доступные только внутри приложения.

В этой главе мы сосредоточимся на приложении диспетчера задач, представленном в главе 2 «Запуск первого приложения с помощью Compose», и преобразуем его в приложение на основе микросервисов. Мы добавим микросервис геолокации, который будет использоваться диспетчером задач. Реализовав эту службу, мы добавим ее в сеть, доступ к которой будет иметь только приложение диспетчера задач. После этого мы добавим еще одну службу, которая будет генерировать аналитику на основе данных, передаваемых в Redis.

В этой главе рассматриваются следующие темы:

- реализация микросервиса определения местоположения;
- добавление микросервиса определения местоположения в Compose;
- добавление микросервиса определения местоположения в приватную сеть;
- выполнение запросов к микросервису определения местоположения;
- потоковая передача событий задач;
- добавление микросервиса обработки событий задач.

Технические требования

Примеры кода для этой книги можно найти в репозитории GitHub по адресу https://github.com/PacktPublishing/A-Developer-s-Essential-Guide-to-Docker-Compose. Если мы будем обновлять код, то все обновления будут доступны в этом репозитории.

Микросервис определения местоположения

В этой главе мы вернемся к приложению диспетчера задач, представленному в главе 2 «Запуск первого приложения с помощью Compose», и расширим его функциональность, добавив в задачи местоположение, где они должны выполняться. Каждой задаче будет назначаться место. Приложение будет запоминать местоположения, назначенные последней задаче, и рекомендовать их при создании следующей.

Службу определения местоположения мы реализуем как микросервис. Она ничего не будет посылать диспетчеру задач и будет иметь свой API. Для простоты мы используем тот же язык программирования, что и прежде, – Golang, а также ту же базу данных Redis.

Начнем с экземпляра Redis. Поскольку мы решили использовать Compose, конфигурация будет выглядеть так:

```
services:
  redis:
    image: redis
```

Мы можем запустить его в автономном режиме:

```
$ docker compose -f redis.yaml up -d
```

Мы создадим проект службы определения местоположения и добавим зависимости `gin` и `redis-go`.

Поскольку наш стек технологий не изменился, мы должны выполнить те же шаги по инициализации, которые выполняли в главе 2.

Вот эти шаги:

```
go mod init location_service
go get github.com/gin-gonic/gin
go get github.com/go-redis/redis/v8
```

После этого мы создадим файл `main.go`, содержащий каркас приложения. Как и раньше, мы будем использовать фреймворк `gin` и базу данных Redis, поэтому мы должны применять те же вспомогательные методы, которые вы можете найти в исходном коде примеров на GitHub: https://github.com/PacktPublishing/A-Developer-s-Essential-Guide-to-Docker-Compose/blob/main/Chapter5/location-service/main.go.

После создания проекта можно приступать к логике приложения. Важной частью нашей службы является модель местоположения. Модель местоположения будет содержать такую информацию, как уникальный идентификатор этого местоположения, долготу и широту, название местоположения и его описание. Поскольку предполагается использовать REST API, модель местоположения будет преобразована в JSON и возвращаться в ответ на вызовы API. Вот как выглядит наша модель местоположения:

```
type Location struct {
    Id          string  `json:"id"`
    Name        string  `json:"name"`
    Description string  `json:"description"`
    Longitude   float64 `json:"longitude"`
    Latitude    float64 `json:"latitude"`
}
```

Эта служба базируется на идее геолокации, поэтому нам нужно выбрать правильные структуры данных Redis. Модель местоположения может быть представлена в виде хеша `hmset`, позволяющего получить объект по ключу. Используя префикс и идентификатор объекта (`location:id`), можно создать несколько объектов местоположений. Кроме того, функциональные возможности `hmset` позволяют получать отдельные атрибуты объектов.

Еще один важный аспект местоположения – расстояние. Было бы желательно иметь возможность получать местоположения из нашей базы данных и расстояния между ними. Для этой цели Redis предоставляет метод Geohash. Используя `GeoAdd`, мы будем добавлять местоположения в отсортированное

множество. Хеш генерируется на основе широты и долготы и используется различными функциями Geohash, поддерживающими работу с отсортированным множеством. Это позволяет, например, обнаруживать местоположения, находящиеся на определенном расстоянии от определенной точки, или даже вычислять расстояние между двумя местоположениями, хранящимися в отсортированном наборе. Учитывая вышесказанное, мы должны сохранить местоположение в хеш и добавить запись в отсортированное множество с помощью `GeoAdd`.

Вот как выглядит функция, сохраняющая местоположение:

```
// Chapter5/location-service/main.go:177
func persistLocation(c context.Context, location Location)
error {
    hmset := client.HSet(c,
        fmt.Sprintf(locationIdFormat, location.Id), "Id",
        location.Id, "Name",
        location.Name, "Description",
        location.Description, "Longitude",
        location.Longitude, "Latitude",
        location.Latitude)
    if hmset.Err() != nil {
        return hmset.Err()
    }
    geoLoc := &redis.GeoLocation{
        Longitude: location.Longitude,
        Latitude: location.Latitude,
        Name: location.Id
    }
    gadd := client.GeoAdd(c, "locations", geoLoc)
    if gadd.Err() != nil {
        return gadd.Err()
    }
    return nil
}
```

А вот метод, извлекающий местоположение из хеша:

```
// Chapter5/location-service/main.go:153
func fetchLocation(c context.Context, id string) (*Location,
error) {
    hgetAll := client.HGetAll(c, fmt.Sprintf(locationIdFormat, id))
    if err := hgetAll.Err(); err != nil {
        return nil, err
    }
    ires, err := hgetAll.Result()
    if err != nil {
        return nil, err
    }
    if l := len(ires); l == 0 {
        return nil, nil
    }
    latitude, _ := strconv.ParseFloat(ires["Latitude"], 64)
```

```
    longitude, _ := strconv.ParseFloat(ires["Longitude"], 64)
    location := Location{
        Id: ires["Id"],
        Name: ires["Name"],
        Description: ires["Description"],
        Longitude: longitude,
        Latitude: latitude
    }
    return &location, nil
}
```

Поскольку мы хотели бы получить существующие местоположения, находящиеся на определенном расстоянии от заданного местоположения, мы будем использовать пространственные функции Redis. В своей реализации мы используем метод `GEORADIUS`, который будет применяться к множеству, созданному с помощью `GEOADD`:

```
// Chapter5/location-service/main.go:124
[...]
func nearByLocations(c context.Context, longitude float64,
                     latitude float64, unit string, distance float64)
                    ([] LocationNearMe, error)
{
    var locationsNearMe []LocationNearMe = make([] LocationNearMe, 0)
    query := &redis.GeoRadiusQuery{Unit: unit, WithDist: true,
                                   Radius: distance, Sort: "ASC"}
    geoRadius := client.GeoRadius(c, "locations", longitude, latitude, query)
    if err := geoRadius.Err(); err != nil {
        return nil, err
    }
    geoLocations, err := geoRadius.Result()
    if err != nil {
        return nil, err
    }
    for _, geoLocation := range geoLocations {
        if location, err := fetchLocation(c, geoLocation.Name); err != nil
        {
            return nil, err
        } else {
            locationsNearMe = append(locationsNearMe, LocationNearMe{
                Location: *location,
                Distance: geoLocation.Dist,
            })
        }
    }
    return locationsNearMe, nil
}
[...]
```

Этот метод принимает координаты некоторого местоположения и предельное расстояние и возвращает местоположения, находящиеся не дальше предельного расстояния, в порядке возрастания удаленности.

Теперь, реализовав основные методы, создадим REST API с помощью `gin`:

```
// Chapter5/location-service/main.go:49
[...]
r.GET("/location/:id", func(c *gin.Context) {
    id := c.Params.ByName("id")

    if location, err := fetchLocation(c.Request.Context(), id); err != nil {
        [...]
    } else if location == nil {
        [...]
    } else {
        [...]
    }

})

r.POST("/location", func(c *gin.Context) {
    var location Location

    [...]
    if err := persistLocation(c, location); err != nil {
        c.JSON(http.StatusInternalServerError,
               gin.H{"location": location,
                     "created": false,
                     "message": err.Error()})
        return
    }

    [...]
})

r.GET("/location/nearby", func(c *gin.Context) {
    [...]

    if locationsNearMe, err := nearByLocations(c, longitude,
latitude, unit, distance); err != nil {
        c.JSON(http.StatusInternalServerError,
               gin.H{"message": err.Error()})
        return
    } else {
        c.JSON(http.StatusOK,
               gin.H{"locations": locationsNearMe})
    }

})
```

Теперь запустим приложение и выполним несколько запросов:

```
$ go run main.go
```

Зададим местоположение с помощью `curl`:

```
$ curl --location --request POST 'localhost:8080/location/' \
--header 'Content-Type: application/json' \
```

```
--data-raw '{
        "id":"0c2e2081-075d-443a-ac20-40bf3b320a6f",
        "name": "Liverpoll Street Station",
        "description": "Station for Tube and National Rail",
        "longitude": -0.082966,
        "latitude": 51.517336
}'

{"created":true,
 "location":{"id":"0c2e2081-075d-443a-ac20-40bf3b320a6f",
             "name":"Liverpoll Street Station",
             "description":"Station for Tube and National Rail",
             "Longitude":-0.082966,
             "Latitude":51.517336},
             "message":"Location Created Successfully"}
```

Найдем местоположение, используя его идентификатор:

```
$ curl --location --request GET \
  'localhost:8080/location/0c2e2081-075d-443a-ac20-40bf3b320a6f'

{"location":{"id":"0c2e2081-075d-443a-ac20-40bf3b320a6f",
             "name":"Liverpoll Street Station",
             "description":"Station for Tube and National Rail",
             "Longitude":-0.082966,
             "Latitude":51.517336}}
```

Наконец, местоположение, находящееся не далее указанного расстояния от заданной точки. Поскольку было бы желательно знать расстояние до каждого местоположения, нам нужно изменить модель. Новая модель будет содержать само местоположение, а также вычисленное расстояние до него:

```
type LocationNearMe struct {
    Location Location `json:"location"`
    Distance float64 `json:"distance"`
}
```

Получим список местоположений рядом с заданной точкой:

```
curl --location --request GET 'localhost:8080/location/
nearby?longitude=-0.0197&latitude=51.5055&distance=5&unit=km'

{"locations":[{"location":{"id":"0c2e2081-075d-443a-ac20-40bf3b320a6f",
                           "name":"Liverpoll Street Station",
                           "description":"Station for Tube and National Rail",
                           "Longitude":-0.082966,
                           "Latitude":51.517336},
                           "distance":4.5729}]}
```

В помощь с предыдущими командами можно использовать коллекцию Postman, доступную в репозитории книги (https://github.com/PacktPublishing/A-Developer-s-Essential-Guide-to-Docker-Compose/blob/main/Chapter5/location-service/Location%20Service.postman_collection.json).

В этом разделе мы создали наш первый микросервис, сохранили несколько местоположений в Redis и использовали пространственные возможности этой базы данных для поиска существующих местоположений. Служба диспетчера задач должна взаимодействовать со службой определения местоположения посредством интерфейса REST. В следующем разделе мы упакуем приложение в образ Docker и создадим приложение Compose.

Добавление службы определения местоположения в Compose

Мы реализовали службу, добавили несколько местоположений и попробовали выполнять пространственные запросы. Теперь нам нужно упаковать приложение с помощью Docker и запустить его под управлением Compose.

Сначала создадим файл Dockerfile. Для этого выполним те же шаги, что и в предыдущей главе:

1. Добавим файл Dockerfile.
2. Создадим образ с помощью Compose.

Вот содержимое файла Dockerfile для службы определения местоположения:

```
# syntax=docker/dockerfile:1
FROM golang:1.17-alpine

RUN apk add curl

WORKDIR /app

COPY go.mod ./
COPY go.sum ./

RUN go mod download

COPY *.go ./

RUN go build -o /location_service

EXPOSE 8080

CMD [ "/location_service" ]
```

Dockerfile на месте, и теперь можно использовать его в Compose. Для тестирования приложения нам понадобятся Redis и образ нашего приложения.

Содержимое файла `docker-compose.yaml` на данном этапе должно выглядеть так:

```
services:
  location-service:
    build:
      context: ./location-service
```

```
    image: location-service:0.1
    environment:
      - REDIS_HOST=redis:6379
    depends_on:
      - redis
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8080/ping"]
      interval: 10s
      timeout: 5s
      retries: 5
      start_period: 5s
  redis:
    image: redis
```

Обратите внимание на отсутствие раздела `ports`. Это связано с тем, что служба определения местоположения будет использоваться только внутри приложения, соответственно, доступ к ней должны иметь только другие контейнеры, размещенные в Compose.

После запуска приложения Compose будет создан образ с именем `location-service`. Образ Redis останется прежним.

У нас получилось упаковать микросервис `location-service` с помощью Docker и запустить его с помощью Compose. Теперь нам нужно вновь заняться настройкой сетей в приложении Compose.

Добавление сети для микросервиса местоположения

Теперь мы должны настроить сеть, которая будет использоваться приложением вместо сети по умолчанию. Назовем эту сеть `geolocation-network`. Также нам нужна сеть для Redis. Давайте добавим эти сети в Compose:

```
services:
  location-service:
[...]
    networks:
      - location-network
      - redis-network
[...]
  redis:
    image: redis
    networks:
      - redis-network
networks:
  location-network:
  redis-network:
```

Redis не экспортирует никаких локальных портов; служба геолокации может обращаться к базе данных только потому, что `redis-network` указана в ее

разделе networks. Имя redis-network уже знакомо нам, мы использовали его в главе 3 «Основы сетей и томов». Закончив настройку сетей для нашего микросервиса, можно приступать к его интеграции в приложение диспетчера задач.

Выполнение запросов к микросервису местоположения

Выше мы благополучно запустили вновь созданный микросервис определения местоположения под управлением Compose. Однако напомню, что эта служба создавалась для работы в составе приложения диспетчера задач. Благодаря интеграции диспетчера задач со службой определения местоположения пользователь получит возможность указывать местоположение при создании задачи. Также, извлекая одну из существующих задач, пользователь сможет получить список местоположений рядом с координатами задачи.

Для этого диспетчер задач должен будет связаться со службой определения местоположения. Поэтому мы создадим службу внутри диспетчера задач, которая будет отправлять запросы микросервису местоположения. В этом модуле будут использоваться те же модели, что и в службе определения местоположения.

Исходный код модуля службы определения местоположения в приложении диспетчера задач можно найти на GitHub: https://github.com/PacktPublishing/A-Developer-s-Essential-Guide-to-Docker-Compose/blob/main/Chapter5/task-manager/location/location_service.go.

Поскольку в этой главе в приложение диспетчера задач будут добавлены новые функции, имеет смысл провести рефакторинг кода. Мы должны изменить существующие модели и включить модели местоположения, которые мы определили выше:

```
type Task struct {
    Id          string             `json:"id"`
    Name        string             `json:"name"`
    Description string             `json:"description"`
    Timestamp   int64              `json:"timestamp"`
    Location    *location.Location `json:"location"`
}
```

Кроме того, отделим методы сохранения данных от методов контроллера и переместим их в файл task_service.go (https://github.com/PacktPublishing/A-Developer-s-Essential-Guide-to-Docker-Compose/blob/main/Chapter5/task-manager/task/task_service.go).

Теперь поговорим немного о логике работы. Пользователь может добавить задачу без местоположения. Если местоположение указано и идентификатор уже существует, он будет передан службе определения местоположения без сохранения в базе данных. Если местоположение указано и прежде не

было сохранено, то мы можем задать все атрибуты местоположения и сохранить его. Указывая только идентификатор известного местоположения, нам не нужно указывать все его атрибуты. Разбив код приложения на модули, мы можем приступить к адаптации http-контроллеров (https://github.com/PacktPublishing/A-Developer-s-Essential-Guide-to-Docker-Compose/blob/main/Chapter5/task-manager/main.go).

После этого остается только обновить приложение Compose и добавить поддержку взаимодействий между диспетчером задач и службой определения местоположения.

Но для этого нужно удовлетворить некоторые требования:

- Redis и служба определения местоположения должны запускаться перед запуском приложения диспетчера задач;
- диспетчер задач должен иметь доступ к сетям предыдущих служб;
- диспетчер задач – это точка входа, поэтому ее порт нужно связать с портом хоста.

Образ Docker создается точно так же, как мы делали это в главе 2 «Запуск первого приложения с помощью Compose», но дополнительно мы должны добавить новый исходный код:

```
# syntax=docker/dockerfile:1
FROM golang:1.17-alpine
RUN apk add curl
WORKDIR /app
RUN mkdir location
RUN mkdir task
RUN mkdir stream
COPY go.mod ./
COPY go.sum ./
RUN go mod download
COPY *.go ./
COPY location/*.go ./location
COPY task/*.go ./task
COPY stream/*.go ./stream
RUN go build -o /task_manager
EXPOSE 8080
CMD [ "/task_manager" ]
```

Теперь можно перейти к файлу Compose, включающему диспетчер задач и службу определения местоположения:

```
// Chapter5/task-manager/docker-compose.yaml:19
[...]
  task-manager:
    build:
      context: ./task-manager/
    image: task-manager:0.1
    ports:
      - 8080:8080
    environment:
      - REDIS_HOST=redis:6379
```

```
      - LOCATION_HOST=http://location-service:8080
    depends_on:
      - redis
      - location-service
    networks:
      - location-network
      - redis-network
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8080/ping"]
      interval: 10s
      timeout: 5s
      retries: 5
      start_period: 5s
[...]
```

Теперь диспетчер задач создан и интегрирован со службой определения местоположения. Служба определения местоположения осталась внутренней службой, и ее не нужно экспортировать. Диспетчер задач, напротив, определяет конечную точку REST, доступную извне. В следующем разделе мы реализуем доступ к приложению с использованием взаимодействий на основе сообщений.

Потоковая передача событий задач

Выше мы благополучно запустили новый микросервис с помощью Compose. Однако было бы желательно знать, сколько раз посещалось каждое место или сколько задач было создано с течением времени.

Это задача, управляемая данными, для решения которой мы должны собирать и передавать информацию о приложении. С этой целью можно использовать потоки данных Redis. С их помощью наше приложение может передавать данные для обработки и анализа в другом приложении.

Для реализации поддержки потоков достаточно лишь немного изменить наш код. Для начала опубликуем сообщение в потоке Redis после добавления задачи.

С этой целью добавим в диспетчер задач службу, которая будет посылать события. Пока сообщения будут отправляться только при добавлении новой задачи.

Следующий код реализует службу `TaskStream`, отвечающую за отправку сообщений о создании задачи:

```
// Chapter5/task-manager/task/task-service.go:14
[...]
type TaskMessage struct {
    taskId      string
    location_id string
    timestamp   int64
}
```

```
func CreateTaskMessage(taskId string, location *location.Location,
        timestamp int64) TaskMessage {
    taskMessage := TaskMessage{
        taskId:    taskId,
        timestamp: timestamp,
    }

    if location != nil {
        taskMessage.location_id = location.Id
    }

    return taskMessage
}

func (ts *TaskMessage) toXValues() map[string]interface{} {
    return map[string]interface{}{"task_id": ts.taskId,
                                  "timestamp": ts.timestamp,
                                  "location_id": ts.location_id}
}

func (ts *TaskStream) Publish(c context.Context, message TaskMessage) error {

    cmd := ts.Client.XAdd(c, &redis.XAddArgs{
        Stream: "task-stream",
        ID:     "*",
        Values: message.toXValues(),
    })

    if _, err := cmd.Result(); err != nil {
        return err
    }

    return nil
}
```

Реализовав функцию отправки сообщений, изменим метод `PersistTask` и добавим в него отправку сообщения после создания задачи:

```
// Chapter5/task-manager/task/task-service.go:28
[...]
func (ts *TaskService) PersistTask(c context.Context, task Task) error {

    values := []interface{}{"Id", task.Id, "Name", task.Name, "Description",
                            task.Description, "Timestamp", task.Timestamp}

    if task.Location != nil {
        if err := ts.LocationService.AddLocation(task.Location); err != nil {
            return err
        }
        values = append(values, "location", task.Location.Id)
    }

    hmset := ts.Client.HSet(c, fmt.Sprintf("task:%s", task.Id), values)

    if hmset.Err() != nil {
```

```
        return hmset.Err()
    }

    z := redis.Z{Score: float64(task.Timestamp), Member: task.Id}
    zadd := ts.Client.ZAdd(c, "tasks", &z)

    if zadd.Err() != nil {
        return hmset.Err()
    }

    mes := stream.CreateTaskMessage(task.Id, task.Location, task.Timestamp)

    return ts.TaskStream.Publish(c, mes)
}
[...]
```

Теперь наше приложение будет отправлять события при добавлении новых задач. В следующем разделе мы приступим к использованию этих сообщений.

Добавление микросервиса обработки событий задач

В предыдущем разделе мы добавили в приложение диспетчера задач отправку событий. Теперь мы можем добавить приложение, которое будет управляться этими событиями. Роль такого приложения у нас будет играть служба `events`. Она будет получать данные из потока Redis и выводить их в консоль.

Объем кода будет небольшим, и нам потребуется только клиент Redis.

Давайте добавим код, который будет извлекать события из потока:

```
[...]
client. XGroupCreateMkStream (ctx, stream, consumerGroup, "0").Result()

for {
    entries, err := client.XReadGroup(ctx,
        &redis.XReadGroupArgs{
            Group:    consumerGroup,
            Consumer: consumer,
            Streams:  []string{stream, ">"},
            Count:    1,
            Block:    0,
            NoAck:    false,
        },
    ).Result()

    for i := 0; i < len(entries[0].Messages); i++ {
        messageID := entries[0].Messages[i].ID
        values := entries[0].Messages[i].Values

        taskId := values["task_id"]
```

```
        timestamp := values["timestamp"]
        locationId := values["location_id"]

        log.Printf("Received %v %v %v", taskId, timestamp, locationId)

        client.XAck(ctx, stream, consumerGroup, messageID)
    }
}
[...]
```

Создадим для службы файл Dockerfile:

```
# syntax=docker/dockerfile:1

FROM golang:1.17-alpine

RUN apk add curl

WORKDIR /app

COPY go.mod ./
COPY go.sum ./

RUN go mod download

COPY *.go ./

RUN go build -o /events_service

CMD [ "/events_service" ]
```

Затем добавим службу обработки событий в Compose:

```
services:
  location-service:
    [...]
  task-manager:
    [...]
  event-service:
    build:
      context: ./events-service/
    image: event-service:0.1
    environment:
      - REDIS_HOST=redis:6379
    depends_on:
      - redis
    networks:
      - redis-network
networks:
  location-network:
  redis-network:
```

Очевидно, что новой службе не требуется столько же настроек Compose, сколько необходимо REST-службе. Поскольку она получает данные исключительно из потока данных, ей требуется только подключение к потоку Redis.

Закончив настройку служб, можно попробовать запустить приложение и понаблюдать, как события добавления новых задач передаются в микросервис `events`:

```
$ docker compose up
...
chapter5-event-service-1     | 2022/05/08 09:03:38 Received
8b171ce0-6f7b-4c22-aa6f-8b110c19f83a 1645275972000 0c2e2081-
075d-443a-ac20-40bf3b320a6f
...
```

Нам удалось организовать получение событий, генерируемых при создании задач, и добавить соответствующий код в наше приложение Compose.

Итоги

В этой главе мы создали два микросервиса и интегрировали их в приложение диспетчера задач. Микросервисы имеют разную природу; один основан на взаимодействиях в стиле REST, а другой управляется сообщениями. Несмотря на различия, Compose позволяет управлять этими микросервисами и изолировать их.

Как нетрудно догадаться, мониторинг играет решающую роль для обеспечения бесперебойной работы служб. Организовав мониторинг надлежащим образом, можно обеспечить высокую доступность служб для конечного пользователя.

Следующая глава будет посвящена мониторингу и его реализации с помощью Prometheus.

Глава 6

Службы мониторинга с Prometheus

В предыдущей главе мы создали несколько служб для приложения диспетчера задач и организовали управление ими с помощью Compose. Преобразовав монолитное приложение в приложение на основе микросервисов, мы внесли коррективы в код и настроили взаимодействие между несколькими микросервисами.

В настоящее время взаимодействия осуществляются посредством REST или обмена сообщениями. В процессе организации взаимодействий между микросервисами мы познакомились с особенностями поддержки сетей в Compose и разграничили общедоступные и приватные службы.

Эта глава будет посвящена мониторингу служб с использованием Compose. Когда дело доходит до микросервисов, желательно иметь возможность наблюдать за их работой, чтобы своевременно устранять любые возникшие проблемы. Мы уже видели некоторые команды мониторинга в главе 4 «Выполнение команд Docker Compose», однако хотелось бы получить больше информации. Наша цель – организовать мониторинг и настроить метрики в нашем приложении. Метрики должны сохраняться в базе данных и быть доступными для запросов. Учитывая все вышесказанное, на роль инструмента мониторинга мы выберем Prometheus.

В этой главе рассматриваются следующие темы:

- что такое Prometheus;
- добавление конечной точки для Prometheus;
- настройка Prometheus для получения метрик;
- добавление Prometheus в сеть Compose;
- первый запрос метрик.

Что такое Prometheus?

Prometheus – это популярное решение с открытым исходным кодом для мониторинга, обладающее широким спектром возможностей, включая мони-

торинг событий и рассылку уведомлений. Prometheus следует активной (pull) модели HTTP, когда приложения предоставляют доступ к своим метрикам через конечную точку HTTP, а Prometheus извлекает их через регулярные интервалы времени и, при необходимости, может анализировать их. Для служб, в которых нет возможности организовать конечную точку HTTP для чтения метрик, Prometheus предлагает **Pushgateway** – промежуточную службу, куда другие службы могут самостоятельно отправлять свои данные.

Извлеченные данные сохраняются в базе данных серий, которая является частью Prometheus. Это позволяет использовать гибкие запросы и рассылать уведомления в режиме реального времени.

Вот некоторые из возможностей Prometheus:

- модель данных для сохранения метрик в виде временных рядов;
- язык запросов для получения данных из временных рядов;
- хранение метрик на одном автономном сервере;
- активная (pull) модель извлечения данных через HTTP;
- передача метрик через промежуточный шлюз для приложений, не поддерживающих HTTP;
- обнаружение или настройка целей, предоставляющих метрики;
- поддержка информационных панелей (dashboard).

В этой главе мы адаптируем наше приложение для экспорта метрик в Prometheus. Подробно рассмотрим возможности Prometheus и создадим информационную панель, отражающую основные рабочие характеристики приложения.

Добавление конечной точки для Prometheus

В этом разделе мы посмотрим, как добавить новую конечную точку в Compose. В качестве основы мы будем использовать код, оставленный нами в конце главы 5 «Подключение микросервисов друг к другу». Мы должны будем добавить в этот код новые конечные точки, которые позволят Prometheus извлекать метрики из нашего приложения.

Добавление конечной точки метрик в диспетчер задач

Добавить поддержку Prometheus в приложение на Go просто. Как описывается в инструкциях на сайте https://prometheus.io/docs/guides/go-application, нужно выполнить команды `go get`, которые загрузят необходимые библиотеки:

```
$ go get github.com/prometheus/client_golang/prometheus
$ go get github.com/prometheus/client_golang/prometheus/promauto
$ go get github.com/prometheus/client_golang/prometheus/promhttp
```

По умолчанию конечная точка Prometheus добавляется с помощью http-сервера Go:

```
func main() {
        http.Handle("/metrics", promhttp.Handler())
        http.ListenAndServe(":2112", nil)
}
```

Приложение диспетчера задач основано на фреймворке `go gin`. Поэтому потребуется использовать небольшой обходной путь, потому что `gin` предлагает обертку для обработчиков HTTP:

```
import (
...
"github.com/prometheus/client_golang/prometheus/promhttp"
...
)
...
// Конечная точка для извлечения метрик
r.GET("/metrics", gin.WrapH(promhttp.Handler()))
...
```

Если теперь запустить приложение, мы сможем получить доступ к метрикам:

```
$ curl localhost:8080/metrics
# HELP go_gc_cycles_automatic_gc_cycles_total Count of
completed GC cycles generated by the Go runtime.
# TYPE go_gc_cycles_automatic_gc_cycles_total counter
go_gc_cycles_automatic_gc_cycles_total 0
# HELP go_gc_cycles_forced_gc_cycles_total Count of completed
GC cycles forced by the application.
# TYPE go_gc_cycles_forced_gc_cycles_total counter
go_gc_cycles_forced_gc_cycles_total 0
# HELP go_gc_cycles_total_gc_cycles_total Count of all
completed GC cycles.
# TYPE go_gc_cycles_total_gc_cycles_total counter
go_gc_cycles_total_gc_cycles_total 0
...
```

Мы получили метрики из нашего приложения, собранные с момента его запуска, в их числе: объем используемой памяти, количество циклов сборки мусора и т. д.

Добавление конечной точки метрик в службу определения местоположения

Как и диспетчер задач, служба определения местоположения тоже основана на `go gin`; следовательно, для получения метрик из нее необходимо выпол-

нить точно такие же шаги. И результаты должны быть такими же. Исходный код обновленной службы можно найти на GitHub (https://github.com/PacktPublishing/A-Developer-s-Essential-Guide-to-Docker-Compose/tree/main/Chapter6/location-service).

Экспорт метрик из службы событий

Служба событий – это микросервис, управляемый событиями. Он получает события из потока Redis, поэтому нет необходимости настраивать конечную точку HTTP. Чтобы включить мониторинг с помощью Prometheus, можно добавить HTTP-сервер в службу только для этой цели или задействовать службу Pushgateway, упомянутую ранее.

Для нас важно знать, насколько быстро обрабатываются полученные сообщения. Поэтому каждый раз, когда микросервис получает и обрабатывает очередное событие, он должен отправлять соответствующую информацию в Prometheus с помощью сервиса Pushgateway.

Для этого мы будем использовать датчик. Согласно документации, датчик – это метрика с единственным числовым значением, которое может произвольно увеличиваться и уменьшаться (https://prometheus.io/docs/concepts/metric_types/#gauge).

Пока будем считать, что служба Pushgateway уже присутствует и готова принимать представленные метрики.

Следующий метод будет отправлять метрику:

```
...
processingTime = prometheus.NewGauge(prometheus.GaugeOpts{
    Name: "task_event_process_duration",
    Help: "Time it took to complete a task",
})

processedCounter = prometheus.NewCounterVec(
    prometheus.CounterOpts{
        Name: "task_event_processing_total",
        Help: "How many tasks have been processed",
    },
    []string{"task"},
).WithLabelValues("task")
...
```

Он будет вызываться в основном методе обработки сообщений после их обработки:

```
...
start := time.Now()
log.Printf("Received %v %v %v", taskId, timestamp, locationId)

client.XAck(ctx, stream, consumerGroup, messageID)
elapsed := time.Since(start)
```

```
processedCounter.Add(1)

millis := float64(elapsed.Milliseconds())
processingTime.Set(millis)

pushProcessingDurationToPrometheus(processingTime)
pushProcessingCount(processedCounter)
...
```

И последнее, но не менее важное: добавим методы, пересылающие данные в шлюз:

```
...
func pushProcessingDurationToPrometheus(processingTime prometheus.Gauge) {
    if err := push.New(getStrEnv("PUSH_GATEWAY",
          "http://localhost:9091"), "task_event_process_duration").
              Collector(processingTime).
              Grouping("db", "event-service").
              Push(); err != nil
    {
        fmt.Println("Could not push completion time to Pushgateway:", err)
    }
}

func pushProcessingCount(processedCounter prometheus.Counter) {
    if err := push.New(getStrEnv("PUSH_GATEWAY",
          "http://localhost:9091"), "task_event_processing_total").
              Collector(processedCounter).
              Grouping("db", "event-service").
              Push(); err != nil
    {
        fmt.Println("Could not push tasks processed to Pushgateway:", err)
    }
}
...
```

Итак, мы изменили код и добавили в наши службы поддержку получения метрик. Следующий наш шаг – настройка экземпляра Prometheus для анализа полученных данных. Поэтому теперь посмотрим, как организовать взаимодействие экземпляра Prometheus с одной из наших служб.

Настройка анализа метрик в Prometheus

Как упоминалось выше, работа Prometheus основана на модели активного извлечения (pull) метрик. Это означает, что службы предоставляют доступ к данным через конечную точку HTTP, а Prometheus должен их извлечь и проанализировать.

Чтобы Prometheus мог выполнять анализ данных, получаемых из конечной точки, необходимо определить конфигурацию задания. Конфигурации заданий для Prometheus описываются в формате YAML.

Будем считать, что нам нужно получать и анализировать метрики с интервалом в одну минуту.

Соответствующая конфигурация должна выглядеть так:

```
scrape_configs:
  - job_name: 'task-manager'
    scrape_interval: 1m
    metrics_path: '/metrics'
    static_configs:
      - targets: ['host.docker.internal:8080']
```

Если предположить, что приложение диспетчера задач запущено и файл YAML готов, мы можем создать YAML-файл Compose:

```
services:
  prometheus:
    image: prom/prometheus
    ports:
      - 9090:9090
    volumes:
      - ./prometheus.yaml:/etc/prometheus/prometheus.yml
```

Следующая команда запустит Prometheus с указанной конфигурацией:

```
$ docker compose -f ./prometheusold.yaml up
```

Подождав несколько минут, можно будет открыть в браузере URL Prometheus и увидеть результаты на экране (рис. 6.1).

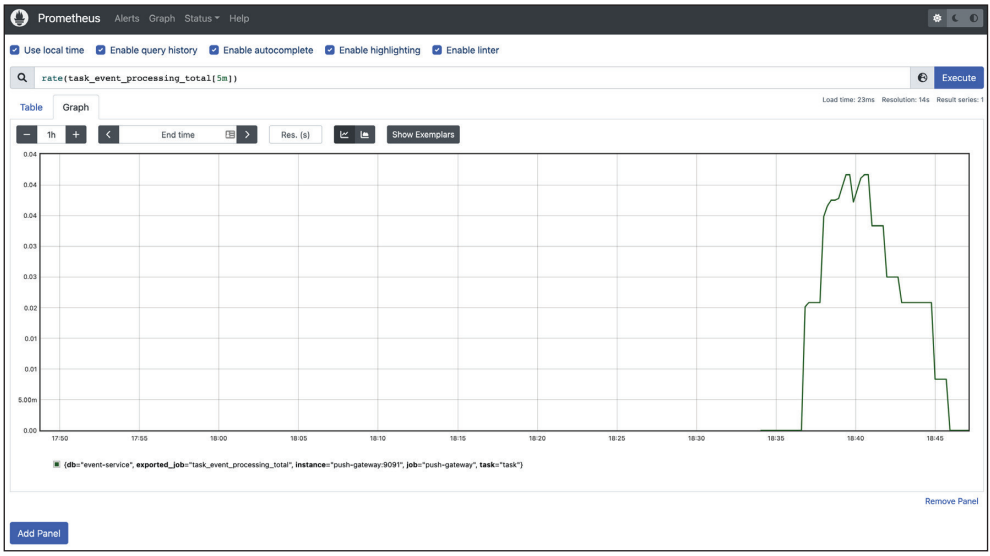

Рис. 6.1 ❖ Метрики в информационной панели Prometheus

На вкладке **Configuration** (Конфигурация) можно увидеть настроенную нами конфигурацию (рис. 6.2).

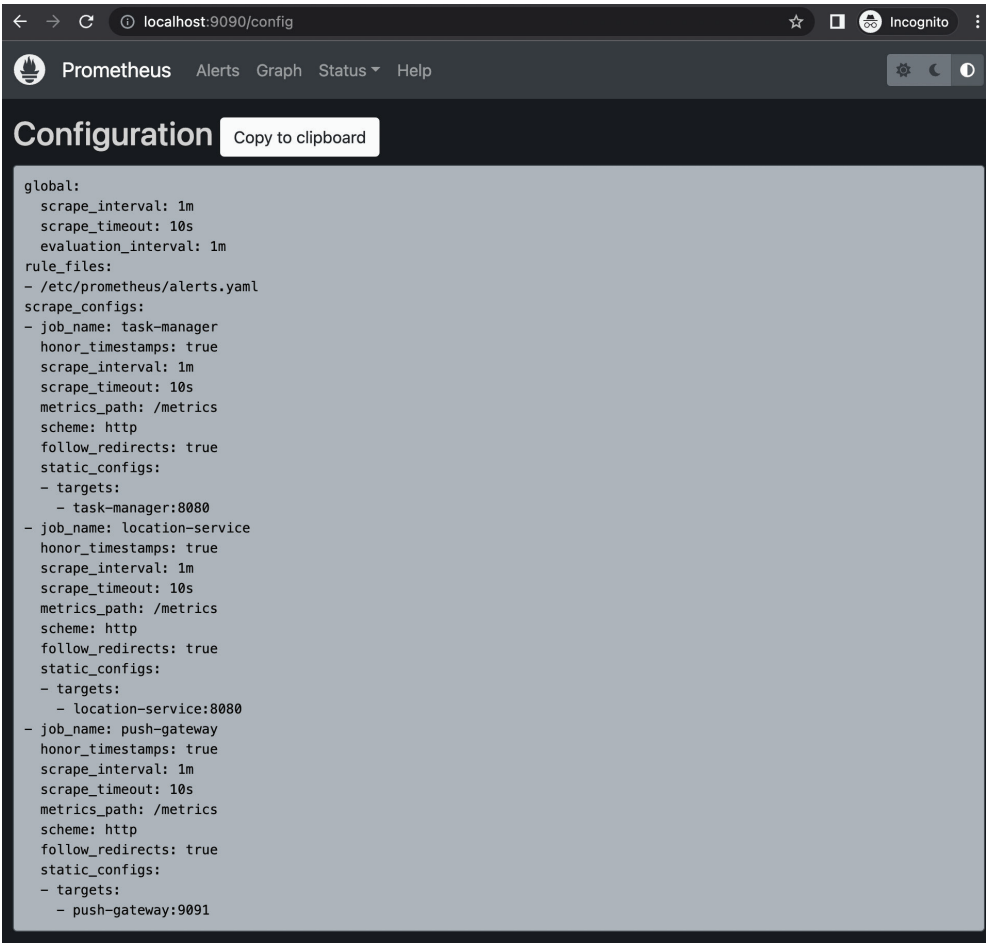

Рис. 6.2 ❖ Конфигурация, полученная из Prometheus

Мы настроили мониторинг диспетчера задач через Prometheus, позволив Prometheus собирать некоторые метрики. В следующем разделе мы проделаем то же самое со всеми нашими службами.

Добавление Prometheus в сеть Compose

Мы настроили Prometheus, и теперь он успешно собирает метрики, характеризующие работу приложения диспетчера задач. Поскольку мы решили добавить поддержку мониторинга во все наши службы, мы должны добавить настройки Prometheus в файл Compose для сбора и анализа метрик из всех наших служб.

Так как Prometheus взаимодействует со всеми существующими службами, эти взаимодействия не должны выполняться из внешней сети. Мы должны

добавить сеть, с которой Prometheus сможет работать. Назовем ее `monitoring-network`:

```
networks:
  location-network:
  redis-network:
  monitoring-network:
```

Затем создадим конфигурационный файл с настройками Prometheus. На данный момент у нас имеются три службы:

- диспетчер задач `task-manager`;
- служба определения местоположения `location-service`;
- служба событий `events-service`.

Службы `task-manager` и `location-service` прослушивают порт `8080`, соответственно, конфигурация должна выглядеть так:

```
scrape_configs:
  - job_name: 'task-manager'
    scrape_interval: 1m
    metrics_path: '/metrics'
    static_configs:
      - targets: ['task-manager:8080']
  - job_name: 'location-service'
    scrape_interval: 1m
    metrics_path: '/metrics'
    static_configs:
      - targets: ['location-service:8080']
```

Как видите, здесь используется внутренний DNS, предоставляемый Compose. Включим настройки Prometheus и добавим в эту же сеть другие службы:

```
  prometheus:
    image: prom/prometheus
    ports:
      - 9090:9090
    volumes:
      - ./prometheus.yaml:/etc/prometheus/prometheus.yml
    networks:
      - monitoring-network
```

Теперь нужно подключить к сети все остальные службы.

Обратите внимание, что `task-manager` и `location-service` остаются подключенными также к своим собственным сетям:

```
...
    networks:
      - location-network
      - redis-network
      - monitoring-network
...
```

После запуска приложения Compose с обновленными настройками Prometheus сможет получать метрики и анализировать их.

Результаты можно наблюдать, открыв в браузере страницу Prometheus по адресу `localhost:9090`.

Отправка метрик в Prometheus

После запуска приложения Compose Prometheus сможет получать метрики и анализировать их. Однако у нас есть служба, для которой предпочтительнее вариант с явной передачей метрик в Prometeus, чтобы исключить возможность непреднамеренного доступа к ним. Наши аналитические службы только выиграют от этого. Для этого нужно включить шлюз Prometheus.

Добавим конфигурацию шлюза в Compose:

```
// Chapter6/docker-compose.yaml:68
  push-gateway:
    image: prom/pushgateway
    networks:
      - monitoring-network
```

Теперь, когда мы настроили службу Pushgateway, служба событий сможет посылать свои метрики в Prometheus, передавая их в Pushgateway, а шлюз позволит Prometheus извлекать их через свою конечную точку. Соответственно, нужно настроить Prometheus для извлечения сообщений из шлюза:

```
// Chapter6/prometheus.yaml
  - job_name: 'push-gateway'
    scrape_interval: 1m
    metrics_path: '/metrics'
    static_configs:
      - targets: ['push-gateway:9091']
```

Основываясь на предыдущих шагах, также нужно применить измененный код службы событий. Теперь все наши службы охвачены мониторингом и передают данные в Kubernetes. Как и ожидалось, мы можем запустить приложение целиком с помощью команды `compose up`:

```
$ docker compose build
$ docker compose up -d
```

Теперь перейдем к следующему разделу и используем то, что было создано.

Первый запрос метрик

Мы благополучно отправляем наши метрики в Prometheus, так что теперь можем выполнять запросы и анализировать их. Имея данные, мы можем создавать информационные панели в Prometheus.

Предположим, что у нас есть много задач, созданных с течением времени, и нам нужно контролировать их. Для имитации создания задач воспользуемся следующим сценарием:

```
while true; do
curl --location --request POST 'localhost:8080/task/' \
--header 'Content-Type: application/json' \
--data-raw '{
    "id": "'$(date +%s%N)'",
    "name": "A task",
    "description": "A task that need to be executed at the timestamp specified",
    "timestamp": 1645275972000,
    "location": {
        "id": "1c2e2081-075d-443a-ac20-40bf3b320a6f",
        "name": "Liverpoll Street Station",
        "description": "Station for Tube and National Rail",
        "longitude": -0.081966,
        "latitude": 51.517336
    }
}'
sleep 1
done
```

После запуска предыдущий сценарий каждую секунду создает новую задачу с именем, сгенерированным с помощью bash-команды `date`.

Теперь сделаем еще шаг и создадим запрос PromQL, проверяющий частоту создания задач.

Вот как выглядит этот запрос:

```
rate(task_event_processing_total[5m])
```

На информационной панели отобразится результат (рис. 6.3).

Теперь можно остановить сценарий нажатием комбинации **Ctrl**+**C**.

Имея метрики, можно конструировать запросы и отыскивать инциденты. Такие запросы могут помочь нам контролировать работу наших приложений. Однако подобный подход требует постоянного внимания. Было бы удобнее иметь механизм, который уведомлял бы нас об определенных событиях. И такой механизм есть в Prometheus.

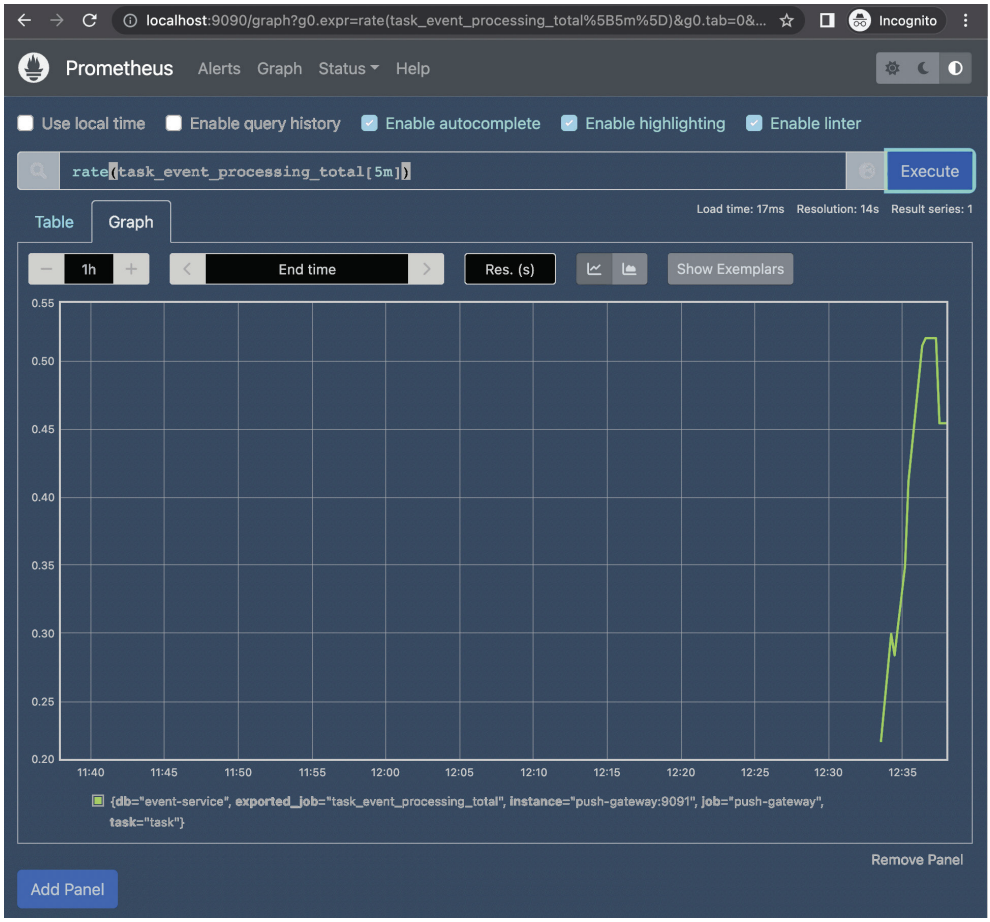

Рис. 6.3 ❖ Запрос, оценивающий частоту создания задач по метрикам, получаемым в Prometheus

Добавление уведомления

Еще одна возможность, поддерживаемая в Prometheus, – возможность рассылки уведомлений. Допустим, нам нужно, чтобы служба мониторинга присылала уведомление, если в течение последних 5 минут частота добавления новых задач превысила 0,2 задачи в минуту:

```
// Chapter6/alerts.yaml
groups:
- name: task-manager
  rules:
  - alert: too-many-tasks
```

```
    expr: rate(task_event_processing_total[5m]) > 0.2
    for: 1m
    annotations:
      summary: Too many tasks
```

Мы должны смонтировать этот файл в Prometheus; поэтому добавим его в конфигурационный файл `prometheus.yaml`:

```
// Chapter6/prometheus.yaml:17
...
rule_files:
  - '/etc/prometheus/alerts.yaml'
```

После непрерывного создания большого количества задач мы получаем ожидаемый результат.

Изначально никаких предупреждений не поступает:

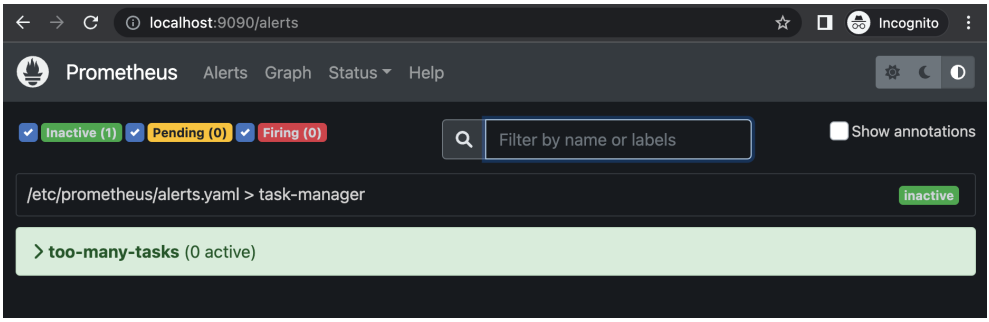

Рис. 6.4 ❖ Экран уведомлений Prometheus

По мере добавления все большего количества новых задач появляется уведомление в состоянии ожидания:

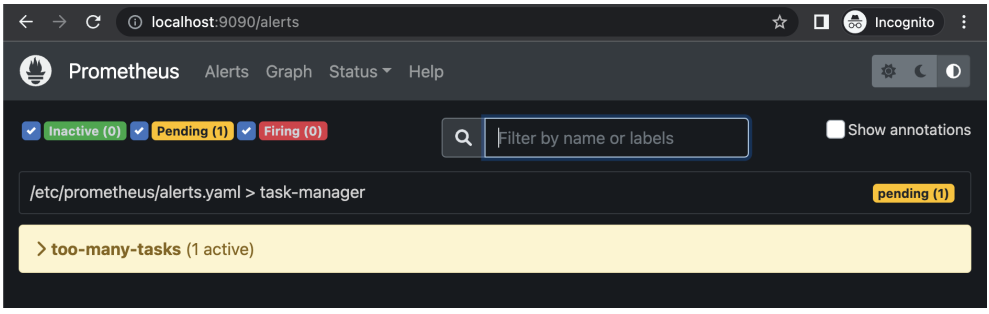

Рис. 6.5 ❖ Уведомление Prometheus в состоянии ожидания

В конце концов, уведомление переходит в активное состояние (рис. 6.6).

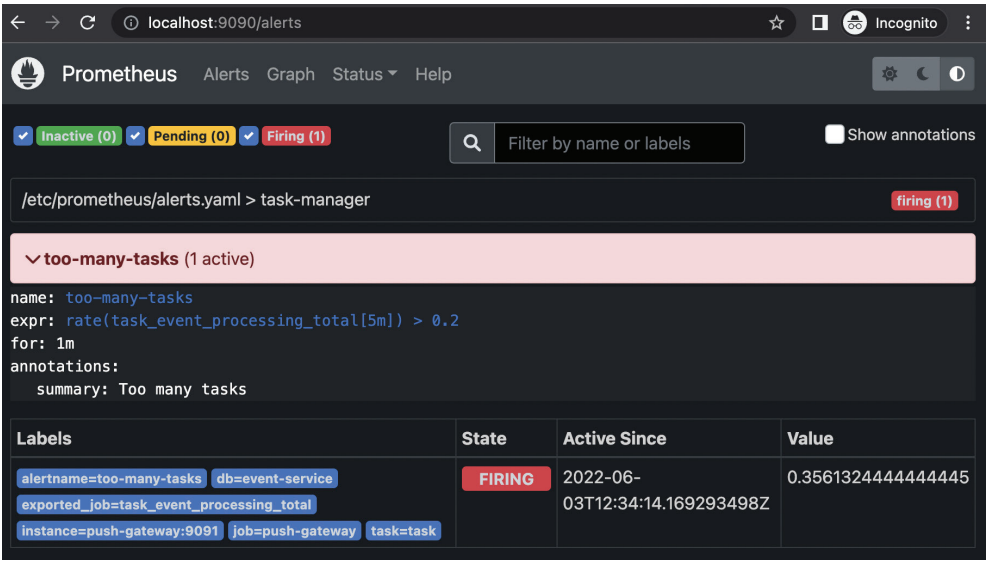

Рис. 6.6 ❖ Сработавшее уведомление Prometheus

У нас получилось создать уведомление на основе запроса, построенного выше. Мы добавили этот запрос в конфигурационный файл и смонтировали в контейнер Prometheus с помощью Compose.

Итоги

В этой главе мы настроили Prometheus для работы в Compose и организовали сбор метрик и отправку уведомлений. Таким образом, мы получили мультиконтейнерное приложение, выполняющееся под управлением Compose и отправляющее метрики. Это позволило нам настроить мониторинг приложения и отправку уведомлений в экстраординарных случаях. Все это стало возможным благодаря Compose и простоте добавления дополнительных конфигурационных файлов с использованием томов. Одна из проблем предыдущего подхода – отсутствие гибкости. Нам нужно, чтобы все службы, которые сейчас настраиваются в одном файле, могли запускаться независимо друг от друга и мы могли бы запускать только те компоненты, которые нам необходимы.

В следующей главе мы посмотрим, как можно разделить приложение Compose на модули и запускать его, используя настройки в нескольких файлах.

Глава 7

Комбинирование файлов Compose

До сих пор мы запускали наше мультиконтейнерное приложение монолитным способом, используя единый файл Compose, содержащий описания контейнеров прикладных служб, базы данных Redis и службы мониторинга Prometheus. Это сослужило нам хорошую службу на начальном этапе. Однако запуск приложения со всеми доступными зависимостями может вызвать проблемы. Полноценное приложение может потреблять много ресурсов и быть сложнее в отладке, и это может помешать сосредоточиться на определенном компоненте, требующем внимания. Иногда может потребоваться использовать только один компонент и исключить взаимодействия с другими компонентами или их запуск. Кроме того, иногда может быть нежелательно включать мониторинг или любой другой стек вспомогательных технологий, но не имеющий прямого отношения к прикладной области.

Compose дает возможность разбить приложение на несколько файлов и запускать приложение целиком, объединяя их. Такой подход позволяет запускать приложение более модульным способом. Мы должны иметь возможность запускать отдельные части приложения и полностью игнорировать весь стек.

В этой главе мы разделим наше приложение на несколько файлов и будем запускать их по модульному принципу.

Здесь рассматриваются следующие темы:

- разделение файлов Compose;
- объединение файлов Compose;
- выбор файлов Compose для запуска;
- создание различных окружений;
- объединение нескольких файлов Compose в один.

Технические требования

Примеры кода для этой книги можно найти в репозитории GitHub по адресу https://github.com/PacktPublishing/A-Developer-s-Essential-Guide-to-Docker-Compose. Если мы будем обновлять код, то все обновления будут доступны в этом репозитории.

Разделение файлов Compose

При разработке приложения диспетчера задач мы начали с одного простого приложения на Go, поддерживаемого базой данных Redis. Затем мы расширили возможности основного приложения, добавив два дополнительных микросервиса. Получив полнофункциональное приложение на основе микросервисов, мы посчитали, что необходимо организовать мониторинг, поэтому добавили Prometheus и Pushgateway, чтобы обеспечить возможность наблюдения за работой наших служб. На каждом шаге мы включали описания служб в файл Docker Compose.

Если детально изучить каждый шаг, можно определить компоненты, являющиеся общими для всего приложения, а также компоненты, которые должны запускаться и быть доступными независимо. К базовым компонентам, используемым всеми службами и необходимым для запуска приложения, относятся сети и база данных, и их можно сгруппировать в файл Compose.

В нашем приложении также есть определенные службы, описание которых можно поместить в отдельные файлы Compose. Например, служба определения местоположения может работать автономно, если она будет иметь свою базу данных. То же самое относится и к службе событий.

Prometheus тоже может работать отдельно, потому что не имеет прямого отношения к цели нашего приложения. Однако он необходим для оценки работоспособности приложения.

Итак, давайте приступим к созданию основы нашего приложения.

Основа для диспетчера задач

Основа для `task-manager` будет включать описание сетей и базы данных Redis. Описания сетей Docker имеют большое значение, потому что образуют основу для взаимодействий внутри приложения. База данных, в которой хранятся данные, тоже важна, потому что все наши службы так или иначе используют ее.

Поэтому в корне нашего проекта создадим файл `base-compose.yaml`:

```
// Chapter7/base-compose.yaml
services:
  redis:
    image: redis
    networks:
```

```
      - redis-network
networks:
  location-network:
  redis-network:
  monitoring-network:
```

Если мы запустим этот базовый файл, то сможем увидеть, что база данных Redis доступна:

```
// Chapter7/base-compose.yaml
$ docker compose -f base-compose.yaml up -d
$ docker compose -f base-compose.yaml ps
NAME                COMMAND                  SERVICE
STATUS              PORTS
chapter7-redis-1    "docker-entrypoint.s…"   redis
running             6379/tcp
$ docker compose -f base-compose.yaml down
```

Итак, основа готова. Она предоставляет базу данных Redis и основные сети. Далее приступим к службе определения местоположения.

Служба определения местоположения

Служба определения местоположения – первая, для которой мы создадим отдельный файл Compose, предназначенный только для запуска этой службы. Мы извлечем ее конфигурацию в отдельный файл Compose и будем использовать его из файла `base-compose.yaml`.

Вот как выглядит содержимое файла Compose для службы определения местоположения:

```
// Chapter7/location-service/docker-compose.yaml
services:
  location-service:
    build:
      context: location-service
    image: location-service:0.1
    environment:
      - REDIS_HOST=redis:6379
    depends_on:
      - redis
    networks:
      - location-network
      - redis-network
      - monitoring-network
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8080/ping"]
      interval: 10s
      timeout: 5s
      retries: 5
      start_period: 5s
```

Как видите, для создания отдельной конфигурации Compose службы определения местоположения достаточно скопировать эту конфигурацию и поместить ее в отдельный файл. И все же здесь можно видеть некоторые небольшие отличия. Параметр `context: .` в новом файле заменил параметр `context: location-service`. Причина в том, что `base-compose.yaml` и `location-service/docker-compose.yaml` находятся в разных файлах. Пути к файлам в Compose являются абсолютными и начинаются от местоположения первого указанного файла. По этой причине мы будем использовать контекстные пути в Compose. Следующая служба, конфигурацию которой мы отделим, – служба событий.

Служба событий

Конфигурация службы событий тоже останется без изменений и будет вынесена в отдельный файл:

```
// Chapter7/event-service/docker-compose.yaml
services:
  event-service:
    build:
      context: event-service
    image: event-service:0.1
    environment:
      - REDIS_HOST=redis:6379
      - PUSH_GATEWAY=push-gateway:9091
    depends_on:
      - redis
    networks:
      - redis-network
      - monitoring-network
```

После отделения первых двух служб можно переходить к основной службе, службе `task-manager`.

Диспетчер задач

Мы переходим к службе `task-manager` в последнюю очередь потому, что она взаимодействует с двумя другими службами.

Извлечем ее конфигурацию и поместим в отдельный файл Compose:

```
// Chapter7/task-manager/docker-compose.yaml
services:
  task-manager:
    build:
      context: task-manager
    image: task-manager:0.1
    ports:
```

```
      - 8080:8080
    environment:
      - REDIS_HOST=redis:6379
      - LOCATION_HOST=http://location-service:8080
    depends_on:
      - redis
      - location-service
    networks:
      - location-network
      - redis-network
      - monitoring-network
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8080/ping"]
      interval: 10s
      timeout: 5s
      retries: 5
      start_period: 5s
```

Мы отделили конфигурацию диспетчера задач от других служб и теперь можем приступить к отделению компонентов Prometheus.

Prometheus

Prometheus – это служба мониторинга, поэтому ее конфигурацию тоже поместим в отдельный файл и дадим пользователю возможность решать, использовать ее или нет. Мы не включили Prometheus в число служб, запускаемых вместе с диспетчером задач, поэтому сразу после запуска функции мониторинга не будут доступны. Однако основное приложение будет иметь меньше зависимостей и потребует меньше ресурсов.

Службе Prometheus необходимы сервер Prometheus и служба `push-gateway`. Оба этих компонента являются решениями для мониторинга, поэтому включение их в службу Prometheus выглядит логичным.

В следующем файле Compose показана конфигурация Prometheus и `push-gateway`:

```
// Chapter7/monitoring/docker-compose.yaml
services:
  prometheus:
    image: prom/prometheus
    ports:
      - 9090:9090
    volumes:
      - ./monitoring/prometheus.yaml:/etc/prometheus/
prometheus.yml
      - ./monitoring/alerts.yml:/etc/prometheus/alerts.yaml
    networks:
      - monitoring-network
    depends_on:
      - task-manager
```

```
  push-gateway:
    image: prom/pushgateway
    networks:
      - monitoring-network
```

Выделением компонентов Prometheus в отдельный файл Compose мы завершили разделение приложения `task-manager` и можем приступить к объединению ранее созданных файлов. Теперь мы должны настроить запуск нашего приложения, чтобы получить точно такое же поведение, как и прежде.

Объединение файлов Compose

Завершив разделение приложения `task-manager`, мы должны организовать запуск нашего приложения, чтобы получить точно такое же поведение, как в главе 6 «Службы мониторинга с Prometheus». Проще говоря, приложение должно давать возможность создавать и хранить задачи, выполнять запросы к диспетчеру задач и определять местоположения задач.

Compose поддерживает возможность объединения нескольких файлов.

Запустим приложение и все его службы вместе:

```
docker compose -f base-compose.yaml \
  -f monitoring/docker-compose.yaml \
  -f event-service/docker-compose.yaml \
  -f location-service/docker-compose.yaml \
  -f task-manager/docker-compose.yaml  \
  up

 Network chapter7_location-network     Created     0.0s
 Network chapter7_redis-network        Created     0.0s
 Network chapter7_monitoring-network   Created     0.0s
 Container chapter7-redis-1            Created     0.0s
 Container chapter7-push-gateway-1     Created     0.0s
 Container chapter7-location-service-1 Created     0.0s
 Container chapter7-event-service1     Created     0.0s
 Container chapter7-task-manager 1     Created     0.0s
 Container chapter7-prometheus-1       Created     0.0s
```

Обратите внимание на префикс `chapter7`. Он определяется базовым файлом Compose, стоящим первым в списке. Файл `base-compose.yaml` находится в каталоге `Chapter7`, поэтому относительный путь начинается с `Chapter7`.

К настоящему моменту у нас получилось разбить исходное приложение на части, а также запустить их все вместе. Функциональность приложения осталась прежней, а модульность упрощает его разработку. Однако нам не хватает гибкости: мы должны запускать приложение, указав все файлы, и мы не можем выбрать отдельную службу, на которой хотелось бы сосредоточиться. В следующем разделе мы посмотрим, как Compose может помочь сделать приложение еще более модульным.

Выбор файлов Compose для запуска

Одна из проблем, с которой мы столкнулись в предыдущем разделе, – необходимость запускать все файлы Compose, составляющие приложение. Однако мы достигли некоторого уровня модульности, разделив единый файл Compose на несколько частей. На следующем шаге мы реализуем возможность запуска различных модулей приложения по отдельности для отладки и тестирования.

Использование Hoverfly

Наши службы зависят друг от друга, поэтому единственный доступный нам вариант – запускать приложение целиком. Однако в процессе разработки и тестирования мы можем имитировать некоторые службы, создающие зависимости, и запускать приложение локально.

Существенную помощь в этом может оказать Hoverfly (https://hoverfly.io/). Hoverfly может перехватывать трафик и имитировать запросы и ответы.

Давайте запустим экземпляр Hoverfly в режиме захвата:

```
services:
  hoverfly:
    image: spectolabs/hoverfly
    ports:
      - :8888
    networks:
      - location-network
      - monitoring-network
    entrypoint: ["hoverfly","-capture","-listen-on-host","0.0.0.0"]
```

Добавив Hoverfly, мы будем использовать его для перехвата, а затем и для воспроизведения трафика при тестировании нашего приложения.

Наследование служб

Compose предоставляет возможность **наследования служб**, с помощью которой можно получить модифицированную версию существующей службы без дублирования ее конфигурации.

Наследование службы заключается в импортировании существующего файла Compose и внесении изменений в интересующие компоненты.

Рассмотрим следующий пример:

```
services:
  db:
    extends:
      file: databases.yml
      service: postgresql
```

```
    environment:
      - AUTOVACUUM=true
```

Это еще один файл Compose, наследующий определение службы `postgresql` из файла Compose `databases.yml` и добавляющий дополнительную переменную окружения.

Захват трафика с помощью Hoverfly

В нашем приложении есть две службы, создающие HTTP-трафик:

- `task-manager`, посылающая запросы службе `location-service`;
- `event-service`, посылающая запросы службе `push-gateway`.

В обоих случаях используется клиент `http` по умолчанию, что упрощает настройку Hoverfly в роли прокси через переменную `env`.

В данном случае мы унаследуем службы `task-manager` и `event-service` и включим http-прокси.

Вот как выглядит измененная конфигурация службы `event-service`:

```
services:
  event-service:
    extends:
      file: ./event-service/docker-compose.yaml
      service: event-service
    environment:
      - HTTP_PROXY=hoverfly:8500
    depends_on:
      - hoverfly
```

А так выглядит измененная конфигурация службы `task-manager`:

```
// Chapter7/task-manager/capture-traffic-docker-compose.yaml
services:
  task-manager:
    extends:
      file: ./task-manager/docker-compose.yaml
      service: task-manager
    environment:
      - HTTP_PROXY=hoverfly:8500
    depends_on:
      - hoverfly 0
```

Используя эту конфигурацию, мы сможем захватывать передаваемый трафик. Давайте запустим все вместе и посмотрим, захватил ли Hoverfly хоть что-нибудь:

```
docker compose -f base-compose.yaml \
  -f monitoring/docker-compose.yaml \
  -f event-service/capture-traffic-docker-compose.yaml \
```

```
-f location-service/docker-compose.yaml \
-f task-manager/capture-traffic-docker-compose.yaml \
-f hoverfly/docker-compose.yaml \
up
```

После создания нескольких задач зайдем в Hoverfly и проверим перехваченные запросы:

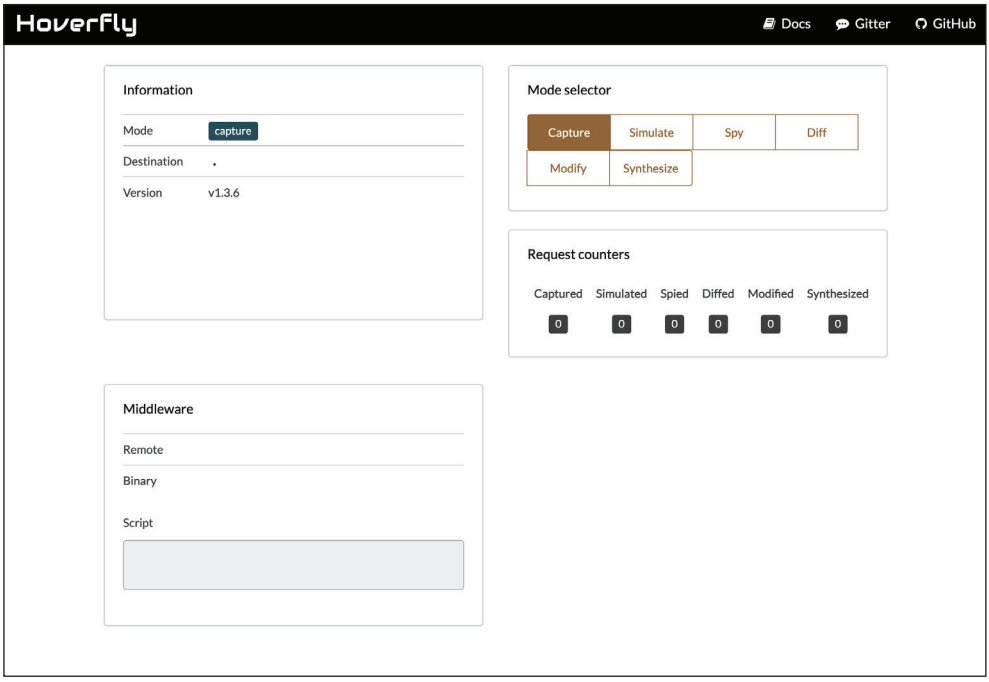

Рис. 7.1 ❖ Начальная страница Hoverfly

Действительно, запросы были перехвачены. Теперь экспортируем данные, захваченные Hoverfly.

Все захваченные данные можно экспортировать в файл JSON:

```
curl http://localhost:8888/api/v2/simulation
{"data":{"pairs":[{"request":{"path":[{"matcher":"exact",
"value":"/location/0c2e2081-075d-443a-ac20-
...
"schemaVersion":"v5.1","hoverflyVersion":"v1.3.6",
"timeExported":"2022-05-22T13:35:46Z"}}
```

Эта команда извлечет все данные из всех служб, которые были захвачены. Теперь сделаем еще один шаг и извлечем захваченный трафик для каждой службы.

Извлечение данных для имитации службы определения местоположения

Обратите внимание, что диспетчер задач `task-manager` использует службу определения местоположения `location-service`. Чтобы иметь возможность запускать диспетчер задач для тестирования и отладки, нужно сымитировать службу `location-service` с помощью Hoverfly.

Файл с данными для имитации уже находится в каталоге службы `location-service`. Чтобы извлечь эти данные из предыдущего запроса, можно выполнить следующий шаг:

```
cd location-service
curl --location --request GET \
  'http://localhost:8888/api/v2/simulation?urlPattern=location-service:8080' \
  > location-simulation.json
```

Данные для имитации будут помещены в файл `location-simulation.json`, который можно использовать, запустив Hoverfly в режиме моделирования.

Извлечение данных для имитации службы Pushgateway

Служба `event-service` посылает службе Pushgateway свои метрики для мониторинга. Этот трафик мы уже перехватили.

Требуемый файл с данными для имитации находится в каталоге `monitoring`. Поэтому следующий наш шаг – эти данные:

```
cd monitoring
curl --location --request GET \
  'http://localhost:8888/api/v2/simulation?urlPattern=push-gateway:9091' \
  >  push-gateway-simulation.json
```

Данные для имитации будут помещены в файл `push-gateway-simulation.json`, который можно использовать, запустив Hoverfly в режиме моделирования.

Адаптация моделирования

В режиме моделирования Hoverfly следует определенным правилам сопоставления компонентов HTTP-запроса. Например, чтобы сымитировать запрос динамической конечной точки с переменной `path`, Hoverfly должен быть настроен на ответ так, чтобы имя целевой конечной точки соответствовало регулярному выражению с именем конечной точки в модели Hoverfly.

В нашем случае тело вызовов REST будет динамическим. Поэтому адаптируем ранее извлеченные данные для имитации и получим полезную нагрузку из тела запроса POST, используя правило `"body":[{"matcher":"glob","value":"*"}]}`.

Создание фиктивных приложений с помощью Hoverfly

Теперь мы можем создавать фиктивные приложения, используя данные для имитации, экспортированные ранее.

Для начала сосредоточимся на развертывании `task-manager` с использованием фиктивной службы `location-service`, сымитированной с помощью Hoverfly.

Имитация службы определения местоположения

Теперь, имея данные для имитации, мы можем сымитировать службу `location-service` без ее запуска. Наш файл Compose, предназначенный для этой цели, будет развертывать только службу `task-manager`.

Вот как выглядит содержимое файла Compose, использующего Hoverfly для имитации:

```
services:
  location-service:
    image: spectolabs/hoverfly:v1.3.6
    ports:
      - 8888:8888
    networks:
      - location-network
      - redis-network
    volumes:
      - ./location-service/location-simulation.json:/etc/
hoverfly/location-simulation.json
    entrypoint: ["hoverfly","-webserver","-listen-on-
host","0.0.0.0", "-import", "/etc/hoverfly/location-simulation.
json", "-pp","8080"]
```

Запустим его и посмотрим на результаты:

```
docker compose -f base-compose.yaml \
  -f task-manager/docker-compose.yaml \
  -f location-service/mock-location-service.yaml \
  up
```

Мы можем взаимодействовать со службой `task-manager`, не запуская службу определения местоположения.

Имитация Pushgateway

Следующей службой, которую мы попробуем запустить в автономном режиме, будет служба событий. Она зависит от компонента `push-gateway`. У нас уже есть данные для имитации, полученные на предыдущем шаге, поэтому просто создадим файл Compose для запуска службы событий без этой зависимости:

```
services:
  push-gateway:
    image: spectolabs/hoverfly:v1.3.6
    ports:
      - 8888:8888
    networks:
      - monitoring-network
      - redis-network
    volumes:
      - ./monitoring/push-gateway-simulation.json:/etc/
hoverfly/push-gateway-simulation.json
    entrypoint: ["hoverfly","-webserver","-listen-on-
host","0.0.0.0", "-import", "/etc/hoverfly/push-gateway-
simulation.json", "-pp","8080"]
```

Теперь запустим службу событий автономно:

```
docker compose -f base-compose.yaml \
  -f event-service/docker-compose.yaml \
  -f monitoring/mock-push-gateway.yaml  \
  up
```

Мы можем взаимодействовать со службой `task-manager` без запуска службы определения местоположения. Кроме того, мы можем запустить службу событий без запуска компонента `push-gateway`. Мы запускаем службы, используя только необходимые компоненты и никакие другие службы. Благодаря этому у нас появляется необходимая гибкость в разработке.

Создание различных окружений

Нам удалось избавиться от зависимостей в наших службах и получить возможность запускать только то, что нужно для тестирования и отладки.

Исследовав команды, которые мы запускали, можно заметить, что в каждом случае использовались разные файлы.

Compose дает нам возможность комбинировать разные файлы и собирать разные окружения.

Запуск с включенным захватом

Как мы увидели ранее, у нас есть возможность организовать захват трафика, которым обмениваются приложения:

```
docker compose -f base-compose.yaml \
  -f monitoring/docker-compose.yaml \
  -f event-service/capture-traffic-docker-compose.yaml \
  -f location-service/docker-compose.yaml \
  -f task-manager/capture-traffic-docker-compose.yaml \
```

```
  -f hoverfly/proxy.yaml \
  up
```

Это окружение можно использовать, если возникнет необходимость создать новые имитации для тестирования.

Запуск с отключенным мониторингом

Кроме того, мы можем создать окружение без мониторинга:

```
docker compose -f base-compose.yaml \
  -f monitoring/mock-push-gateway.yaml \
  -f event-service/docker-compose.yaml \
  -f location-service/docker-compose.yaml \
  -f task-manager/docker-compose.yaml \
  up
```

Это окружение может помочь запустить приложение Compose с меньшим количеством ресурсов.

Запуск приложений по отдельности

Во время разработки бывает крайне важно сосредоточиться на каком-то одном компоненте. Теперь мы можем сделать это, запуская службы изолированно и используя имитации везде, где это необходимо:

- `task-manager`:

```
docker compose -f base-compose.yaml \
  -f location-service/mock-location-service.yaml \
  -f task-manager/docker-compose.yaml \
  up
```

- `location-service`:

```
docker compose -f base-compose.yaml \
  -f location-service/docker-compose.yaml \
  up
```

- `event-service`:

```
docker compose -f base-compose.yaml \
  -f monitoring/mock-push-gateway.yaml \
  -f event-service/docker-compose.yaml \
  up
```

Разделив исходное приложение на несколько файлов Compose, мы смогли попробовать разные их комбинации и получить на выходе разные приложения. Объединив различные файлы Compose, мы получили разные окружения, служащие определенным целям. Например, мы можем создать окружение

без мониторинга, окружение для захвата трафика или окружение для тестирования. Теперь мы знаем все интересующие нас комбинации и вместо объединения файлов Compose вручную в командной строке можем создать унифицированные конфигурации для каждого случая.

Объединение нескольких файлов Compose в один

Мы создали различные окружения, комбинируя файлы Compose. Это способствует процессу разработки, но делает его более сложным. Compose дает возможность объединить различные файлы в разных ситуациях.

Использование команды config

Обратите внимание, что `config` – это команда Docker Compose, объединяющая указанные файлы.

Например, у нас может появиться желание запустить службу определения местоположения в автономном режиме:

```
docker compose -f base-compose.yaml \
  -f location-service/docker-compose.yaml \
  config
```

В результате получится унифицированный файл JSON:

```
name: chapter7
services:
  location-service:
    build:
      context: /path/to/git/A-Developer-s-Essential-Guide-to-
Docker-Compose/Chapter7/location-service
      dockerfile: Dockerfile
    depends_on:
      redis:
        condition: service_started
    environment:
      REDIS_HOST: redis:6379
    healthcheck:
      test:
      - CMD
      - curl
      - -f
      - http://localhost:8080/ping
      timeout: 5s
      interval: 10s
      retries: 5
```

```
      start_period: 5s
    image: location-service:0.1
    networks:
      location-network: null
      monitoring-network: null
      redis-network: null
  redis:
    image: redis
    networks:
      redis-network: null
networks:
  location-network:
    name: chapter7_location-network
  monitoring-network:
    name: chapter7_monitoring-network
  redis-network:
    name: chapter7_redis-network
```

Здесь мы создали объединенную конфигурацию с помощью `config`. Как видите, команда `config` дает нам более управляемый способ объединения файлов Compose для использования в различных сценариях.

Итоги

У нас получилось превратить монолитное приложение в гибкое модульное приложение с несколькими конфигурационными файлами Compose. Кроме того, основываясь на зависимостях между файлами Compose, мы создали фиктивные компоненты, позволяющие в процессе разработки запускать службы по отдельности. Затем мы объединили различные файлы Compose и создали разные окружения для нашего приложения. После этого мы рассмотрели способ объединения различных файлов Compose в один для создания необходимого окружения.

В следующей главе мы увидим, как с помощью Compose можно моделировать промышленные окружения.

Глава 8

Локальное моделирование промышленного окружения

В предыдущей главе у нас получилось разбить конфигурацию приложения на несколько файлов Compose. Ближе к концу главы мы создали несколько разных окружений для нужд тестирования и отладки. В частности, мы создали окружения с фиктивными службами, окружение для перехвата трафика между службами и окружение с включенным мониторингом.

Возможность использовать фиктивные службы, создавать различные окружения и осуществлять мониторинг приложения способствует более высокой продуктивности и эффективности повседневной разработки. В этой главе мы сосредоточимся на локальном моделировании промышленного окружения с помощью Compose.

Продуктивность команды разработчиков можно увеличить с самого начала, если уменьшить число зависимостей и создать окружение разработки, готовое к тестированию.

На роль целевого сценария мы выберем окружение AWS и попробуем смоделировать службы AWS локально, а также создадим представление окружения AWS Lambda через приложение Docker Compose.

Целевым окружением будет служить простое приложение, получающее данные в формате JSON. Приложение должно хранить информацию в DynamoDB и посылать обновления в службу **Simple Queue Service (SQS)**. Другое приложение Lambda будет читать сообщения из SQS и сохранять их в службе **Simple Storage Service (S3)** для архивирования.

В реальном окружении AWS все задействованные компоненты, включая SQS, **Simple Notification Service (SNS)**, S3 и DynamoDB, хорошо интегрированы, что упрощает работу приложения. Однако, чтобы это окружение было

доступно для локального тестирования, потребуется применить некоторые обходные решения для создания интегрированного окружения AWS. Наше приложение будет включать следующие компоненты: приложение Lambda на основе REST, сохраняющее запрос в DynamoDB, приложение, имитирующее публикацию сообщений SQS в функции Lambda, и приложение Lambda на основе SQS, сохраняющее события SQS в S3.

В этой главе рассматриваются следующие темы:

- разделение приватных и общедоступных рабочих нагрузок;
- настройка локальной службы DynamoDB;
- настройка локальной службы SQS;
- настройка локальной службы S3;
- настройка функции Lambda на основе REST;
- настройка функции Lambda на основе SQS;
- подключение функций Lambda.

Технические требования

Примеры кода для этой книги можно найти в репозитории GitHub по адресу https://github.com/PacktPublishing/A-Developer-s-Essential-Guide-to-Docker-Compose. Если мы будем обновлять код, то все обновления будут доступны в этом репозитории.

Разделение приватных и общедоступных рабочих нагрузок

Поскольку действия, происходящие в AWS, являются внутренними, мы должны разделить рабочие нагрузки на приватные и общедоступные.

Приложение Lambda на основе REST, получающее данные в формате JSON, должно находиться в общедоступной сети, потому что оно будет взаимодействовать с конечным пользователем. Приложение Lambda на основе SQS, читающее события SQS и сохраняющее их в S3, должно быть приватным. Приложение, имитирующее события SQS для приложения Lambda на основе SQS, тоже будет приватным.

Имитируемые компоненты AWS, такие как DynamoDB, SQS и S3, должны использовать приватную сеть.

Мы определим сети, как показано в следующей конфигурации Compose:

```
networks:
  aws-internal:
  aws-public:
```

Теперь, когда сети определены, можно приступать к добавлению фиктивных компонентов AWS в приложение Compose.

Настройка локальной службы DynamoDB

База данных DynamoDB часто используется в AWS. DynamoDB – это бессерверная база данных **NoSQL** типа «ключ-значение». Для тестирования в локальных окружениях AWS предоставляет локальную версию DynamoDB.

Используем образы Docker, предоставленные AWS, и добавим их в конфигурацию Compose. Для удобства экспортируем порт локально.

Как упоминалось выше, служба DynamoDB будет использовать приватную сеть:

```
services:
  dynamodb:
    image: amazon/dynamodb-local
    ports:
     - 8000:8000
    networks:
     - aws-internal
```

Поскольку база данных DynamoDB будет запускаться и работать локально, создадим в ней таблицу.

Создание таблиц DynamoDB

В отличие от Redis, в DynamoDB нужно заранее создать таблицу. Мы добавим в приложение Compose контейнер, который будет создавать нужную нам таблицу в DynamoDB.

Мы делали нечто подобное в главе 2 «Запуск первого приложения с помощью Compose». Контейнер будет использовать образ AWS CLI (https://hub.docker.com/r/amazon/aws-cli) и переопределять команду, использующую утилиту `DynamoDB` для создания таблицы.

Контейнер инициализации будет зависеть от службы DynamoDB, потому что база данных DynamoDB должна быть доступна всегда. Остальная часть приложения будет зависеть от службы инициализации, потому что таблица должна быть создана до первой попытки ее использования.

Следующий сценарий создаст нужную нам таблицу:

```
#!/bin/sh
aws dynamodb create-table \
    --table-name newsletter \
    --attribute-definitions \
        AttributeName=email,AttributeType=S \
    --key-schema \
        AttributeName=email,KeyType=HASH \
    --provisioned-throughput \
        ReadCapacityUnits=5,WriteCapacityUnits=5 \
    --table-class STANDARD --endpoint-url  http://dynamodb:8000
http://host.docker.internal:8000
```

Далее добавим контейнер инициализации в приложение Compose:

```
services:
  dynamodb-initializer:
    image: amazon/aws-cli
    env_file:
      - ./mock_crentials.env
    entrypoint: "/create_table.sh"
    depends_on:
      - dynamodb
    volumes:
      - ./create_table.sh:/create_table.sh
    networks:
     - aws-internal
```

Как видите, мы добавили несколько фиктивных учетных данных, чтобы использовать интерфейс командной строки AWS, а также переопределить конечную точку для DynamoDB. Теперь мы можем протестировать DynamoDB локально.

Взаимодействие с локальной DynamoDB

Мы можем протестировать локальную DynamoDB, настроенную выше, запустив небольшой фрагмент.

Сначала запустим DynamoDB:

```
$ docker compose -f base-compose.yaml up
```

Наш пример мы напишем на языке Go, поэтому можем использовать существующий проект `go` или создать новый проект с командами инициализации, использованными в предыдущих главах.

После этого нужно загрузить дополнительные зависимости (следующие команды должны выполняться в каталоге `dynamodb-snippet`):

```
$ go get github.com/aws/aws-sdk-go/aws
$ go get github.com/aws/aws-sdk-go-v2/service/dynamodb
```

Теперь можно использовать следующий небольшой фрагмент, помещающий запись в таблицу DynamoDB:

```
sess, _ := session.NewSession(&aws.Config{
    Region:      aws.String("us-west-2"),
    Credentials: credentials.NewStaticCredentials("fakeMyKeyId",
                                                  "fakeSecretAccessKey", ""),
})

svc := dynamodb.New(sess, aws.NewConfig().WithEndpoint(
    "http://localhost:8000").WithRegion("eu-west-2"))

item := Subscribe{
    Email: "john@doe.com",
```

```
    Topic: "what I subscribed",
}

av, _ := dynamodbattribute.MarshalMap(item)
input := &dynamodb.PutItemInput{
    Item:      av,
    TableName: aws.String("Newsletter"),
}
svc.PutItem(input)
```

Мы смоделировали локальную службу DynamoDB и создали таблицу с помощью контейнера. Мы также запустили пример кода, который будет сохранять элементы в созданной нами таблице DynamoDB. В нашем приложении Compose имеется база данных DynamoDB, которую смогут использовать наши службы. Следующий шаг – добавление в приложение Compose фиктивного компонента SQS.

Настройка локальной службы SQS

Служба SQS будет использоваться для уведомления о создании записи в DynamoDB. Сообщения в SQS будет отправлять приложение Lambda на основе REST.

`elasticmq` – очень популярный инструмент эмулятора SQS (https://github.com/softwaremill/elasticmq), который имитирует большинство функций, предоставляемых SQS.

Для отправки данных в SQS необходимо создать очередь. `elasticmq` дает возможность создать очередь при инициализации.

Вот как выглядит необходимая для этого конфигурация:

```
//sqs.conf
include classpath("application.conf")

queues {
  subscription-event{}
}
```

Теперь добавим конфигурацию `elasticmq` в наш файл Compose:

```
services:
  sqs:
    image: softwaremill/elasticmq
    ports:
     - 9324:9324
     - 9325:9325
    networks:
     - aws-internal
    volumes:
      - ./sqs.conf:/opt/elasticmq.conf
...
```

Как и в случае с DynamoDB, мы экспортируем порт локально. Кроме того, `elasticmq` предоставляет интерфейс администратора, доступный через порт `9325` (`http://localhost:9325/`).

Давайте реализуем взаимодействия с локальным брокером SQS на языке Go.

Для начала нужно подключить следующий модуль (следующие команды должны выполняться в каталоге `sqs-snippet`):

```
$ go get github.com/aws/aws-sdk-go/aws
$ go get github.com/aws/aws-sdk-go/service/sqs
```

Следующий фрагмент кода выведет очереди, доступные в службе:

```
session, _ := session.NewSession(&aws.Config{
  Region:      aws.String("us-west-2"),
  Credentials: credentials.NewStaticCredentials("fakeMyKeyId",
                                                "fakeSecretAccessKey", ""),
})

svc := sqs.New(session, aws.NewConfig().WithEndpoint(
    "http://localhost:9324").WithRegion(os.Getenv(AWS_REGION_ENV)))

result, _ := svc.ListQueues(nil)

for i, url := range result.QueueUrls {
  fmt.Printf("%d: %s\n", i, *url)
}
```

Мы успешно запустили эмулятор локальной службы SQS, создали очередь SQS, используя встроенные функции эмулятора, а также реализовали пример, публикующий данные в очереди SQS с помощью конечной точки эмулятора. Службы, размещенные в Compose, должны иметь возможность взаимодействовать с SQS и публиковать сообщения. В следующем разделе мы настроим фиктивный сервер S3 в Compose, чтобы получить возможность сохранения больших двоичных объектов.

Настройка локальной службы S3

S3 – это высокодоступная служба хранения объектов, предоставляемая AWS. Как и большинство других служб AWS, S3 предоставляет REST API для взаимодействия, а также SDK.

Чтобы смоделировать S3 локально, используем `S3mock` (https://github.com/adobe/S3Mock), популярный проект, доступный на GitHub.

Для него имеется образ Docker, который дает возможность настройки для создания корзины (хранилища).

Добавим его в наш файл Compose и подключим к приватной сети:

```
services:
...
```

```
  s3:
    image: adobe/s3mock
    ports:
     - 9090:9090
    networks:
     - aws-internal
    environment:
      - initialBuckets=subscription-bucket
```

Напишем также фрагмент кода, для чего сначала добавим дополнительные пакеты (следующие команды должны выполняться в каталоге `s3-snippet`):

```
$ go get github.com/aws/aws-sdk-go/aws
$ go get github.com/aws/aws-sdk-go/service/s3
```

Наш фрагмент кода перечислит все доступные корзины:

```
sess := session.Must(session.NewSessionWithOptions(session.
Options{
  SharedConfigState: session.SharedConfigEnable,
}))

s3 := s3.New(sess, aws.NewConfig().WithEndpoint(
    "http://localhost:9090").WithRegion("us-west-2"))

buckets, _ := s3.ListBuckets(nil)

for i, bucket := range buckets.Buckets {
  fmt.Printf("%d: %s\n", i, *bucket.Name)
}
```

У нас получилось запустить эмулятор службы S3. Мы настроили этот эмулятор и инициализировали корзину в нем. Затем запустили пример кода, перечисляющий созданные корзины S3. В следующем разделе мы настроим функцию Lambda на основе REST.

Настройка функции Lambda на основе REST

AWS предоставляет поддержку бессерверных функций Lambda. AWS Lambda – это решение для бессерверных вычислений, которое можно интегрировать и использовать различными способами. Один из способов – использовать в качестве основы для реализации REST API.

Функция Lambda на основе REST, которую мы реализуем, будет получать данные в формате JSON и сохранять их в DynamoDB.

Это можно легко смоделировать локально, благодаря наличию `docker-lambda`.

С помощью `docker-lambda` можно создать образ контейнера, имитирующего функцию AWS Lambda. Для этой цели AWS предоставляет образы, включающие также клиентов, упрощающих взаимодействия между функцией на Go и Lambda (https://github.com/lambci/docker-lambda).

Кроме того, этот компонент позволяет локально имитировать вызовы функций Lambda.

Начнем с реализации функции.

Сначала сохраним запрос в DynamoDB:

```
type Subscribe struct {
  Email string `json:"email"`
  Topic string `json:"topic"`
}

func HandleRequest(ctx context.Context, subscribe Subscribe)
(string, error) {
  dynamoDb, _:= dynamoDBSession()
  marshalled, _ := dynamodbattribute.MarshalMap(subscribe)
  input := &dynamodb.PutItemInput{
    Item:      marshalled,
    TableName: aws.String(TableName),
  }

  dynamoDb.PutItem(input)
  sendToSQS(subscribe)

  return fmt.Sprintf("You have been subscribed to the %s newsletter",
      subscribe.Topic), nil
}
```

Затем отправим сообщение в SQS:

```
func sendToSQS(subscribe Subscribe) {
  if !isSimulated() {
    return
  }
  if session, err := sqsSession(); err == nil {
    if bytes, err := jsonutil.BuildJSON(subscribe); err == nil {
      smsInput := &sqs.SendMessageInput{
        MessageBody: aws.String(string(bytes)),
        QueueUrl:    aws.String(os.Getenv(SQS_TOPIC_ENV)),
      }
      if _, err := session.SendMessage(smsInput); err != nil {
        fmt.Println(err)
      }
[...]

func sqsSession() (*sqs.SQS, error) {
  session, _ := session.NewSession()
  return sqs.New(session, aws.NewConfig().WithEndpoint(
      os.Getenv(SQS_ENDPOINT_ENV)).
      WithRegion(os.Getenv(AWS_REGION_ENV))), nil
}
```

Полный исходный код можно найти на GitHub (https://github.com/PacktPublishing/A-Developer-s-Essential-Guide-to-Docker-Compose/blob/main/Chapter8/newsletter-lambda/newsletter.go).

Теперь создадим Dockerfile для приложения:

```
FROM amazon/aws-lambda-go:latest as build
RUN yum install -y golang
RUN go env -w GOPROXY=direct
COPY go.mod ./
COPY go.sum ./
RUN go mod download
COPY *.go ./
RUN go build -o /main
FROM amazon/aws-lambda-go:latest
COPY --from=build /main /var/task/main
CMD [ "main" ]
```

и добавим файл Compose для приложения:

```
services:
  newsletter-lambda:
    build:
      context: ./newsletter-lambda/
    image: newsletter_lambda
    ports:
     - 8080:8080
    environment:
     - SIMULATED=true
     - DYNAMODB_ENDPOINT=http://dynamodb:8000
     - SQS_ENDPOINT=http://sqs:9324
     - SQS_TOPIC=/000000000000/subscription-event
    depends_on:
      - dynamodb-initializer
      - sqs
    env_file:
      - ./mock_crentials.env
    networks:
      aws-internal:
      aws-public:
```

Как видите, здесь мы ссылаемся на фиктивные службы AWS, созданные ранее, и собираем образ Docker. Это общедоступная служба с точкой входа в наше приложение; поэтому экспортируем порт локально.

Запустим приложение с помощью Compose:

```
docker compose -f docker-compose.yaml \
  -f newsletter-lambda/docker-compose.yaml \
  build

docker compose -f docker-compose.yaml \
  -f newsletter-lambda/docker-compose.yaml \
  up
```

Мы объединили файлы Compose, как это делалось в главе 7 «Комбинирование файлов Compose».

Запустив службу, можно протестировать ее, отправив запрос с помощью `curl`:

```
curl -XPOST \
  "http://localhost:8080/2015-03-31/functions/function/invocations" \
  -d '{"email":"john@doe.com","topic":"Books"}' \
  "You have been subscribed to the Books newsletter"
```

Итак, мы создали функцию AWS Lambda в Compose и организовали для нее возможность взаимодействия с фиктивными службами DynamoDB и SQS. Затем с помощью Compose мы смоделировали бессерверное приложение AWS. В следующем разделе мы сделаем еще один шаг и реализуем функцию AWS Lambda на основе SQS.

Настройка функции Lambda на основе SQS

У нас получилось запустить локально функцию AWS Lambda на основе REST. Наш следующий компонент тоже будет бессерверной функцией Lambda, но основанной на сообщениях. В частности, он будет получать события SQS, которые мы сгенерировали ранее.

Получаемые события SQS приложение Lambda будет сохранять в S3. Это приложение будет применять те же компоненты, которые мы использовали ранее.

Ниже приводится реализация обработчика функции:

```
func HandleRequest(ctx context.Context, sqsEvent events.SQSEvent) error {
    session := s3Session()
    for _, message := range sqsEvent.Records {
        var subscribe Subscribe
        json.Unmarshal([]byte(message.Body), &subscribe)

        key := fmt.Sprintf("%s.%d", hash(subscribe.Email),
            time.Now().UnixNano()/int64(time.Millisecond))

        marshalled, _ := json.Marshal(subscribe)

        session.PutObject(&s3.PutObjectInput{
            Bucket: aws.String(os.Getenv(SUBSCRIPTION_BUCKET_ENV)),
            Key:    aws.String(key),
            Body:   bytes.NewReader(marshalled),
            })
        }
    return nil
}
```

Обработчик функции получает экземпляр `SQSEvent` с сообщением SQS. Сообщения анализируются и сохраняются в S3 с использованием ключа, сгенерированным на основе хеша и времени.

AWS упрощает обработку сообщений SQS. Если вызов функции завершился успехом, то сообщение должно быть удалено из SQS. В противном случае сообщение останется в очереди.

Для сборки образа необходим Dockerfile:

```
FROM amazon/aws-lambda-go:latest as build
RUN yum install -y golang
RUN go env -w GOPROXY=direct
COPY go.mod ./
COPY go.sum ./
RUN go mod download
COPY *.go ./
RUN go build -o /main
FROM amazon/aws-lambda-go:latest
COPY --from=build /main /var/task/main
CMD [ "main" ]
```

Поскольку это приложение основано на AWS Lambda, файл Dockerfile идентичен тому, который мы написали для приложения Lambda на основе REST.

Далее создадим файл Compose:

```
services:
  s3store-lambda:
    build:
      context: ./s3store-lambda/
    image: s3store-lambda
    environment:
     - SIMULATED=true
     - S3_ENDPOINT=http://s3:9090
     - SUBSCRIPTION_BUCKET=subscription-bucket
     - AWS_REGION=eu-west-2
    links:
    -  "s3:subscription-bucket.s3"
    depends_on:
      - s3
    env_file:
      - ./mock_crentials.env
    networks:
      aws-internal:
```

Приложение должно быть доступно только изнутри, поэтому оно подключено лишь к приватной сети. Также в конфигурации присутствует ссылка на фиктивную службу AWS, которая была определена выше; однако в разделе `links` есть одна важная деталь.

Ссылки Docker Compose

Из-за особенностей работы S3 вместо корня конечных точек S3 используются конечные точки, к которым добавляются имена корзин.

Если предположить, что корзина имеет имя `my-bucket`, то URL для взаимодействий с этой корзиной будет иметь вид `https://my-bucket.s3.your-region.amazonaws.com/`.

Этот URL не соответствует нашему окружению, потому что у нас есть конечная точка `s3` и наш код будет пытаться получить доступ к URL `subscription-bucket.s3`.

Чтобы решить данную проблему, используем возможности `links` в Compose.

Используя `links`, можно определить `subscribe-bucket.s3` как дополнительный псевдоним для службы `s3` и таким способом связаться с ним через нашу службу.

К данному моменту мы успешно создали функцию Lambda на основе SQS и запустили ее локально. Нам удалось задействовать S3, применив обходное решение с псевдонимом для конечной точки. В следующем разделе мы объединим два приложения с помощью промежуточного локального приложения, имитирующего окружение AWS для функций Lambda на основе SQS.

Соединение функций Lambda

На данный момент мы подготовили и настроили фиктивные компоненты AWS для S3 и SQS, создали две функции Lambda: одну на основе REST и одну на основе SQS. В окружении AWS обе функции легко интегрируются, потому что при публикации сообщения в SQS AWS автоматически отправляет это сообщение функции Lambda, которая должна его обработать.

Эта бесшовная интеграция – то, чего нам не хватает на данный момент. Чтобы обеспечить эту функциональность, создадим службу, извлекающую сообщения из SQS и отправляющую их в функцию на основе SQS.

Код этой службы выглядит просто:

```
session, _ := sqsSession()
queueUrl := aws.String(os.Getenv(SQS_TOPIC_ENV))
msgResult, _ := session.ReceiveMessage(&sqs.
ReceiveMessageInput{
    QueueUrl: queueUrl,
})
if msgResult != nil && len(msgResult.Messages) > 0 {
    sqsEvent := map[string][]*sqs.Message{
    "Records": msgResult.Messages,
}

marshalled, _ := json.Marshal(sqsEvent)
http.Post(os.Getenv(S3STORE_LAMBDA_ENDPOINT_ENV),
    "application/json", bytes.NewBuffer(marshalled))

     for i := 0; i < len(msgResult.Messages); i++ {
        session.DeleteMessage(&sqs.DeleteMessageInput{
            QueueUrl:      queueUrl,
```

```
            ReceiptHandle: msgResult.Messages[i].ReceiptHandle,
        })
    }
}
```

Сообщения будут извлекаться из службы SQS в формате, который функция Lambda ожидает получить. После отправки сообщений в функцию Lambda они должны быть удалены из очереди.

Эта служба работает только локально, поэтому процедура создания образа будет намного проще.

Создадим Dockerfile для сборки образа:

```
# syntax=docker/dockerfile:1
FROM golang:1.17-alpine
WORKDIR /app
COPY go.mod ./
COPY go.sum ./
RUN go mod download
COPY *.go ./
RUN go build -o /main
CMD [ "/main" ]
```

и определим конфигурацию Compose:

```
services:
  sqs-to-lambda:
    build:
      context: ./sqs-to-lambda/
    image: sqs-to-lambda
    environment:
     - SQS_ENDPOINT=http://sqs:9324
     - SQS_TOPIC=/000000000000/subscription-event
     - S3STORE_LAMBDA_ENDPOINT=http://s3store-
lambda:8080/2015-03-31/functions/function/invocations
    depends_on:
      - sqs
      - s3store-lambda
    env_file:
      - ./mock_crentials.env
    networks:
      aws-internal:
networks:
  aws-internal:
```

Служба является внутренней и будет использовать только SQS, а поскольку она обращается к `s3store-lambda`, то зависит от этого компонента.

Если у вас запущены какие-либо активные сеансы Compose, остановите их, прежде чем продолжить и выполнить следующие команды.

Давайте запустим все приложение и посмотрим, как службы взаимодействуют друг с другом:

```
docker compose -f docker-compose.yaml \
  -f newsletter-lambda/docker-compose.yaml \
  -f s3store-lambda/docker-compose.yaml \
  -f sqs-to-lambda/docker-compose.yaml
  build

docker compose -f docker-compose.yaml \
  -f newsletter-lambda/docker-compose.yaml \
  -f s3store-lambda/docker-compose.yaml \
  -f sqs-to-lambda/docker-compose.yaml
  up
```

Вызовем функцию Lambda на основе REST, как мы делали это ранее:

```
curl -XPOST \
  "http://localhost:8080/2015-03-31/functions/function/invocations" \
  -d '{"email":"john@doe.com","topic":"Books"}' \
  "You have been subscribed to the Books newsletter"
```

В результате в консоли должны появиться журнальные записи всех служб:

```
...
chapter8-newsletter-lambda-1     | START RequestId: f2dcc750-
35a1-40d8-9c54-f7c2edc3bcfe Version: $LATEST
chapter8-newsletter-lambda-1     | END RequestId: f2dcc750-
35a1-40d8-9c54-f7c2edc3bcfe
...
chapter8-sqs-to-lambda-1         | 2022/07/24 21:31:03
Dispatching 1 received messages
...
chapter8-s3store-lambda-1        | START RequestId: 7caff9ab-
ddb4-46c7-b75f-0f726eaf2ae8 Version: $LATEST
chapter8-s3store-lambda-1        | END RequestId: 7caff9ab-
ddb4-46c7-b75f-0f726eaf2ae8
```

С помощью этой дополнительной службы мы смоделировали функциональность, которую AWS предоставляет автоматически. Ограничения, имевшиеся изначально, мы устранили с помощью решения на основе Compose. Будучи точкой входа для нашего приложения, служба на основе REST сохраняет данные в DynamoDB и отправляет сообщения в SQS. Затем сообщения SQS передаются в функцию Lambda на основе SQS с использованием этой внутренней службы.

Итоги

В данной главе у нас получилось развернуть облачную инфраструктуру локально на рабочей станции. Мы настроили эквивалентные фиктивные компоненты, представляющие службы AWS: DynamoDB, SQS и S3. Используя конфигурационные файлы Compose, мы настроили их и устранили некото-

рые ограничения, возникающие при локальной разработке. Это дало нам возможность разрабатывать код служб без необходимости взаимодействия с реальным промышленным окружением.

Далее мы приступили к реализации бессерверных функций для окружения AWS Lambda. Мы успешно запустили эти функции с помощью Compose, подготовив их к развертыванию в облачном окружении. И последнее, но не менее важное: мы смоделировали некоторые функции, которые предоставляет AWS, реализовав локальное приватное приложение. На протяжении всей этой главы у нас ни разу не возникла потребность взаимодействовать с консолью AWS и реальным промышленным окружением, и мы смогли все свое внимание сконцентрировать на разработке кода.

В следующей главе мы воспользуемся проектом из этой главы и реализуем **конвейер непрерывной интеграции и непрерывного развертывания** (Continuous Integration and Continuous Deployment, CI/CD).

Глава 9

Создание расширенных заданий CI/CD

В предыдущей главе мы смоделировали промышленное окружение AWS с помощью приложения Compose. Для этого были созданы фиктивные службы AWS, такие как DynamoDB, S3 и SQS. Также были смоделированы вызовы функций Lambda через контейнеры Docker и реализовано обходное решение для имитации передачи трафика службам Lambda на основе SQS за счет добавления дополнительной службы в Compose.

Благодаря этому мы смогли сосредоточиться на разработке приложения и избавиться от необходимости использовать консоль AWS и вообще какую-либо облачную инфраструктуру AWS. С самого начала мы сосредоточились на локальной разработке приложения и моделировании необходимых компонентов.

До сих пор мы вполне продуктивно занимались разработкой приложения, поэтому следующим логическим шагом будет добавление некоторого процесса CI/CD. Было бы очень желательно, чтобы на протяжении всего жизненного цикла разработки наше приложение создавалось, тестировалось и развертывалось автоматически.

Наше приложение на основе Lambda – хороший пример, как можно извлечь выгоду из Compose и смоделировать сложное приложение, основанное на выбранном решении CI/CD. Для тестирования приложению Lambda потребуется несколько компонентов. Compose может помочь в их развертывании в выбранном нами решении CI/CD.

В этой главе основное внимание будет уделено включению Docker Compose в решение CI/CD. Существует множество решений CI/CD от разных производителей. Поэтому мы рассмотрим несколько решений.

В этой главе рассматриваются следующие темы:

- введение в CI/CD;
- использование действий GitHub Actions с Docker Compose;
- использование конвейеров Bitbucket с Docker Compose;
- использование Travis с Docker Compose.

Технические требования

Примеры кода для этой книги можно найти в репозитории GitHub по адресу https://github.com/PacktPublishing/A-Developer-s-Essential-Guide-to-Docker-Compose. Если мы будем обновлять код, то все обновления будут доступны в этом репозитории.

Введение в CI/CD

CI/CD расшифровывается как **«continuous integration and continuous delivery»** – непрерывная интеграция и непрерывная доставка и представляет сочетание методов, облегчающих непрерывную интеграцию, непрерывную доставку и непрерывное развертывание. Частью этого процесса является автоматизация создания, тестирования и развертывания приложений.

Например, возьмем наше приложение Lambda. Это сложное окружение, состоящее из двух приложений на основе Lambda и трех служб AWS.

В нашем случае предполагается, что у нас есть команда, следующая принципам магистральной разработки (trunk-based development), способствующим внедрению непрерывной интеграции. Наша команда каждый раз будет вносить небольшие изменения в основную ветку кода, возможно, используя короткоживущие ветви, существующие лишь в период реализации новых возможностей. Для слияния изменений из этих ветвей в основную ветвь будут создаваться запросы на включение изменений (pull request). Запрос на включение изменений должен быть подтвержден командой разработчиков, и параллельно должен выполняться автоматизированный процесс CI/CD, создающий и тестирующий вновь добавленный код, как часть проверки возможности слияния. Если проверка выполнится успешно, то ветвь можно считать готовой к слиянию, и после этого должно произойти развертывание компонента.

Независимо от изменяемого компонента, будь то функция Lambda на основе REST или функция Lambda на основе SQS, мы должны убедиться, что изменения в этой функции не нарушат работу кода приложения, взаимодействующего с этим компонентом.

После слияния измененный код должен быть собран повторно и отправлен в рабочее окружение. Развертывание кода в реальной среде может отличаться от развертывания в тестовом окружении, в зависимости от того, где развертываются рабочие нагрузки. Например, для развертывания функции Lambda требуется новый образ Docker и вызов AWS API, в котором указывается вновь созданный образ Docker. Также могут потребоваться дополнительные настройки окружения, в котором будет работать функция AWS Lambda. Если в будущем мы переключимся на Kubernetes, то для развертывания приложения можно будет использовать Helm Chart или решение GitOps, такое как Argo CD. Решение GitOps будет выявлять изменения, произошедшие

в главной ветви, выбирать последние артефакты сборки и развертывать их в рабочем окружении без всякого вмешательства пользователя.

Наша цель в этой главе – задействовать Compose в заданиях CI/CD. К двоичному файлу Go, реализующему функцию Lambda, или к образу Docker можно применить канареечное развертывание. Однако при смене окружения в будущем может потребоваться настроить другой вид развертывания. Поэтому мы не будем фокусироваться на каком-то конкретном развертывании и сосредоточимся на включении команд Compose в задания CI/CD.

В ответ на каждую фиксацию в основной ветви мы будем запускать приложение Compose в задании CI/CD. Наша цель – запустить приложение Compose в процессе сборки CI/CD и протестировать его, прежде чем переходить к развертыванию.

Итак, мы кратко познакомились с идеей CI/CD и определились с желаемыми целями в отношении нашего приложения на основе Lambda. Исходный код приложения хранится в GitHub, поэтому мы продолжим реализацию заданий CI/CD для нашего приложения с помощью GitHub Actions.

Использование действий GitHub Actions с Docker Compose

Если ваш код размещен на GitHub, то, скорее всего, вы знакомы с GitHub Actions. GitHub Actions – это платформа CI/CD, предоставляемая GitHub. Используя GitHub Actions, можно добавлять процессы, создающие и тестирующие наш код. Эти процессы можно адаптировать для каждой ветви и каждого запроса на включение или использовать для добавления своих действий по развертыванию кода через GitHub.

Создание первого действия GitHub Action

Чтобы добавить рабочий процесс GitHub, необходимо поместить файлы YAML вместе с инструкциями, определяющими рабочий процесс, в каталог `.github/workflows`. Можно создать несколько таких файлов, и все они будут выполняться GitHub независимо.

В данный момент мы сосредоточимся на выполнении нашего приложения в основной ветви.

Вот основа нашего рабочего процесса:

```
name: subscription-service

on:
  push:
    branches:
```

```
    - main
jobs:
  build:
    timeout-minutes: 10
    runs-on: ubuntu-latest
    steps:
    - name: Checkout
      uses: actions/checkout@v2
```

Имя рабочего процесса – `subscription-service`, и этот процесс будет выполняться после отправки изменений в основную ветвь.

Процесс будет выполняться в виртуальном окружении – последней версии Ubuntu, предоставляемой GitHub Actions. Преимущество этого окружения заключается в том, что в нем уже установлено программное обеспечение Compose.

Далее мы добавляем шаг проверки репозитория. Поскольку каждое задание будет выполняться в новом экземпляре виртуального окружения, мы должны позаботиться о зависимостях и артефактах, создаваемых на каждом этапе. Для этого случая в GitHub имеется механизм кеширования, ускоряющий загрузку зависимостей.

Кеширование собранных образов

Сборка образов может занять много времени, чего желательно избегать в CI/CD, потому что процессы должны выполняться быстро и максимально гладко.

Задания, выполняющиеся продолжительное время, могут негативно повлиять на процесс автоматизации:

- время ожидания выполнения задания может истечь, что сделает невозможным получение и использование его результатов;
- процесс разработки замедляется;
- процесс CI/CD становится болезненным для разработчиков.

По этой причине мы используем возможность кеширования, предоставляемую в GitHub Actions:

```
...
    - name: Cache Local Images
      id: local-images
      uses: actions/cache@v3
      with:
        path: /var/lib/docker/
        key: local-docker-directory
...
```

Здесь мы потребовали кешировать каталог `/var/lib/docker`, в котором хранятся образы. В результате все этапы, использующие этот каталог, будут кешировать сгенерированное содержимое, а значит, все последующие этапы смогут получить артефакты, загруженные на предыдущих шагах.

Создание образов приложений

Мы готовы добавить следующий шаг, который будет создавать образы приложений. Как было показано в главе 4 «Выполнение команд Docker Compose», образы приложений можно создавать с помощью Compose.

Вот наш следующий шаг:

```
...
    - name: Build Images
      working-directory: ./Chapter8
      run: |
        docker compose -f docker-compose.yaml -f newsletter-
lambda/docker-compose.yaml -f s3store-lambda/docker-compose.
yaml -f sqs-to-lambda/docker-compose.yaml build
...
```

Поскольку мы решили сосредоточиться на приложении, разработанном в главе 8 «Локальное моделирование промышленного окружения», мы должны сменить рабочий каталог. Для этого используем раздел `working-directory`, ссылающийся на каталог `Chapter8`.

В разделе `run` указывается команда сборки, которую мы использовали ранее. Это может быть любая команда bash, доступная в выбранном виртуальном окружении.

Результатом этого действия будет создание образов Docker приложения.

После создания образов можно переходить к тестированию работоспособности нашего приложения.

Тестирование приложения Compose

Пришло время добавить следующий шаг, осуществляющий тестирование. Для этого мы запустим приложение Compose, а затем выполним команду `curl`, как делали это ранее, и проверим результаты.

Вот этот шаг:

```
...
    - name: Test application
      working-directory: ./Chapter8
      run: |
        docker compose -f docker-compose.yaml -f newsletter-
lambda/docker-compose.yaml -f s3store-lambda/docker-compose.
yaml -f sqs-to-lambda/docker-compose.yaml up -d
        sleep 20
        curl -XPOST "http://localhost:8080/2015-03-31/
functions/function/invocations" -d '{"email":"john@doe.
com","topic":"Books"}'
        sleep 20
```

```
        docker compose logs --tail="all"
...
```

Здесь мы выбираем `Chapter8` в качестве рабочего каталога. Затем запускаем приложение Compose в режиме демона. Режим демона дает возможность продолжать использовать сеанс терминала во время работы приложения.

Приостановка работы на 20 секунд дает возможность убедиться, что все службы работают. Затем команда `curl` вызывает функцию Lambda, служащую точкой входа.

Так как действия выполняются асинхронно, нужно подождать несколько секунд.

Проверить успешность выполнения команды можно по записям в журнале с помощью команды Compose `logs` с параметром `-tail`.

Итак, у нас получилось запустить приложение Compose в конвейере. Кроме того, у нас получилось протестировать его. Этот успех делает наши усилия по автоматизации более эффективными, так как показывает, что мы можем использовать Compose для моделирования инфраструктуры, подобной промышленному окружению, взаимодействия с ней и применения автоматических проверок в процессе разработки. В дальнейшем мы реализуем все то же самое в виде конвейера Bitbucket.

Использование конвейеров Bitbucket с Docker Compose

Конвейеры Bitbucket – это решение CI/CD для репозиториев, размещенных в Bitbucket. Имея репозиторий в Bitbucket, вместо внешних решений для CI/CD можно использовать легкодоступные конвейеры Bitbucket. Как и в предыдущем случае с GitHub Actions, мы последуем тому же процессу.

Создание первого конвейера Bitbucket

Прежде чем включить поддержку конвейеров Bitbucket, необходимо создать файл `bitbucket-pipelines.yml` в корневом каталоге проекта. После этого можно включить конвейеры в своем репозитории через настройки, как показано на рис. 9.1.

После включения конвейеров Bitbucket будет обрабатывать инструкции, указанные в файле `bitbucket-pipelines.yml`.

Вот основа содержимого файла `bitbucket-pipelines.yml`:

```
image: atlassian/default-image:3
options:
  docker: true
definitions:
```

```
  caches:
    compose: ~/.docker/cli-plugins

pipelines:
  default:
    - step:
        name: "Install Compose"
        caches:
          - compose
        script:
          - mkdir -p ~/.docker/cli-plugins/
          - curl -SL https://github.com/docker/compose/
releases/download/v2.2.3/docker-compose-linux-x86_64 -o
~/.docker/cli-plugins/docker-compose
          - chmod +x ~/.docker/cli-plugins/docker-compose
          - docker compose version
```

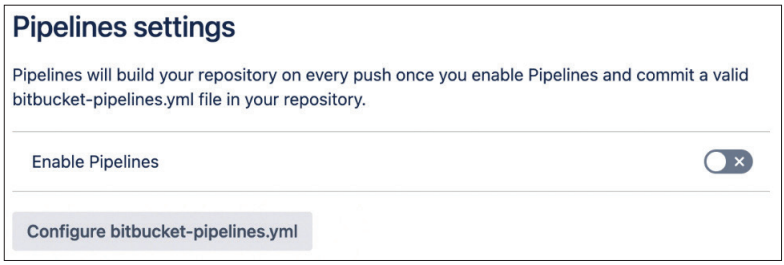

Рис. 9.1 ❖ Включение поддержки конвейеров Bitbucket

В разделе `image` указывается образ Docker, который должен использоваться при выполнении заданий CI/CD. Образ `atlassian/default-image:3` основан на дистрибутиве Linux Ubuntu 20.04 LTS. По умолчанию этот образ не поддерживает Compose, поэтому нам придется установить его. Для этого выполним тот же шаг, что и в главе 1 «Введение в Docker Compose».

Также можно заметить, что мы включили в конвейер службы Docker. Это необходимо для доступа к демону Docker и, соответственно, для выполнения команд Docker. Включив службы Docker с помощью этих параметров, сделали доступными возможности Docker во всех этапах конвейера.

Кеширование образов Compose и Docker

На первом шаге мы добавили поддержку Compose в образ Docker. Поскольку на каждом шаге создается новый контейнер Docker, нам нужно выполнять одни и те же команды на каждом шаге конвейера. Чтобы избежать этого, организуем кеширование каталога плагинов Docker. После выполнения следующего шага каталог и его содержимое появятся в кеше, что позволит использовать Compose.

Итак, в раздел `caches` мы добавили свой кеш для конвейеров Bitbucket, ссылающийся на каталог.

Обратите внимание, что это обходное решение для текущих конвейеров, но вообще существует более эффективный способ создания образа на основе `atlassian/default-image:3`, для которого будут выполнены инструкции по установке Compose.

Вместо создания собственных кешей можно использовать существующие реализации. В Bitbucket имеются различные предустановленные кеши, и один из них предназначен для Docker. Включив кеширование Docker на уровне этапа конвейера, мы можем быть уверены, что уже загруженные и созданные образы будут кешированы и готовы к использованию на следующих этапах.

Кеши для использования на каждом шаге указываются в разделе `caches`:

```
...
    - step:
        name: "Hello world"
        caches:
          - docker
        script:
          - docker run --rm hello-world
...
```

В этом случае будет загружен и кеширован образ для `hello-world`.

Включив кеширование для Compose и Docker и избавившись от необходимости извлекать зависимости, необходимые на каждом этапе, мы ускорили выполнение конвейера.

А теперь перейдем к следующему шагу – созданию образов.

Создание образов приложений

Мы получили возможность выполнять команды Compose в конвейере Bitbucket и теперь можем приступить к взаимодействию с нашим приложением Compose с помощью этих команд.

Добавим в конвейер шаг, создающий образы:

```
...
    - step:
        name: "Chapter 8 Build Images"
        caches:
          - docker
          - compose
        script:
```

```
          - cd Chapter8
          - docker compose -f docker-compose.yaml -f
newsletter-lambda/docker-compose.yaml -f s3store-lambda/docker-
compose.yaml -f sqs-to-lambda/docker-compose.yaml build
...
```

Он почти не отличается от предыдущего примера с GitHub Actions. Мы сменили каталог и запустили ту же команду сборки.

Как видите, у нас включены кеши `docker` и `compose`, поэтому нет необходимости снова устанавливать Compose. Кроме того, созданные нами образы будут доступны на следующем этапе конвейера. После создания образов можно переходить к организации тестирования нашего приложения.

Тестирование приложения Compose

Образы созданы, и теперь можно протестировать приложение Compose, как мы делали это ранее.

Вот как выглядит шаг тестирования приложения:

```
...
    - step:
        name: "Chatper 8 Test Application"
        caches:
          - docker
          - compose
        script:
          - cd Chapter8
          - docker compose -f docker-compose.yaml -f
newsletter-lambda/docker-compose.yaml -f s3store-lambda/docker-
compose.yaml -f sqs-to-lambda/docker-compose.yaml up -d
          - sleep 20
          - curl -XPOST "http://localhost:8080/2015-03-31/
functions/function/invocations" -d '{"email":"john@doe.
com","topic":"Books"}'
          - sleep 20
          - docker compose logs --tail="all"
...
```

Результат получится тем же, что и при использовании GitHub Actions. Настройка Docker Compose в конвейерах Bitbucket позволяет выполнять более сложные задания CI/CD, использующие Compose.

А теперь приступим к реализации той же логики с помощью еще одного популярного решения CI/CD – Travis CI.

Использование Travis с Docker Compose

Travis – это решение CI/CD на основе YAML, позволяющее хранить исходный код и очень хорошо интегрирующееся с GitHub. Поддержка CI/CD в Travis предоставляется бесплатно для проектов с открытым исходным кодом, поэтому этот инструмент пользуется большой популярностью среди таких проектов. Далее мы реализуем в Travis те же шаги, что и для предыдущих решений CI/CD.

Создание первого задания в Travis

Travis основан на YAML, как и предыдущие рассмотренные нами инструменты CI/CD. После включения интеграции Travis с проектом на GitHub в корневой каталог проекта необходимо поместить файл `.travis.yml` с инструкциями.

Вот основа содержимого в файле `.travis.yml`:

```
services:
  - docker

cache:
  directories:
    - $HOME/.docker/cli-plugins

jobs:
  include:
    - stage: "Install Compose"
      script:
      - mkdir -p /home/travis/.docker/cli-plugins/
      - curl -SL https://github.com/docker/compose/releases/
download/v2.2.3/docker-compose-linux-x86_64 -o ~/.docker/cli-
plugins/docker-compose
      - chmod +x ~/.docker/cli-plugins/docker-compose
      - docker compose version
```

Описание задания должно показаться вам знакомым. Как и в случае с конвейерами Bitbucket, здесь мы установили двоичный файл Compose для дистрибутива Linux. Поэтому кеширование этого шага так же важно, как и раньше.

Кеширование Compose

Travis поддерживает два варианта кеширования. Как можно видеть в примере выше, мы кешировали каталог, куда установлен Compose. Настроенный кеш будет доступен всем заданиям, включенным в конфигурацию.

В документации к Travis CI не рекомендует кешировать образы Docker, поэтому мы не будем делать этого. Однако если кеширование необходимо, то можно кешировать определенные образы, сохраняя и загружая их через кешированный каталог.

Создание образов приложений

Получив возможность выполнять команды Compose в Travis, можно приступать к созданию образов.

Вот задание, которое создаст образы:

```
...
    - stage: "Build Images"
      script:
      - cd Chapter8
      - docker compose -f docker-compose.yaml -f newsletter-
lambda/docker-compose.yaml -f s3store-lambda/docker-compose.
yaml -f sqs-to-lambda/docker-compose.yaml build
...
```

Это задание выполняется успешно, поэтому следующим нашим шагом будет организация тестирования приложения.

Тестирование приложения Compose

Этап тестирования в Travis не имеет существенных отличий от аналогичных этапов, реализованых выше, за исключением используемого синтаксиса.

Вот как выглядит содержимое раздела тестирования:

```
...
    - stage: "Test application"
      script:
        - cd Chapter8
        - docker compose -f docker-compose.yaml -f newsletter-
lambda/docker-compose.yaml -f s3store-lambda/docker-compose.
yaml -f sqs-to-lambda/docker-compose.yaml up -d
        - sleep 20
        - curl -XPOST "http://localhost:8080/2015-03-31/
functions/function/invocations" -d '{"email":"john@doe.
com","topic":"Books"}'
        - sleep 20
        - docker compose logs --tail="all"
...
```

Как и ожидалось, образы успешно созданы и протестированы.

Теперь вы знаете, как использовать Compose с разными инструментами CI/CD и организовать автоматизированный конвейер сборки и тестирования.

Итоги

Мы сделали это! Нам удалось запустить задания CI/CD, использующие функциональные возможности Compose, и теперь у нас есть возможность моделировать сложные промышленные окружения с помощью CI/CD, интегрировать свой код и выполнять тестирование сразу после слияния изменений.

В следующей главе мы посмотрим, как с помощью Compose реализовать развертывание приложения на удаленном хосте.

Часть III

РАЗВЕРТЫВАНИЕ С ПОМОЩЬЮ DOCKER COMPOSE

В этой части основное внимание будет уделено извлечению выгоды из применения Docker Compose для развертывания в промышленном окружении. Мы подготовим инфраструктуру общедоступных облаков, таких как AWS и Azure, и реализуем развертывание приложения в этой инфраструктуре с помощью Compose. Наконец, мы посмотрим, как перенести наши рабочие нагрузки Compose под управление механизма оркестрации Kubernetes.

Данная часть включает следующие главы:

- главу 10 «Развертывание Docker Compose на удаленных хостах»;
- главу 11 «Развертывание Docker Compose в AWS»;
- главу 12 «Развертывание Docker Compose в Azure»;
- главу 13 «Миграция на конфигурацию Kubernetes с помощью Compose».

Глава 10

Развертывание Docker Compose на удаленных хостах

В предыдущей главе мы с помощью Docker Compose создали задания CI/CD, а также различные окружения, которые можно использовать и в этом сценарии.

В данной главе мы сосредоточимся на развертывании приложений Docker на удаленном хосте. При разработке может быть множество различных причин, по которым нежелательно или невозможно развернуть приложение на другом хосте: приложение может быть ресурсоемким, вы решили поделиться кодом с коллегой, развертываемое приложение может использовать ресурсы в сети, к которым у вас нет доступа. Развертывание на удаленном хосте может стать решением этих проблем, поскольку позволяет развернуть приложение Docker на другой рабочей станции и сделать его доступным извне.

В этой главе рассматриваются следующие темы:

- удаленные хосты Docker;
- создание удаленного хоста Docker;
- контексты Docker;
- развертывание Compose на удаленных хостах;
- развертывание на удаленном хосте через IDE.

Технические требования

Примеры кода для этой книги можно найти в репозитории GitHub по адресу https://github.com/PacktPublishing/A-Developer-s-Essential-Guide-to-Docker-Compose. Если мы будем обновлять код, то все обновления будут доступны в этом репозитории.

Удаленные хосты Docker

Представьте, что у вас есть приложение, работающее на локальной машине, и вам нужно сделать его доступным другим, установив на машину Linux, находящуюся в облаке. Если это приложение основано на Docker Compose, его можно развернуть на виртуальной машине вручную с помощью команд оболочки. Однако существует более простой способ развернуть приложение на целевой виртуальной машине. Если на сервере установлен Docker, он может стать хостом Docker. Docker позволяет использовать возможности другого компьютера, если на нем установлен Docker и у вас настроен доступ к этому компьютеру.

Примером удаленного хоста могут служить установленные экземпляры Docker в Windows и macOS. Им обоим требуется виртуальная машина Linux для запуска Docker. Виртуальная машина Linux – это удаленный узел, с которым взаимодействует Docker CLI.

Теперь давайте посмотрим, как создать удаленный хост Docker.

Создание удаленного хоста Docker

Для создания хоста Docker нам понадобится машина с Linux. Это может быть запасной ноутбук или запасная виртуальная машина, на которой установлен дистрибутив Linux. Для подготовки такого хоста используются те же команды, которые мы применяли в главе 1 «Введение в Docker Compose». Поскольку у нас может не быть запасной рабочей станции с Linux, мы создадим хост Docker в AWS EC2.

Создание хоста Docker в AWS EC2

В этом разделе мы развернем виртуальную машину в AWS с использованием EC2. Она станет нашим удаленным хостом. Эти шаги применимы к любому доступному серверу на базе Linux, поэтому часть, касающуюся EC2, можно пропустить, если у вас есть доступная рабочая станция Linux.

Перейдя в раздел **IAM** в консоли AWS, получите ключ и секрет. Они необходимы, чтобы иметь возможность подготовить машину EC2 (рис. 10.1).

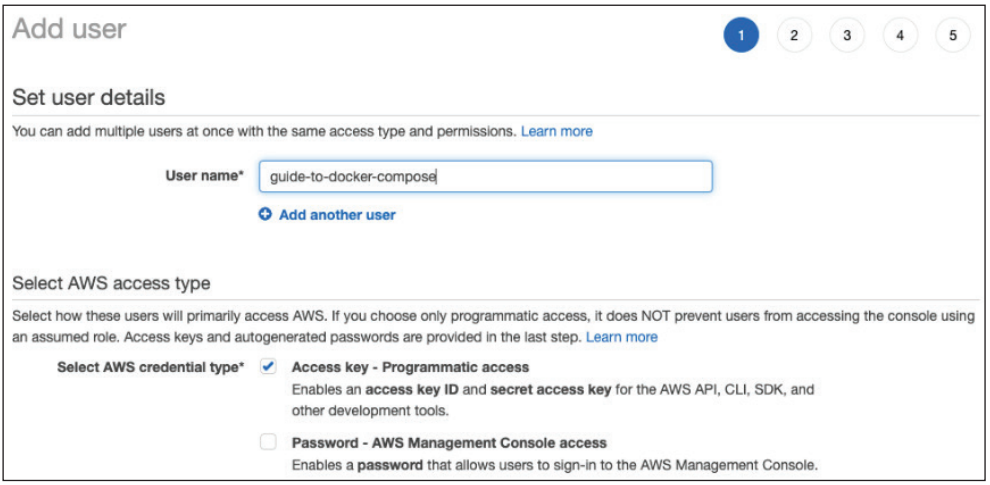

Рис. 10.1 ❖ Пользователь AWS

Получив учетные данные, можно перейти в раздел **VPC** и отыскать VPC по умолчанию для выбранного региона (рис. 10.2).

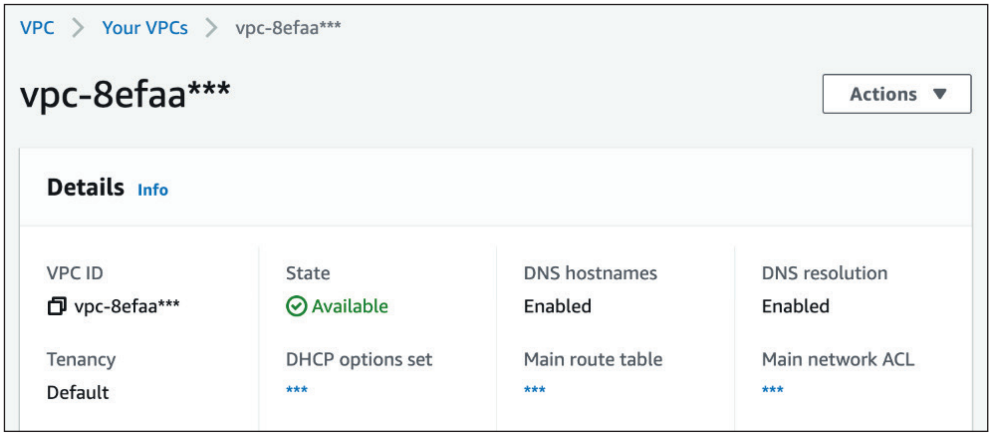

Рис. 10.2 ❖ Сеть VPC

Скопируйте идентификатор VPC ID, так как он понадобится нам позже.
Чтобы упростить подготовку машины EC2, используем Terraform.

Установка Terraform

Terraform – это современное решение **инфраструктура как код (Infrastructure as Code, IaC)**. Инфраструктуру и ресурсы в облаке можно определить с помощью декларативного языка конфигурации.

Для установки Terraform в вашей системе следуйте инструкциям в официальной документации (https://learn.hashicorp.com/tutorials/terraform/install-cli).

Как только двоичный файл Terraform появится в командной строке, мы можем проверить его версию:

```
$ terraform version
Terraform v1.2.3
on darwin_arm64
```

Terraform инициализирует вашу инфраструктуру и начнет отслеживать изменения в состоянии Terraform. Состоянием Terraform может быть локальный файл, наблюдение за файлом, размещенным в AWS S3, можно организовать с использованием эквивалентных решений поддержки больших двоичных объектов облачных провайдеров или настроить при наличии соответствующего плагина. Например, состояние можно сохранить в базе данных, такой как RavenDB, если, конечно, вы разработаете для нее плагин. Мы же будем использовать простой локальный файл.

При запуске Terraform подключится к выбранному облачному провайдеру и загрузит необходимые двоичные файлы. Например, если мы предоставим код для AWS, то Terraform загрузит плагины AWS.

Настройка машины EC2 с включенным SSH

Следующая наша цель – настроить экземпляр EC2, чтобы получить возможность входа в систему с помощью SSH. На этом экземпляре должны быть установлены Docker и Docker Compose.

Нам нужно, чтобы эта машина была доступна только с IP-адреса нашей рабочей станции. Соответственно, мы должны указать этот IP-адрес при настройке инфраструктуры. Кроме того, машина EC2 будет находиться в виртуальной приватной сети. В данном случае мы будем использовать VPC по умолчанию с идентификатором, который скопировали ранее.

Мы должны настроить переменные с IP-адресом и идентификатором VPC, чтобы потом использовать их при подготовке инфраструктуры:

```
variable "myvpc" {
}
variable "myip" {
}
```

Теперь сгенерируем ключи SSH, которые будут использоваться для доступа к машине EC2, и зарегистрируем закрытый ключ в агенте аутентификации OpenSSH:

```
// Chapter10/generate-key.sh
$ ssh-keygen -t rsa -b 2048 -f $(pwd)/ssh.key -N ""
$ ssh-add ssh.key
```

Этот шаг нужно выполнить перед инициализацией экземпляра EC2, чтобы подготовить машину EC2 с использованием уже имеющегося ключа. Кроме того, регистрация ключа в агенте аутентификации SSH упростит процесс подключения к серверу.

Следующий шаг – определение инфраструктуры. Для подключения к машине по SSH нужно определить группу безопасности, разрешающую доступ к экземпляру с нашей рабочей станции.

Следующее правило разрешает прохождение входящего трафика:

```
resource "aws_security_group" "remote_docker_host_security_group" {
  ...
  ingress {
    description      = "SSH from workstation"
    from_port        = 22
    to_port          = 22
    protocol         = "tcp"
    cidr_blocks      = ["${var.myip}/32"]
  }
  ...
}
```

Как видите, здесь используется созданная выше переменная `myip`.

Нам также нужно определить правило, пропускающее исходящий трафик. Собираясь разместить удаленный хост на этой машине, мы должны предусмотреть возможность взаимодействия с внешними реестрами Docker:

```
resource "aws_security_group" "remote_docker_host_security_group" {
  ...
  egress = [
    {
      cidr_blocks      = [ "0.0.0.0/0", ]
      description      = ""
      from_port        = 0
      ipv6_cidr_blocks = []
      prefix_list_ids  = []
      protocol         = "-1"
      security_groups  = []
      self             = false
      to_port          = 0
    }
  ]
  ...
}
```

Когда ключи будут сгенерированы, можно с помощью AWS загрузить открытый ключ как ресурс. Это может несколько замедлить процедуру начальной инициализации, зато позволит добавить ключ SSH на несколько компьютеров, которые могут действовать как хосты Docker.

Вот как выглядит определение ресурса ключа SSH:

```
resource "aws_key_pair" "docker_remote_host_key" {
  key_name   = "docker-remote-host-key"
  public_key = file("${path.module}/ssh.key.pub")
}
```

После создания машина EC2 будет использовать ранее созданный ключ.

Наконец, можно создать сам экземпляр EC2:

```
resource "aws_instance" "remote_docker_host" {
    ami = "ami-078a289ddf4b09ae0"
    instance_type = "t2.micro"

    key_name = aws_key_pair.docker_remote_host_key.key_name

    vpc_security_group_ids = [
        aws_security_group.remote_docker_host_security_group.id
    ]
}
```

Этот экземпляр EC2 будет создан с соответствующими настройками, и мы сможем получить к нему доступ извне.

Однако нам еще нужно установить Docker на работающую машину. Чтобы упростить эту задачу, можно воспользоваться механизмом автоматического запуска сценария `user_data` после инициализации машины EC2.

Используя этот механизм, можно настроить Docker Compose на машине EC2:

```
resource "aws_instance" "remote_docker_host" {
...
  user_data = <<-EOF
    #!/bin/bash
    yum install docker -y
    usermod -aG docker ec2-user
    systemctl start docker
    su ec2-user
    mkdir -p /home/ec2-user/.docker/cli-plugins
    curl -SL https://github.com/docker/compose/releases/
download/v2.2.3/docker-compose-linux-x86_64 -o /home/ec2-user/.
docker/cli-plugins/docker-compose
    chmod +x /home/ec2-user/.docker/cli-plugins/docker-compose
  EOF
...
}
```

Предыдущие команды должны показаться вам знакомыми; мы запускали их в главе 1 «Введение в Docker Compose». Мы использовали образ виртуальной машины на основе Red Hat, поэтому для установки необходимых пакетов здесь используется `yum`.

Поскольку нам предстоит подключаться к этой машине извне, давайте также выведем на экран ее IP-адрес:

```
output "instance_ip" {
  description = "Remote host ip"
  value       = aws_instance.remote_docker_host.public_ip
}
```

Теперь у нас есть все необходимое для подготовки инфраструктуры.

Для выполнения необходимых команд Terraform в AWS требуется передать учетные данные. Это можно сделать через переменные окружения `AWS_AC-`

CESS_KEY_ID и AWS_SECRET_ACCESS_KEY. Также требуется указать регион, в котором мы будем работать, для чего можно настроить переменную AWS_REGION. Эти переменные можно экспортировать или передать непосредственно в команду Terraform. Еще можно использовать файлы учетных данных и конфигурацию, созданную при настройке aws-cli.

Для начала инициализируем Terraform:

```
AWS_ACCESS_KEY_ID=key-id
AWS_SECRET_ACCESS_KEY=access-key AWS_REGION="eu-west-2"
terraform init
```

Эти команды сохранят наше состояние в файле.

Теперь выполним следующие команды:

```
AWS_ACCESS_KEY_ID=***
AWS_SECRET_ACCESS_KEY=***
AWS_REGION="eu-west-2"
terraform apply -var myip=51.241.***.182 -var myvpc=vpc-a8d1b***
```

Они подготовят инфраструктуру и выведут IP-адрес, необходимый для подключения по SSH:

```
instance_ip = "18.133.27.148"
```

Теперь можно подключиться к машине EC2 по SSH, используя ранее созданный ключ, и проверить, установлен ли Docker Compose:

```
$ ssh ec2-user@18.130.80.179
[ec2-user@ip-172-31-37-105 ~]$ docker compose version
Docker Compose version v2.2.3
```

Как видите, сценарий user_data выполнился успешно. Мы также смогли без проблем подключиться к экземпляру с помощью ssh благодаря настройке ключей.

Использование удаленного хоста Docker

Теперь, создав удаленный хост, посмотрим, как можно заставить его выполнять команды Docker.

Попробуем запустить Redis:

```
DOCKER_HOST="ssh://ec2-user@18.130.80.179" docker run -it --rm redis
```

Если сейчас войти в экземпляр EC2 и выполнить docker ps, то можно увидеть, что на этой машине запущена база данных Redis:

```
$ ssh ec2-user@18.130.80.179 docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED
STATUS         PORTS      NAMES
e44e3bd3a41d   redis     "docker-entrypoint.s…"   10 seconds
ago   Up 9 seconds   6379/tcp   nifty_aryabhata
```

После создания удаленного хоста Docker на нем можно создавать контейнеры Docker, используя локальную рабочую станцию. Возможность использовать несколько хостов расширяет наши горизонты. Однако управление удаленными хостами – не самая простая задача. Поэтому в следующем разделе мы посмотрим, как для этой цели применять контексты Docker.

Контексты Docker

Необходимость использовать хост в каждой команде, выполняемой нами, может оказаться утомительной и приводить к ошибкам. Например, развертывание может завершиться ошибкой из-за того, что мы не указали хост при запуске команды и выполнили другую команду на локальном хосте.

Устранить такие проблемы могут помочь контексты Docker.

Создавая контексты, можно переключать конфигурацию Docker между несколькими контекстами и выбирать правильный контекст для каждого случая.

Итак, давайте создадим контекст для нашего хоста EC2:

```
$ docker context create ec2-remote \
  --docker host=ssh://ec2-user@18.130.80.179
```

Мы создали новый контекст, но все еще находимся в контексте по умолчанию. Перейдем во вновь созданный контекст:

```
$ docker context use ec2-remote
```

Выполним следующую команду:

```
$ docker run -it --rm redis
```

И проверим результат на сервере:

```
ssh ec2-user@18.130.80.179 docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED
   STATUS          PORTS      NAMES
1b5b0459bf48   redis     "docker-entrypoint.s…"   15 seconds
ago   Up 13 seconds   6379/tcp   peaceful_feynman
```

Однако нет необходимости идти таким сложным путем и запускать эту команду на сервере. Благодаря контексту `ec2-remote` можно выполнить команду `docker ps` локально и получить тот же результат. Контекст продолжает действовать, пока мы снова не переключим его:

```
$ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED
STATUS         PORTS      NAMES
1b5b0459bf48   redis     "docker-entrypoint.s…"   3 minutes ago
Up 3 minutes   6379/tcp   peaceful_feynman
```

Благодаря поддержке контекстов Docker мы можем использовать Docker Compose на удаленном хосте.

Развертывание Compose на удаленных хостах

Давайте переместим пример запуска Redis в файл Compose:

```
services:
  redis:
    image: redis
```

В следующей команде вы должны скорректировать значение переменной `DOCKER_HOST`, потому что в действительности вашему экземпляру EC2 будет выделен другой IP-адрес, но результат должен быть тем же самым:

```
$ DOCKER_HOST="ssh://ec2-user@18.130.80.179" docker compose up
[+] Running 2/2
 Network chapter10_default    Created
0.1s
 Container chapter10-redis-1  Created
0.1s
Attaching to chapter10-redis-1
chapter10-redis-1  | 1:C 22 Jun 2022 22:50:52.725 #
oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
```

Проверив хост, можно убедиться, что экземпляр Redis запущен.

Мы только что использовали контексты Docker, поэтому нам не нужно указывать хост. Итак, попробуем еще раз без окружения `DOCKER_HOST`:

```
$ docker context use ec2-remote
$ docker compose up
[+] Running 2/2
...
```

Здесь мы запустили наше приложение на удаленном хосте на машине EC2. Теперь нам нужно очистить подготовленную нами инфраструктуру, чтобы минимизировать затраты. Удалить инфраструктуру можно вручную через консоль AWS, но, поскольку мы создали инфраструктуру с помощью Terraform, можно воспользоваться командой `destroy`.

Давайте удалим нашу инфраструктуру:

```
AWS_ACCESS_KEY_ID=***
AWS_SECRET_ACCESS_KEY=***
AWS_REGION="eu-west-2"
terraform destroy -var myip=51.241.***.182 -var myvpc=vpc-a8d1b***
```

Рассмотренные нами варианты использования охватывают почти все потребности разработчика. Теперь вы можете разрабатывать необходимый код, подготавливать окружения, развертывать приложения через Compose на удаленном хосте и открывать доступ к ним для других пользователей. Следу-

ющим шагом будет расширение наших возможностей путем развертывания приложения на хосте Docker прямо из IDE.

Удаленное развертывание хоста через IDE

При разработке приложения **интегрированная среда разработки (Integrated Development Environment, IDE)** играет решающую роль в повышении нашей продуктивности. С помощью фреймворка Compose мы можем развертывать и моделировать окружения, поэтому он стал частью нашей повседневной разработки. В этом разделе мы объединим применение IDE и Compose.

Сейчас мы будем использовать IntelliJ IDEA Ultimate Edition (https://www.jetbrains.com/idea/download/#section=mac) в качестве нашей IDE. Ultimate Edition имеет бесплатную пробную версию.

Давайте настроим хост Docker. Сначала загляните в разделы **Preferences** (Настройки), затем **Build** (Сборка), **Execution** (Выполнение), **Deployment** (Развертывание) и **Docker**. Теперь добавим новую конфигурацию Docker, как показано на рис. 10.3.

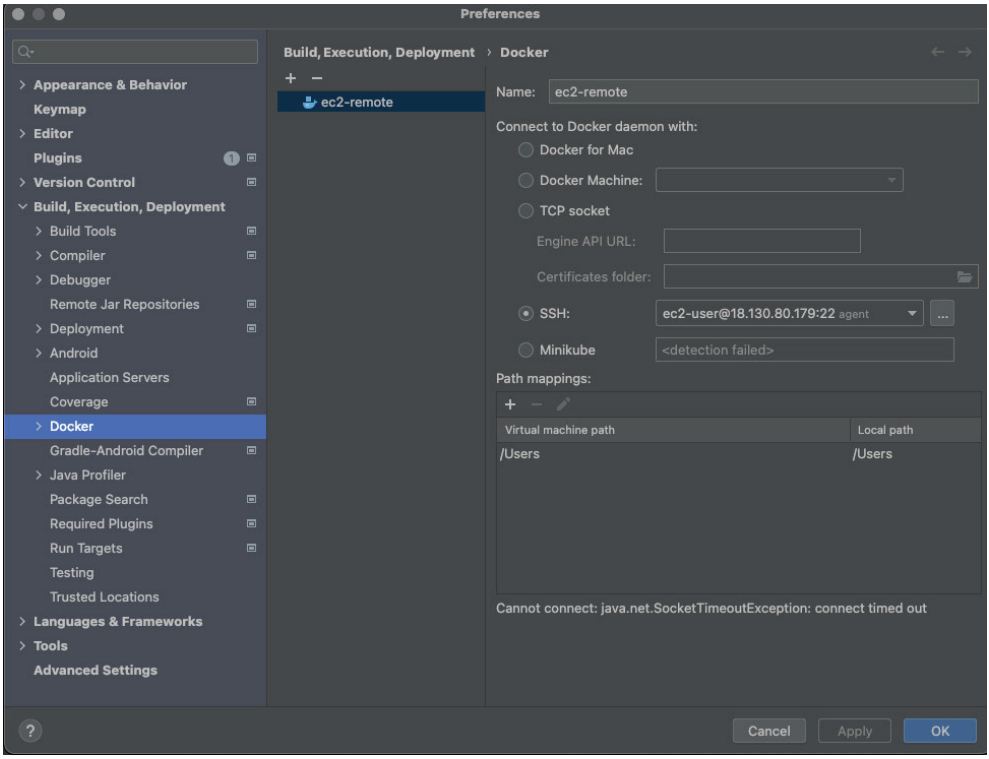

Рис. 10.3 ❖ Конфигурация Docker

Затем, если уже есть файл `docker-compose.yaml`, мы можем запустить его локально (рис. 10.4).

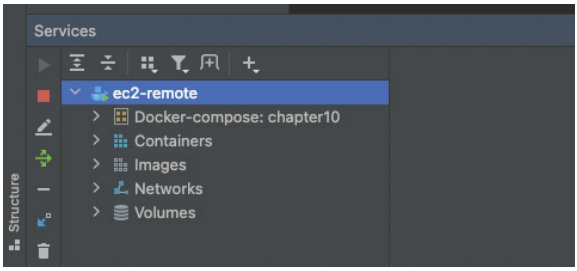

Рис. 10.4 ❖ Запуск Compose

Когда файл Compose запускается из IDE, он будет использовать удаленный хост (рис. 10.5).

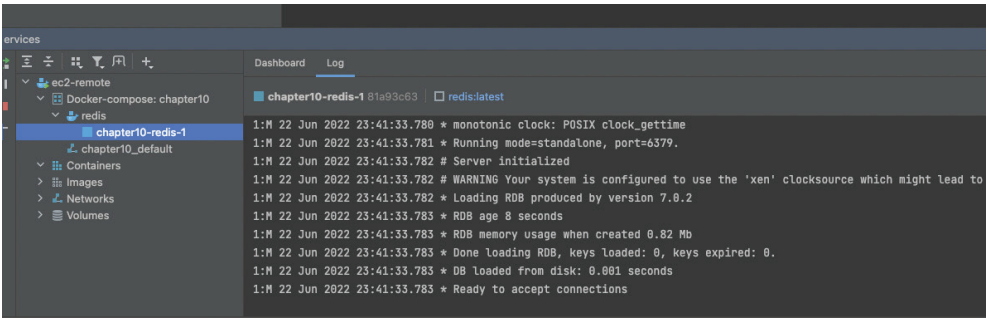

Рис. 10.5 ❖ Журналы

Таким образом, мы можем разрабатывать приложения локально, развертывать их на удаленном хосте и позволить другим людям пользоваться плодами нашего труда.

Итоги

В этой главе мы развернули приложение Compose на удаленном сервере. Это помогло нам использовать ресурсы удаленного сервера и дать возможность другим применять данное приложение через удаленный сервер. Однако это не лучший способ развертывания приложения Compose в промышленном окружении.

В следующей главе мы поговорим о том, как с помощью соответствующих инструментов развернуть приложение Compose в облаке и подготовить его к работе.

Глава 11

Развертывание Docker Compose в AWS

В предыдущей главе мы развернули наше приложение на удаленном хосте Docker. Возможность развертывания на удаленном хосте может помочь во многих ситуациях, например мы можем поделиться приложением с другим человеком или использовать удаленный хост для разработки и тестирования. Развертывание на удаленном хосте приближает нас к контексту развертывания в промышленном окружении. Однако описанный способ развертывание на удаленном узле не соответствует стандартам развертывания в промышленном окружении. Для развертывания в промышленном окружении приложение должно быть высокодоступным, безопасным и находиться за балансировщиком нагрузки, а журналы приложения должны быть легкодоступными и храниться в надежном хранилище.

Эта глава показывает, как запустить приложение Docker Compose в промышленном окружении. **Elastic Container Service (ECS)** – это одна из служб оркестрации контейнеров, предоставляемых AWS. Служба ECS хорошо интегрируется с Docker Compose, поэтому мы можем развернуть существующее приложение Compose через ECS.

Начнем с отправки наших образов Docker в реестр **AWS Elastic Container Registry (ECR)**. Затем внесем некоторые изменения в существующее приложение Compose, чтобы использовать образы из реестра, и выполним развертывание в ECS с использованием профиля Docker для AWS. После развертывания приложения мы перейдем к более сложным концепциям, таким как использование существующего кластера в приватной сети, а также масштабирование и управление секретами.

В этой главе рассматриваются следующие темы:

- введение в AWS ECS;
- размещение образов Docker в AWS ECR;
- развертывание приложения в кластере ECS;
- адаптация файлов Compose для развертывания через ECS;
- запуск приложения Compose в существующей инфраструктуре;
- расширенные концепции Docker Compose в ECS.

Технические требования

Примеры кода для этой книги можно найти в репозитории GitHub по адресу https://github.com/PacktPublishing/A-Developer-s-Essential-Guide-to-Docker-Compose. Если мы будем обновлять код, то все обновления будут доступны в этом репозитории.

Для подготовки ресурсов AWS из командной строки необходимо установить инструмент AWS CLI, доступный по адресу https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html. Инструмент AWS CLI может помочь в решении административных задач, а также с устранением неполадок в инфраструктуре AWS.

В этой главе будут использоваться планы Terraform. Чтобы установить Terraform в вашей системе, следуйте инструкциям из официальной документации (https://learn.hashicorp.com/tutorials/terraform/install-cli).

Введение в AWS ECS

ECS – это механизм оркестрации контейнеров, предоставляемый AWS. ECS можно использовать для развертывания, управления и масштабирования контейнерных приложений. Он предоставляется AWS, поэтому хорошо интегрируется с остальной частью платформы AWS. Приложение, развернутое в ECS, будет использовать балансировщик нагрузки, предоставляющий доступ к приложению. Балансировщик будет запускать приложения в экземплярах EC2, находящихся в **виртуальном приватном облаке (Virtual Private Cloud, VPC)** AWS. Журналы приложения будут доступны через CloudWatch.

ECS включает поддержку AWS Fargate. AWS – это платформа бессерверных вычислений, позволяющая развертывать рабочие нагрузки Docker без необходимости управлять экземплярами EC2 и группами автоматического масштабирования. Если типичная рабочая нагрузка на приложение невелика, не требует небольших накладных расходов, но имеют место редкие всплески запросов, то Fargate сможет дать приложению дополнительные выгоды. Мы используем Fargate для управления нашим приложением, потому что оно все еще находится на стадии прототипа.

Для приложений с более тяжелыми рабочими нагрузками или рабочими нагрузками, предъявляющими особые требования, можно организовать поддержку ECS с помощью EC2 и групп автоматического масштабирования. Например, если приложение интенсивно использует процессор, то имеет смысл создать группу автоматического масштабирования, использующую тип EC2, оптимизированный для вычислений.

Использование контейнеров в локальной системе не вызывает сложностей, и поначалу могут отсутствовать такие проблемы, как распространение образов. Однако для использования созданного нами контейнера необходимо иметь возможность распространять его образ. Для этой цели AWS предоставляет реестр контейнеров. В следующем разделе мы отправим наши образы в реестр, размещенный в AWS.

Размещение образов Docker в AWS ECR

Ранее мы создавали образы для приложений Compose, которые впоследствии сохранялись на хосте Docker и извлекались оттуда. Для использования образов Docker в ECS необходим реестр образов. В AWS есть свой реестр – ECR. Используя ECR, мы получаем возможность отправлять и извлекать образы из реестра и использовать их на любой рабочей станции или сервере, где установлен Docker, при условии что на этой станции или сервере настроен доступ к этому реестру.

ECR – это полностью управляемый реестр контейнеров; высокодоступное решение, поддерживаемое в **AWS Simple Storage Service (S3)**, поэтому образы хранятся в нескольких системах. Кроме того, он обладает такими функциями, как сканирование изображений на наличие уязвимостей. Использование управляемого реестра контейнеров снижает затраты на обслуживание, например избавляет от необходимости выделять сервер или планировать емкость хранилища.

Еще одно преимущество ECR – отличная интеграция с остальными службами AWS. Сервер ECS или функция Lambda, при наличии соответствующих разрешений AWS IAM, может беспрепятственно извлекать образы без какой-либо дополнительной настройки.

Поскольку в роли реестра образов Docker мы будем использовать ECR, продолжим и подготовим наш реестр.

Подготовка ECR с помощью AWS CLI

Для подготовки реестра ECR с помощью AWS CLI достаточно одной команды:

```
$ aws ecr create-repository --repository-name developer-guide-to-compose-ecr
{
    "repository": {
...
     "repositoryUri": "111111111111.dkr.ecr.eu-west-1.
amazonaws.com/developer-guide-to-composer",
...
     "imageScanningConfiguration": {
         "scanOnPush": false
        },
     "encryptionConfiguration": {
         "encryptionType": "AES-56"
        }
    }
}
```

Из полученного вывода можно извлечь ценную информацию. Например, мы будем использовать `repositoryUri` для маркировки наших образов, параметр `imageScanningConfiguration` управляет сканированием образов на наличие уязвимостей, когда включен, а `encryptionConfiguration` определяет спо-

соб шифрования образов в состоянии покоя. Параметр `repositoryUri` имеет числовой префикс. В предыдущем примере указан префикс `111111111111`, но вообще это должен быть номер вашей учетной записи в AWS, поэтому при использовании другой учетной записи он будет другим.

Теперь, создав реестр из командной строки, можно попробовать сделать то же самое с помощью Terraform.

Имейте в виду, что перед созданием нового реестра нужно удалить существующий:

```
$ aws ecr delete-repository --repository-name developer-guide-to-compose-ecr
```

Итак, создадим новый реестр ECR с помощью Terraform!

Создание ECR с помощью Terraform

В предыдущей главе мы использовали Terraform для создания и подготовки инфраструктуры. В этом разделе мы используем этот инструмент для создания реестра Docker.

Вот как это сделать:

```
resource "aws_ecr_repository" "developer_guide_to_compose_ecr"
{
  name                 = "developer-guide-to-compose-ecr"
  image_tag_mutability = "MUTABLE"

  image_scanning_configuration {
    scan_on_push = true
  }
}
```

Теперь можно инициализировать план Terraform:

```
$ AWS_ACCESS_KEY_ID=***
$ AWS_SECRET_ACCESS_KEY=***
$ AWS_DEFAULT_REGION=eu-west-1
$ terraform init
```

А затем приступим к выполнению плана Terraform:

```
$ AWS_ACCESS_KEY_ID=***
$ AWS_SECRET_ACCESS_KEY=***
$ AWS_DEFAULT_REGION=eu-west-1
$ terraform apply
```

Эти команды определяют локальное состояние Terraform. Для общедоступного приложения, с которым взаимодействуют различные инженеры или создают для него инфраструктуру, нужен более подходящий способ хранения и совместного использования состояния.

Хранение файла состояния Terraform

При применении команд плана Terraform для тестирования состояние можно хранить локально. Однако для использования в промышленном окружении состояние должно находиться в хранилище, доступном другим. Давайте попробуем сохранить состояние Terraform в корзине S3.

Перед созданием корзины в AWS следует сказать несколько слов о правилах их именования. Имя корзины должно быть уникальным для всех учетных записей AWS. Это означает, что если какая-то другая учетная запись AWS создала корзину с именем, которое вы решили использовать, то у вас ничего не получится и придется выбрать другое имя.

Теперь создадим корзину:

```
$ AWS_ACCESS_KEY_ID=***
$ AWS_SECRET_ACCESS_KEY=***
$ AWS_DEFAULT_REGION=eu-west-1
$ aws s3api create-bucket --bucket developer-guide-to-compose-state \
  --region eu-west-1 \
  --create-bucket-configurationLocationConstraint=eu-west-1

$ AWS_ACCESS_KEY_ID=***
$ AWS_SECRET_ACCESS_KEY=***
$ AWS_DEFAULT_REGION=eu-west-1
$ aws s3api put-bucket-versioning \
  --bucket developer-guide-to-compose-state \
  --versioning-configurationStatus=Enabled
```

После создания корзины можно настроить Terraform для сохранения состояния в этой корзине:

```
//provider.tf
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.16"
    }
  }
  backend "s3" {
    bucket = "developer-guide-to-compose-state"
    region = "eu-west-1"
    key = "terraform.tfstate"
  }
  required_version = ">= 1.2.0"
}

provider "aws" {
  region  = "eu-west-1"
}
```

После создания корзины в S3 ее имя резервируется. При применении сценария для создания корзины необходимо выбрать другое имя для нее.

Перед запуском `terraform apply` желательно убедиться, что произойдут именно те изменения, которые нам нужны. Для этого можно использовать команду `terraform plan`, которая покажет, какие изменения будут применены к нашей инфраструктуре:

```
$ AWS_ACCESS_KEY_ID=***
$ AWS_SECRET_ACCESS_KEY=***
$ AWS_DEFAULT_REGION=eu-west-1
$ terraform plan
```

Убедившись, что изменения полностью соответствуют нашим ожиданиям, применим план Terraform:

```
$ AWS_ACCESS_KEY_ID=***
$ AWS_SECRET_ACCESS_KEY=***
$ AWS_DEFAULT_REGION=eu-west-1
$ terraform apply
```

Включив версионирование в корзине, можно получить дополнительные выгоды. Помимо истории фиксаций Git, версионирование помогает получить историю изменений состояния.

Создав реестр контейнеров, продолжим и отправим в него свои образы.

Отправка образов в ECR

Прежде чем взаимодействовать с реестром, нужно настроить аутентификацию.

Для аутентификации в ECR мы будем использовать интерфейс командной строки AWS:

```
aws ecr get-login-password \
  --region eu-west-1|docker login \
  --username AWS \
  --password-stdin \
  111111111111.dkr.ecr.eu-west-1.amazonaws.com
```

Эта команда сгенерирует токен для клиента Docker, с помощью которого тот будет подключаться к реестру. Токен генерируется первой частью команды, а вторая часть (после символа конвейера) использует вывод первой части для входа в реестр, который мы создали ранее.

Теперь проверим, работает ли он, и отправим образ:

```
$ docker tag nginx \
111111111111.dkr.ecr.eu-west-1.amazonaws.com/developer-guide-to-compose-ecr:ngnix

$ docker push \
111111111111.dkr.ecr.eu-west-1.amazonaws.com/developer-guide-to-compose-ecr:ngnix

f2089ca22bc1: Pushed
```

```
9e13ccef5ed0: Pushed
9dfe3def52f1: Pushed
7b11943dbe46: Pushed
80730baf8465: Pushed
5978b6b69f17: Pushed
ngnix: digest: sha256:ec2290b7c5d15abb4b3384ad66a89e9c523a4668c057898f3114fa61df4a5586
size: 1570
```

Мы снабдили образ Nginx меткой и отправили его в реестр. С помощью реестра мы можем делиться нашими образами с механизмом оркестрации контейнеров, таким как ECS.

Адаптация образов приложения Compose

Роль приложения в наших экспериментах будет играть приложение диспетчера задач, созданное в главе 5 «Подключение микросервисов друг к другу». Далее мы изменим файл Compose, чтобы передать образы в ECR.

Мы должны изменить образы всех служб:

```
services:
  location-service:
...
    image: 111111111111.dkr.ecr.eu-west-1.amazonaws.com/
developer-guide-to-compose-ecr:location-service_0.1
...
  event-service:
    image: 111111111111.dkr.ecr.eu-west-1.amazonaws.com/
developer-guide-to-compose-ecr:events-service_0.1
...
  task-manager:
...
    image: 111111111111.dkr.ecr.eu-west-1.amazonaws.com/
developer-guide-to-compose-ecr:task-manager_0.1
...
```

Образы, созданные ранее, зависят от платформы, используемой на рабочей станции. ECS поддерживает разные платформы, но мы для простоты развертывания выберем платформу `linux/amd64`.

Укажем платформу для использования перед сборкой:

```
services:
  location-service:
    platform: linux/amd64
   ...
  event-service:
    platform: linux/amd64
   ...
  task-manager:
    platform: linux/amd64
...
```

Теперь мы можем создавать и отправлять образы с помощью Compose:

```
$ docker compose build --no-cache
$ docker compose push
```

Здесь использован параметр `--no-cache`. Это гарантирует, что образы будут собраны для указанной нами платформы. Кроме того, он предотвратит использование кешированных образов из наших предыдущих сборок. Команда `compose push` отправит все наши образы в реестр ECR.

Теперь мы можем продолжить и развернуть приложение в кластере ECS.

Развертывание приложения в кластере ECS

Развертывание в ECS не вызывает затруднений, однако нужно создать профиль в Docker, который будет использовать учетные данные AWS для взаимодействий с AWS и выделения ресурсов.

Соответственно, нам нужно создать контекст Docker для сценариев AWS ECS:

```
$ docker context create ecs guide-to-compose
? Create a Docker context using:  [Use arrows to move, type to filter]
> An existing AWS profile
  AWS secret and token credentials
  AWS environment variables
? Select AWS Profile  [Use arrows to move, type to filter] default
> guide-to-docker-compose
Successfully created ecs context "guide-to-compose"
```

Используя учетные данные, контекст проверит наличие инфраструктуры для приложения Compose и при ее отсутствии попытается подготовить ее. Поэтому важно, чтобы у пользователя/роли, стоящих за профилем AWS, имелись достаточные разрешения для этих действий.

За кулисами Compose сгенерирует и применит шаблон CloudFormation, предоставив необходимую инфраструктуру для запуска приложения.

Мы можем проверить шаблон CloudFormation с помощью команды convert:

```
$ docker --context=guide-to-compose compose convert
AWSTemplateFormatVersion: 2010-09-09
Resources:
  CloudMap:
    Properties:
      Description: Service Map for Docker Compose project aws
      Name: aws.local
      Vpc: vpc-8efaaceb
...
```

CloudFormation – это решение «инфраструктура как код», аналогичное Terraform.

Пример файла можно найти на странице https://github.com/PacktPublishing/A-Developer-s-Essential-Guide-to-Docker-Compose/tree/main/Chapter11/compose-cloudformation.yaml.

При применении шаблона CloudFormation будет создана новая инфраструктура, а если к этому времени инфраструктура уже существовала, то она будет удалена или обновлена, в зависимости от изменений. По сути, на основе служб и различных элементов, определенных с помощью приложения Compose, будет создан файл CloudFormation и применен к существующему стеку, созданному ранее.

Обратите внимание, что в предыдущей команде мы указали контекст с помощью параметра `--context`. Как было показано в главе 10 «Развертывание Docker Compose на удаленных хостах», мы можем установить контекст на постоянной основе. В данном случае это сделано исключительно в иллюстративных целях, потому что мы не будем выполнять большого количества административных действий, а сосредоточимся только на развертывании.

Теперь можно развернуть приложение Compose:

```
$ docker --context=guide-to-compose compose -f compose.backup.yaml up
 LocationserviceService     CreateComplete      56.0s
 EventserviceService        CreateComplete      76.1s
 TaskmanagerTCP8080Listener CreateComplete      2.0s
 TaskmanagerService         CreateComplete      55.1s
...
```

Эта команда создаст приложение Compose и инфраструктуру.

Мы также можем зайти в CloudFormation и проверить ход выполнения приложения (https://eu-west-2.console.aws.amazon.com/cloudformation/home?rfilteringText=&viewNested=true&hideStacks=false#/stacks?filteringStatus=active&filteringText =&viewNested=true&hideStacks=false), как показано на рис. 11.1.

Как видите, стек имеет имя `aws`. Это связано с тем, что он использует то же имя, что и развертывание Compose. В нашем случае это имя совпадает с именем каталога, в котором мы сейчас находимся.

Мы можем изменить имя, явно указав имя проекта:

```
$ docker --context=guide-to-compose compose -p guidetocompose  up
```

После развертывания приложения можно проверить работающие контейнеры с помощью команды `docker compose ps`:

```
$ docker --context=guide-to-compose compose ps
NAME                                        COMMAND
  SERVICE             STATUS              PORTS
task/aws/337ff364e53e4a59b14bdaf65e4fe655   ""
  redis               Running
task/aws/6f08bff5e5ee42a0b2d9995124debd8c
""                  location-service    Running
task/aws/bf76ab48000d49b38017e6b3b4d6b073   ""
  event-service       Running
```

```
task/aws/cea75425182c4e08ad571efcce0c82f2    ""
   task-manager         Running
aws-LoadBal-1Q44XRW9WNYF5-5fdfe7e98727ac85.elb.eu-west-1.
amazonaws.com:8080:8080->8080/tcp
```

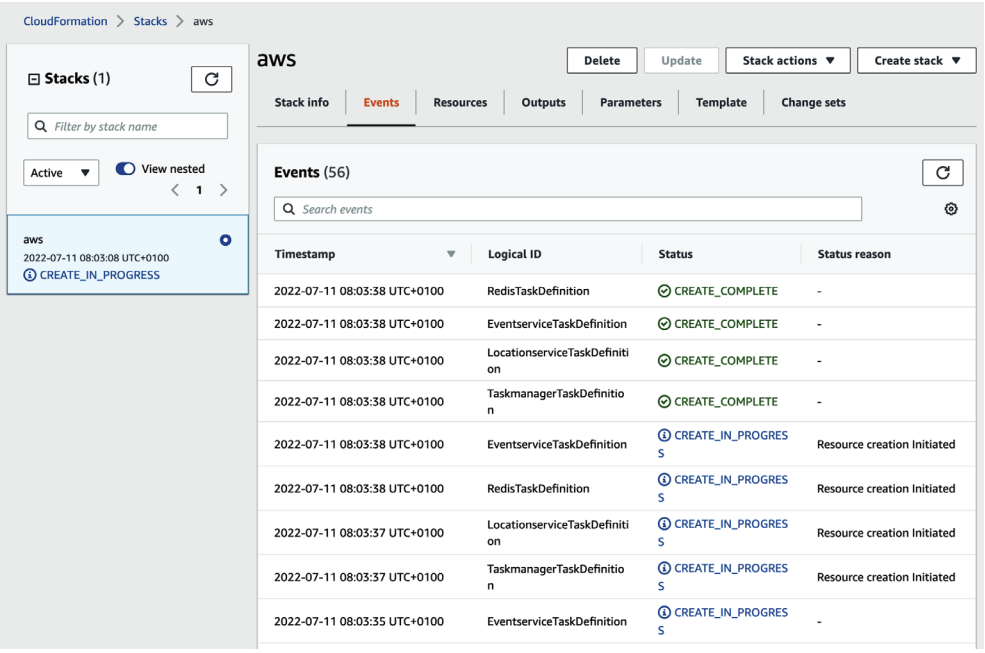

Рис. 11.1 ❖ Ход выполнения CloudFormation

Как видите, приложение диспетчера задач, порт которого мы открыли, работает и доступно через запись DNS.

Выполним тестовый запрос:

```
$ curl aws-LoadBal-1Q44XRW9WNYF5-5fdfe7e98727ac85.elb.eu-west-1.amazonaws.com:8080/ping
pong
```

Мы достигли своей цели и теперь удалим инфраструктуру:

```
$ docker --context=guide-to-compose compose -f compose.backup.yaml down
```

Нам удалось развернуть приложение Compose в AWS ECS и не понадобилось предоставлять какую-либо инфраструктуру, потому что она была создана Compose с использованием учетных данных, настроенных в контексте.

Это вполне подходящий способ развертывания приложения, однако он заставляет создавать новый кластер для каждого приложения. Это может вызвать проблемы с выставлением счетов. К тому же это не самый оптимальный способ запуска приложений, потому что может привести к лишним затратам на обслуживание. В следующем разделе мы развернем Compose в существующем кластере ECS.

Запуск приложения Compose в существующем кластере

Выше мы смогли запустить приложение Compose в ECS с помощью контекста ECS Docker. При развертывании приложения через CloudFormation были созданы новая инфраструктура и полностью новый кластер ECS.

Но если не спешить и заглянуть в файл CloudFormation, то можно увидеть, что были созданы различные компоненты AWS:

- приватный кластер VPC и его подсети;
- группа журналов CloudWatch;
- группы безопасности;
- балансировщик нагрузки;
- CloudMap для обнаружения служб;
- кластер ECS;
- задачи ECS.

По умолчанию CloudFormation использует стандартные VPC и подсети, которые уже существуют в нашей учетной записи AWS. Остается только создать балансировщика нагрузки, группы безопасности и CloudMap для обнаружения служб, а также кластер ECS и задачи ECS. Эти приложения будут развернуты в AWS Fargate.

Очевидно, что эти ресурсы предоставляются автоматически, и мы не можем контролировать их настройки. Но в бизнесе может быть желательно иметь приватную сеть. Кроме того, нам могут понадобиться более строгие правила управления входящим и исходящим трафиками.

Compose дает возможность развернуть приложение в существующем кластере и использовать уже подготовленные ресурсы.

В следующих подразделах мы создадим кластер ECS, а затем внесем изменения в существующее приложение, чтобы оно могло использовать уже имеющуюся инфраструктуру.

Для подготовки инфраструктуры мы используем Terraform.

Создание группы журналов

Допустим, что мы решили хранить журналы нашего приложения в одной группе Cloudwatch. Поэтому создадим группу и поток журналов для нашего приложения:

```
resource "aws_cloudwatch_log_group" "task_api" {
  name              = "/ecs/task-api"
}

resource "aws_cloudwatch_log_stream" "cb_log_stream" {
  name           = "tasl-log-stream"
  log_group_name = aws_cloudwatch_log_group.task_api.name
}
```

Давайте теперь перейдем к другому важному аспекту – настройке сетей.

Создание приватной сети

Мы создадим приватную сеть, охватывающую указанные зоны доступности, для чего создадим подсеть для каждой такой зоны.

План Terraform для VPC:

```
resource "aws_vpc" "compose_vpc" {
  cidr_block = "172.17.0.0/16"
  enable_dns_hostnames = true
  enable_dns_support = true
}
```

Как видите, мы создали VPC и указали значение `cidr_block`. Мы также включили поддержку DNS-имен хостов и DNS. Эта поддержка очень важна для нашего приложения ECS, потому что необходима для взаимодействий между службами.

Создадим подсеть для каждой зоны доступности:

```
resource "aws_subnet" "private_subnet" {
  count = length(var.availability_zones)
  cidr_block        = cidrsubnet(aws_vpc.compose_vpc.cidr_
block, 8, count.index)
  availability_zone = var.availability_zones[count.index]
  vpc_id            = aws_vpc.compose_vpc.id
}

resource "aws_subnet" "public_subnet" {
  count = length(var.availability_zones)
  cidr_block        = cidrsubnet(aws_vpc.compose_vpc.cidr_
block, 8, length(var.availability_zones) + count.index)
  availability_zone = var.availability_zones[count.index]
  vpc_id            = aws_vpc.compose_vpc.id
}
```

Регион указывается при запуске команды `terraform`. В файле, содержащем переменные, можно указать выбранные зоны доступности:

```
variable "availability_zones" {
  type = list(string)
  default = [ "eu-west-1a" ,"eu-west-1b" ]
}
```

Нашему приложению может потребоваться доступ к интернету. Например, мы используем образ Redis, который не развертывали в созданном нами реестре ECR. Чтобы включить доступ к интернету, нужно настроить интернет-шлюз и указать таблицу маршрутизации, направляющую трафик в интернет-шлюз:

```
resource "aws_internet_gateway" "internet_gateway" {
  vpc_id = aws_vpc.compose_vpc.id
}

resource "aws_route" "internet_route" {
  route_table_id         = aws_vpc.compose_vpc.main_route_table_id
  destination_cidr_block = "0.0.0.0/0"
  gateway_id             = aws_internet_gateway.internet_gateway.id
}
```

Такой вариант подходит для экземпляров, развернутых в общедоступной подсети, но не подходит для экземпляров, находящихся в приватной подсети. Чтобы обеспечить подключение к интернету, нужно использовать шлюз с сетевой трансляцией адресов (NAT).

Шлюзу NAT для работы нужен общедоступный IP-адрес, поэтому создадим два таких адреса, по одному для каждой зоны доступности:

```
resource "aws_eip" "nat_ips" {
  count      = length(var.availability_zones)
  vpc        = true
  depends_on = [aws_internet_gateway.internet_gateway]
}
```

Шлюз NAT будет установлен в каждой общедоступной подсети:

```
resource "aws_nat_gateway" "nat_gateway" {
  count         = length(var.availability_zones)
  subnet_id     = element(aws_subnet.public_subnet.*.id, count.index)
  allocation_id = element(aws_eip.nat_ips.*.id, count.index)
}

resource "aws_route_table" "private_route_table" {
  count  = length(var.availability_zones)
  vpc_id = aws_vpc.compose_vpc.id

  route {
    cidr_block     = "0.0.0.0/0"
    nat_gateway_id = element(aws_nat_gateway.nat_gateway.*.id, count.index)
  }
}
```

Наконец, определим таблицу маршрутов для приватных подсетей:

```
resource "aws_route_table_association" "private_association" {
  count          = length(var.availability_zones)
  subnet_id      = element(aws_subnet.private_subnet.*.id, count.index)
  route_table_id = element(aws_route_table.private_route_table.*.id, count.index)
}
```

Это самая важная часть, потому что именно она позволяет использовать приватную сеть в ECS и подключаться к ней.

Группы безопасности

Группы безопасности управляют входящим и исходящим трафиками. Полную конфигурацию групп безопасности можно найти на GitHub. Важным параметром является возможность подключения двух служб Compose.

Разрешим входящий трафик между службами Compose:

```
resource "aws_security_group_rule" "allow_services_
connectivity" {
  type                     = "ingress"
  from_port                = 0
  to_port                  = 0
  protocol                 = "-1"
  source_security_group_id = aws_security_group.compose_security_group.id
  security_group_id        = aws_security_group.compose_security_group.id
}
```

Теперь службы Compose смогут взаимодействовать друг с другом.

Настройка кластера ECS и балансировщика нагрузки

После настройки правил для входящего и исходящего сетевых трафиков можно настроить балансировщик нагрузки и кластер ECS:

```
resource "aws_alb" "compose_alb" {
  name            = "guide-to-compose-load-balancer"
  subnets         = aws_subnet.public_subnet.*.id
  security_groups = [aws_security_group.lb.id]
}

resource "aws_ecs_cluster" "compose_ecs" {
  name = "guide-to-compose-ecs"
}
```

Обновление файла Compose

Давайте адаптируем конфигурацию Compose, чтобы получить возможность использовать существующий кластер, VPC и балансировщик нагрузки:

```
x-aws-vpc: "vpc-0144f03210f0da8e5"
x-aws-cluster: "guide-to-compose-ecs"
x-aws-loadbalancer: "guide-to-compose-load-balancer"

services:
  location-service:
   ...
    logging:
```

```
      options:
        awslogs-group: "/ecs/task-api"
  redis:
    ...
    logging:
      options:
        awslogs-group: "/ecs/task-api"
  task-manager:
    ...
    logging:
      options:
        awslogs-group: "/ecs/task-api"
    links:
      - "redis:redis"
...
```

Здесь мы указали VPC, в котором будут действовать наши рабочие нагрузки, кластер ECS, который будет управлять контейнерами приложения, и балансировщик нагрузки, который будет служить точкой входа в приложение. Теперь развернем приложение во вновь подготовленном кластере.

Запуск приложения Compose в существующей инфраструктуре

Мы подготовили инфраструктуру и настроили приложение Compose, чтобы оно могло использовать существующую инфраструктуру. Для подготовки инфраструктуры будут использоваться те же профиль Docker и команда, что и раньше. Запустим приложение:

```
$ docker --context=guide-to-compose compose \
  -f ./compose.aws.yaml -p guidetocompose up -d
```

Проверим контейнеры приложений:

```
$ docker --context=guide-to-compose compose -f ./compose.aws.
yaml -p guidetocompose ps
task/guide-to-compose-ecs/13ddf3ab03f146cc95b59005c308c5ad
""                  redis               Running
task/guide-to-compose-ecs/af645c4f87ad4a9f8d3aac5fd313bdc5
""                  location-service    Running
task/guide-to-compose-ecs/d96338ba685a401394ba9d61b802
38e6   ""                   task-manager        Running
guide-to-compose-load-balancer-1956561308.eu-west-1.elb.
amazonaws.com:80:80->80/http
$ curl guide-to-compose-load-balancer-1956561308.eu-west-1.elb.
amazonaws.com:80/ping
```

Мы сделали это! Теперь наше приложение работает в приватной сети в кластере ECS, созданном с помощью Terraform.

Когда инфраструктура станет не нужна, не забудьте удалить ее, кластер ECS и другие ресурсы, иначе это приведет к дополнительным расходам.

Сделать это можно командой `terraform destroy`:

```
$ AWS_ACCESS_KEY_ID=***
$ AWS_SECRET_ACCESS_KEY=***
$ AWS_DEFAULT_REGION=eu-west-1
$ terraform destroy
```

Или вручную, из консоли AWS.

Получив дополнительные преимущества от использования компонентов AWS, можно переходить к знакомству с более сложными концепциями приложений Compose, развернутых на ECS. Давайте посмотрим, как можно обновлять и масштабировать наше приложение, а также как настраивать секреты.

Расширенные концепции Docker Compose в ECS

Нам удалось развернуть приложение Compose в выделенном VPC и кластере ECS. Это дает нам больший контроль над приложением и используемыми ресурсами. Облачные приложения имеют целый ряд преимуществ, например возможность последовательного обновления приложения без простоев, масштабирование приложения в соответствии с потребностями, а также эффективное управление секретами.

Обновление приложения

Чтобы обновить приложение, достаточно выполнить команду `docker compose up`. Соответственно, для обновления служб можно использовать ту же самую команду, что и для запуска служб.

Compose поддерживается CloudFormation, поэтому обновление безусловно будет выполнено, но такой подход может привести к простою, если определенный компонент инфраструктуры будет удален и создан заново. Чтобы избежать простоев, необходимо настроить последовательное обновление. При последовательном обновлении экземпляры контейнеров службы обновляются постепенно. Благодаря этому приложение продолжит обслуживать запросы, пока контейнеры по одному обновляются до последней версии. Мы предлагаем глубже погрузиться в масштабирование приложения и настройку непрерывного обновления.

Масштабирование приложения

Поскольку приложение развернуто в механизме оркестрации, мы можем масштабировать реплики служб. У нас будет по две реплики каждой службы, кроме Redis, и мы настроим для них последовательное обновление.

Настроим масштабирование всех служб:

```
  deploy:
    mode: replicated
    replicas: 2
    update_config:
      parallelism: 1
      delay: 10s
      order: start-first
```

Теперь будет запускаться по два экземпляра каждой службы, а ECS будет осуществлять балансировку нагрузки. Кроме того, с помощью `update-config` мы гарантируем обновление служб по одному экземпляру за раз.

Используя `compose ps`, можно увидеть увеличение количества запущенных контейнеров Docker:

```
$ docker --context=guide-to-compose compose -f ./compose.aws.yaml \
  -p guidetocompose ps
NAME
COMMAND             SERVICE
STATUS              PORTS
task/guide-to-compose-ecs/505c7b2c962642459a8b156c4c069f73
""                  redis               Running
task/guide-to-compose-ecs/50c74375216842dba62fea2c19772e55
""                  location-service    Running
task/guide-to-compose-ecs/8b84d785dee14441bb4a3b1808b02419
""                  event-service       Running
task/guide-to-compose-ecs/ac84fe91790244d1836238a40890c54d
""                  event-service       Running
task/guide-to-compose-ecs/b42f9c9b18704dfb888f97112b343e0b
""                  task-manager        Running
guide-to-compose-load-balancer-1757643662.eu-west-1.elb.
amazonaws.com:80:80->80/http
task/guide-to-compose-ecs/bcce928092294fc1bcc57875954307bc
""                  task-manager        Pending
guide-to-compose-load-balancer-1757643662.eu-west-1.elb.
amazonaws.com:80:80->80/http
task/guide-to-compose-ecs/cfd3370bfb61467e9ec15a2a1fc0763c
""                  location-service    Running
```

Вместо увеличения числа реплик вручную можно настроить автомасштабирование:

```
    deploy:
      mode: replicated
      replicas: 2
      update_config:
        parallelism: 1
        delay: 10s
        order: start-first
      x-aws-autoscaling:
        min: 2
        max: 3
        cpu: 75
```

Мы сделали это! Теперь у нас есть как минимум по две реплики каждой службы, и их число будет увеличиваться до трех, если потребление процессора превысит 75 %.

Использование секретов

Секреты – важная часть нашего приложения. Доступ к базе данных или облачным ресурсам должен быть правильно настроен. Compose дает возможность создавать секреты и монтировать их в контейнеры. Смонтируем секрет в диспетчере задач:

```
    secrets:
      - secret-file
    command:
      - /bin/sh
      - -c
      - |
        ls /run/secrets/secret-file
        /task_manager
networks:
  location-network:
  redis-network:
secrets:
  secret-file:
    file: ./secret.file.txt
```

Мы также изменили команду `command`, которая перед запуском приложения диспетчера задач вызывает утилиту `ls`, чтобы убедиться, что файл `secret-file` действительно смонтирован и находится в контейнере. По умолчанию секреты монтируются в каталог `/run/secrets/<secret-name>`.

После запуска контейнера в консоли должен появиться такой вывод:

```
/run/secrets/secret-file[GIN-debug] [WARNING] Creating an
Engine instance with the Logger and Recovery middleware already
attached.
```

В промышленном окружении обращаться с секретами нужно очень осторожно. Локальный файл, монтируемый как секрет, лучше зашифровать. Для шифрования секретов можно использовать KMS и расшифровывать их после развертывания во время инициализации. Например, для безопасного хранения секретов в репозитории с помощью KMS можно использовать инструмент **sops** (https://github.com/mozilla/sops).

Итоги

В этой главе мы успешно развернули наше приложение Compose в AWS. Для этого был создан реестр контейнеров в AWS, куда мы выгрузили свои образы Docker. Затем мы развернули приложение в ECS и подготовили для него новую инфраструктуру. Потом создали приватную сеть и кластер ECS. Наше приложение Compose обладает преимуществами безопасности приватной сети, а созданную инфраструктуру можно повторно использовать для других приложений Compose. Все перечисленное было получено путем адаптации нашего файла Compose и настройки используемой инфраструктуры. Мы рассмотрели более продвинутые концепции развертывания, такие как автоматическое масштабирование и хранение секретов. Адаптировав приложение Compose, мы воспользовались возможностями автоматического масштабирования ECS и функцией последовательного обновления, а также добавили свои секреты в несколько приложений.

В следующей главе мы развернем наше приложение Compose в другом популярном облачном окружении, Microsoft Azure.

Глава 12

Развертывание Docker Compose в Azure

В предыдущей главе мы развернули наше приложение в AWS, добавили возможность автоматического масштабирования и проверку работоспособности и даже организовали доступ к приложению через домен DNS. В этой главе мы сосредоточимся на другом популярном облачном окружении – **Azure**. Azure предоставляет **Azure Container Instances (ACI)** – удобный способ запуска приложений на основе Compose без необходимости управлять какой-либо инфраструктурой. Развертывание в ACI реализуется просто; сначала мы отправим образы приложений в реестр контейнеров Azure, а затем, внеся несколько корректировок, развернем наше приложение в Azure ACI.

Развертывание в ACI требует меньше затрат на обслуживание инфраструктуры и предлагает более простотой синтаксис Compose.

В этой главе рассматриваются следующие темы:

- введение в ACI;
- отправка контейнеров в реестр Azure;
- адаптация файлов Compose для поддержки групп контейнеров Azure;
- развертывание приложения Compose в группах контейнеров Azure.

Технические требования

Примеры кода для этой книги можно найти в репозитории GitHub по адресу https://github.com/PacktPublishing/A-Developer-s-Essential-Guide-to-Docker-Compose. Если мы будем обновлять код, то все обновления будут доступны в этом репозитории.

В этой главе будут использоваться планы Terraform. Чтобы установить Terraform в вашей системе, следуйте инструкциям из официальной документации (https://learn.hashicorp.com/tutorials/terraform/install-cli).

Также необходимо установить `az` – интерфейс командной строки Azure. Инструкции можно найти в официальной документации (https://docs.microsoft.com/ru-ru/cli/azure/).

Введение в ACI

ACI – это бессерверное решение, предоставляемое Microsoft Azure и позволяющее развертывать контейнеры без необходимости управлять какими-либо серверами и инфраструктурой.

Преимущество ACI заключается в возможности запуска нескольких контейнеров без настройки сложного механизма оркестрации Docker.

Однако у ACI есть некоторые ограничения. Например, в зависимости от региона применяются ограничения на максимальное количество процессоров (четыре процессора для каждого региона), а также на объем доступной памяти. Дополнительные сведения об ограничениях можно найти в документации (https://docs.microsoft.com/ru-ru/azure/container-instances/container-instances-region-availability). Еще одно ограничение – недоступность масштабирования контейнеров (https://docs.microsoft.com/ru-ru/azure/container-instances/container-instances-faq#how-do-i-scale-a-container-group-).

Важно помнить об этих ограничениях, потому что они играют решающую роль в понимании пригодности приложения для запуска в ACI. В нашем случае приложение может извлечь определенную выгоду из ACI, потому что его легко развернуть и к нему предъявляются простые требования.

Для развертывания в ACI необходимо обеспечить доступность наших образов для ACI. С этой целью мы используем реестр Docker, предоставляемый окружением Azure.

Отправка контейнеров в реестр Azure

Реестр контейнеров Azure играет важную роль в развертывании приложения. Мы используем общедоступный реестр, хорошо интегрирующийся с инфраструктурой Azure.

Первым шагом к созданию ресурсов в Azure является настройка группы ресурсов. Группу ресурсов Azure можно рассматривать как контейнер, включающий все ресурсы, необходимые для нашего приложения. Группы помогают логически разделить инфраструктуру. Например, удаление группы ресурсов приведет к удалению ресурсов, подготовленных в этой группе ресурсов.

Наш реестр контейнеров Azure, а также ACI будут находиться в группе ресурсов, которую мы настроим далее. Сделать это можно разными способами, например через командную строку, портал Azure, а также с помощью Terraform.

Вот как выглядит добавление группы ресурсов через Terraform:

```
resource "azurerm_resource_group" "guide_to_docker_compose_resource_group" {
  name     = "guidetodockercompose"
  location = "eastus"
}
```

Мы специально выбрали регион `eastus`, так как он имеет больше возможностей и меньше ограничений, в отношении ACI.

Теперь, определив группу ресурсов, можно создать реестр:

```
resource "azurerm_container_registry" "guide_to_docker_compose_registry" {
  name                = "developerguidetocomposeacr"
  resource_group_name = azurerm_resource_group.guide_to_docker_
compose_resource_group.name
  location            = azurerm_resource_group.guide_to_docker_
compose_resource_group.location
  sku                 = "Basic"
  admin_enabled       = false
}
```

Прежде чем приступить к подготовке инфраструктуры с помощью Terraform, нужно пройти этап аутентификации в Azure через командную строку. Это необходимо для Terraform. Один из способов – использовать Azure CLI. Существуют и другие способы аутентификации Terraform в Azure, например с использованием субъекта-службы в Azure Active Directory, ключа или сертификата клиента. Мы же используем команду `az` для простоты.

Аутентификация с помощью az:

```
$ az login
A web browser has been opened at https://login.microsoftonline.
com/organizations/oauth2/v2.0/authorize. Please continue the
login in the web browser. If no web browser is available or if
the web browser fails to open, use device code flow with `az
login --use-device-code`.
[
  {
    ...
    "user": {
      "name": "john@doe",
      "type": "user"
    ...
    }
  }
]
```

После запуска этой команды в браузере откроется страница входа в систему. Пройдя процедуру аутентификации, мы сможем выполнять команды с помощью `az` и запускать планы Terraform в нашей учетной записи Azure.

Используем те же команды, что и в предыдущей главе:

```
$ terraform init
$ terraform apply
```

Terraform подготовит инфраструктуру и разместит свое состояние в нашей локальной файловой системе. Однако есть более управляемый способ хранения состояния Terraform – передать его другим членам команды.

Сохранение файла состояния Terraform

После выполнения предыдущих команд состояние подготовленной инфраструктуры будет сохранено локально. Это не подходит для промышленного окружения, потому что состояние должно физически храниться там, где другие члены команды смогут получить к нему доступ. Поэтому мы используем учетную запись в хранилище Azure.

Учитывая, что мы можем создать несколько групп ресурсов, выделим для наших планов Terraform отдельную учетную запись хранилища, создав ее с помощью сценария:

```
#!/bin/bash
RESOURCE_GROUP_NAME=guide-to-docker-compose-tf
STORAGE_ACCOUNT_NAME=guidetodockercomposetf
CONTAINER_NAME=tfstate
az group create --name $RESOURCE_GROUP_NAME --location northeurope
az storage account create --resource-group $RESOURCE_GROUP_NAME \
  --name $STORAGE_ACCOUNT_NAME --sku Standard_LRS --encryption-services blob
az storage container create --name $CONTAINER_NAME \
  --account-name $STORAGE_ACCOUNT_NAME
```

Теперь изменим провайдера, чтобы состояние сохранялось в учетной записи хранилища:

```
terraform {
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "=2.48.0"
    }
  }
    backend "azurerm" {
        resource_group_name  = "guide-to-docker-compose-tf"
        storage_account_name = "guidetodockercomposetf"
        container_name       = "tfstate"
        key                  = "terraform.tfstate"
    }
}
provider "azurerm" {
  features {}
}
```

Мы создали основу для использования ACI. Теперь у нас есть реестр контейнеров, а также группа ресурсов, в которой мы подготовим наше приложение ACI. Продолжим и адаптируем наше приложение для развертывания в ACI.

Развертывание в ACI

Имея реестр контейнеров, можно выгрузить в него образы Docker, созданные через наше приложение Compose. Конечную точку контейнера можно получить, проверив реестр в учетной записи Azure.

Конечная точка реестра должна иметь вид: `developerguidetocomposeecr.azurecr.io/developer-guide-to-compose`.

После этого можно адаптировать наш файл Compose и указать имена образов контейнеров, которые мы будем отправлять:

```
services:
  location-service:
...
    image: developerguidetocomposeacr.azurecr.io/developer-
guide-to-compose:location-service_0.1
...
  event-service:
...
    image: developerguidetocomposeacr.azurecr.io/developer-
guide-to-compose:events-service_0.1
...
  task-manager:
...
    image: developerguidetocomposeacr.azurecr.io/developer-
guide-to-compose:task-manager_0.1
```

Подставив полный путь к реестру Azure, мы вплотную приблизились к развертыванию; однако из-за особенностей ACI необходимо внести дополнительные коррективы. В частности, существуют некоторые ограничения на порты, которые можно использовать в приложении, а также ограничения в отношении доступных ресурсов.

Одна из необходимых корректировок связана с портами.

Как указано в документации (https://docs.microsoft.com/ru-ru/azure/container-instances/container-instances-container-groups#networking): *«В пределах группы контейнеров экземпляры контейнеров могут взаимодействовать друг с другом через localhost на любом порте, даже если эти порты не предоставляются извне в IP-адресе группы или из контейнера»*.

Если диспетчер задач и служба определения местоположения будут использовать один и тот же порт, то возникнет конфликт и произойдет следующая ошибка:

```
listen tcp :8080: bind: address already in use
```

Чтобы решить эту проблему, настроим диспетчер задач на использование порта 80:

```
  task-manager:
    container_name: task-manager
    ...
    image: developerguidetocomposeacr.azurecr.io/developer-
```

```
guide-to-compose:task-manager_0.1
    ports:
      - 80:80
    environment:
    ...
      - TASK_MANAGER_HOST=:80
```

Это поможет избежать конфликта.

Другой вопрос связан с готовностью некоторых служб. Служба `event-service` попытается получить доступ к службе Redis сразу после запуска, и это может привести к сбою. На момент написания этих строк параметр `depend_on` не поддерживался (https://docs.microsoft.com/ru-ru/azure/app-service/configure-custom-container?pivots=containerlinux#unsupported-options).

Мы можем решить эту проблему, применив творческий подход. Адаптируем службу событий `event-service` и добавим команду `sleep` перед запуском двоичного файла службы:

```
  event-service:
    container_name: event-service
    ...
    command:
      - /bin/sh
      - -c
      - |
        sleep 120
        /events_service
```

Это, пожалуй, самое простое решение, однако есть другие более эффективные обходные пути, включая изменение кода службы и использование механизма проверки работоспособности.

Следующая проблема, которую мы должны решить, – ограничения ресурсов в ACI.

В большинстве регионов ACI ограничивает количество доступных процессоров четырьмя. По умолчанию каждой службе выделяется один процессор. Это ограничение делает наше развертывание неосуществимым, потому что у нас больше четырех служб.

Чтобы решить эту проблему, скорректируем ресурсы, используемые контейнерами в нашем приложении Compose.

Docker Compose позволяет определять ограничения ресурсов для контейнеров служб. К таким ресурсам относятся процессор и память.

Определяя объем памяти в разделе `resources`, например `memory: 512M`, мы требуем выделить контейнеру службы до 512 Мбайт памяти.

Определяя количество процессоров в разделе `resources`, например `cpu: 0.5`, мы сообщаем, что контейнеру службы следует выделить до половины доступного процессорного времени одного ядра.

Определяя ресурсы, которые будут доступны контейнеру, мы можем использовать два параметра: `limits` и `reservations`.

Цель `limits` – ограничить объем используемых ресурсов. То есть если случится всплеск запросов к нашему приложению, для обработки которых по-

требуются дополнительные ресурсы, то эти ресурсы будут ограничены указанными ограничениями.

Определяя параметр `reservations`, мы резервируем минимальный объем ресурсов, необходимых нашему приложению для работы.

Чтобы преодолеть проблему ограничения объема ресурсов, мы настроим объемы используемых ресурсов с помощью Compose:

```
services:
  location-service:
  ...
    deploy:
      resources:
        limits:
          cpus: '0.5'
          memory: 512M
        reservations:
          cpus: '0.5'
          memory: 512M
  event-service:
    ...
    deploy:
      resources:
        limits:
          cpus: '0.5'
          memory: 1024M
        reservations:
          cpus: '0.5'
          memory: 1024M
  task-manager:
    ...
    deploy:
      resources:
        limits:
          cpus: '0.5'
          memory: 1024M
        reservations:
          cpus: '0.5'
          memory: 1024M
...
```

Теперь можно отправить образы и запустить наше приложение. Перед отправкой образов Docker необходимо пройти аутентификацию в недавно созданном реестре Azure.

Сначала войдем в Azure, а потом в реестр:

```
$ az login
$ az acr login --name developerguidetocomposeacr
```

Затем создадим контейнеры и отправим их в реестр:

```
$ docker compose build --no-cache
$ docker compose push
...
```

```
Pushing location-service: cf8204bbc172 Pushing [============
==================================
==>]  557.8MB                                        265.5s
[+] Running 1/43ion-service: db60c013991c Pushed
                             265.5s
...
 Pushing event-service: 24302eb7d908 Pushed
                       269.7s
```

Далее, благополучно выгрузив контейнеры, мы должны сделать так, чтобы Docker мог создавать ресурсы в Azure:

```
$ docker login azure
```

Выполнив эту команду и пройдя аутентификацию, мы дадим Docker возможность взаимодействовать с Azure. Далее создадим контекст Docker для взаимодействий с ранее созданной группой ресурсов Azure:

```
$ docker context create aci guide-to-compose-azure
Using only available subscription : Pay-As-You-Go (c8ce802e-
f8d5-4634-b279-8d203a9c4882)
? Select a resource group  [Use arrows to move, type to filter]
  create a new resource group
> guidetodockercompose (eastus)
```

Покончив с настройками, можно приступать к развертыванию приложения в ACI:

```
$ docker --context=guide-to-compose-azure compose up
[+] Running 5/5
 Group azure       Created                      10.1s
 redis             Created                     144.7s
 event-service     Created                     144.7s
 task-manager      Created                     144.7s
 location-service  Created                     144.7s
```

С помощью `docker compose ps` получим IP-адрес нашего приложения:

```
$ docker --context=guide-to-compose-azure compose ps
NAME                    COMMAND
SERVICE             STATUS              PORTS
azure_event-service     ""                  event-
service       Running
azure_location-service  ""                  location-
service    Running
azure_redis             ""                  redis
Running
azure_task-manager      ""                  task-
manager       Running             20.237.67.76:80->80/tcp:80-
>80/TCP
```

Как определено в настройках портов в Compose, ACI настроит балансировщик нагрузки для переадресации запросов с этого IP-адреса в нашу службу. Каждый порт может принадлежать только одной службе.

Как видите, порт 80 открыт. Дополнительно настроим приложение, чтобы оно было доступно через домен DNS:

```
services:
  task-manager:
    …
    domainname: "developerguidetocompose"
```

После обновления приложения с помощью compose up мы должны получить доступ к службе через DNS:

```
$ docker --context=guide-to-compose-azure compose up
...
$ docker --context=guide-to-compose-azure compose ps
...
azure_task-manager       ""                  task-
manager        Running             developerguidetocompose.
eastus.azurecontainer.io:80->80/tcp:80->80/TCP
```

Теперь наше приложение доступно через DNS. Также можно использовать собственное DNS-имя.

Итак, мы настроили доступ к нашему приложению через DNS, выгрузили контейнеры в реестр Azure и адаптировали приложение для развертывания в ACI.

Итоги

В этой главе мы развернули наше приложение Compose в ACI. Для этого мы создали реестр Docker в Azure и отправили образы Docker в реестр. Затем развернули наше приложение в ACI без подготовки какой-либо инфраструктуры. Внесли некоторые важные изменения, необходимые для правильного запуска приложения, а также протестировали доступ к приложению через DNS.

К настоящему моменту мы рассмотрели запуск Compose в двух облачных окружениях, AWS и Azure, с использованием их механизмов оркестрации контейнеров.

В следующей главе мы познакомимся с еще одним популярным механизмом оркестрации – Kubernetes – и перенесем существующее приложение Compose в окружение Kubernetes.

Глава 13

Миграция на конфигурацию Kubernetes с помощью Compose

В последних нескольких главах мы успешно развернули приложение Compose в двух облачных окружениях: AWS и Azure. Для этого мы подготовили реестры контейнеров, VPC и другие облачные компоненты, такие как балансировщики нагрузки. Затем создали облачные приложения и воспользовались функциями облачных окружений, такими как автоматическое масштабирование и балансировка нагрузки.

В этой главе мы сосредоточимся на миграции нашего приложения Compose на популярный фреймворк оркестрации контейнеров: Kubernetes. За последние несколько лет известность Kubernetes значительно выросла, к тому же он имеет богатую экосистему утилит и инструментов. У современного инженера есть много причин выбрать Kubernetes на роль механизма оркестрации контейнеров. Это не противоречит предназначению Compose и Kubernetes. Compose – это инструмент, упрощающий локальную разработку, а Kubernetes – отличный выбор для организации промышленного окружения. В этой главе мы совершим переход от Compose к Kubernetes: определим компоненты Kubernetes, соответствующие компонентам Compose, и создадим конфигурации развертывания в Kubernetes.

В данной главе рассматриваются следующие темы:

- введение в Kubernetes;
- компоненты Kubernetes и Compose;
- преобразование файлов с помощью Kompose;
- введение в Minikube;
- развертывание приложения в Kubernetes.

Технические требования

Примеры кода для этой книги можно найти в репозитории GitHub по адресу https://github.com/PacktPublishing/A-Developer-s-Essential-Guide-to-Docker-Compose. Если мы будем обновлять код, то все обновления будут доступны в этом репозитории.

Введение в Kubernetes

Kubernetes – это механизм оркестрации контейнеров. Kubernetes позволяет автоматизировать развертывание, масштабирование и управление контейнерными приложениями. Так как это фреймворк с открытым исходным кодом, кластер Kubernetes можно настроить с нуля в центре обработки данных, как в локальном, так и в облачном. Также есть вариант использования управляемого кластера Kubernetes. В связи с растущей популярностью каждый крупный облачный провайдер, такой как AWS, Google Cloud и Azure, предлагает свое решение управляемого Kubernetes.

Развертывая приложение в Kubernetes, мы развертываем его в окружении с различными возможностями, помогающими подготовить приложение и управлять им.

Kubernetes предоставляет надежные и распределенные средства хранения секретов и конфигураций приложений. Предлагает систему проверки работоспособности и готовности развернутого приложения. Может масштабировать подготовленное приложение и балансировать нагрузку. Предоставляет инструменты для мониторинга и отладки. Наконец, помогает организовать взаимодействия между службами, предлагая механизм обнаружения служб.

Учитывая вышесказанное, при развертывании приложения в Kubernetes можно ожидать, что произойдет следующее:

- приложение будет запланировано для развертывания и запущено на узле кластера Kubernetes;
- приложение будет использовать конфигурацию, секреты и переменные окружения, определенные при развертывании;
- Kubernetes будет масштабировать приложение в определенных нами границах количества экземпляров;
- Kubernetes будет следить за работоспособностью и готовностью приложения и заменять контейнеры, прекратившие откликаться;
- Kubernetes будет балансировать нагрузку на приложение между несколькими экземплярами, работающими внутри кластера;
- Kubernetes обеспечит дополнительный уровень безопасности, разрешая и запрещая трафик между приложениями;
- Kubernetes обеспечит обнаружение служб, чтобы приложения могли беспрепятственно связываться с другими приложениями через кластер без ручного вмешательства.

Теперь, узнав немного больше о Kubernetes, давайте посмотрим, как взаимосвязаны компоненты Compose и Kubernetes и как можно осуществить миграцию.

Компоненты Kubernetes и Compose

Для примеров мы использовали довольно простые приложения Compose, но если присмотреться, то в них можно заметить определенные компоненты. Эти компоненты Compose имеют соответствующие им альтернативы в Kubernetes.

Приложения Compose и пространства имен

Как было показано в главе 11 «Развертывание Docker Compose в AWS», в кластере ECS можно разместить несколько приложений Compose. В некотором роде приложение Compose предоставляет способ группировки ресурсов, выделяемых в кластере ECS. В Kubernetes то же самое делается через пространства имен. Пространства помогают разным приложениям совместно использовать кластер, будучи при этом логически изолированными друг от друга.

Службы Compose и службы Kubernetes

Согласно спецификации Compose, служба определяет контекст, поддерживаемый одним или несколькими контейнерами. Как мы уже знаем, определяя службу, можно настроить имена базовых контейнеров.

Эквивалентом служб Compose в Kubernetes является комбинация объектов Pod, Deployment и Service. Pod – это наименьшая вычислительная единица, которую можно развернуть в Kubernetes. Чтобы развернуть Pod, Kubernetes запустит на узле кластера отдельный контейнер с указанной конфигурацией. Этот единственный развертываемый экземпляр ведет нас к определению развертывания – объекту Deployment.

Объект Deployment – это способ декларативного определения приложения и перечня модулей Pod, из которых оно состоит, а также количества реплик для каждого модуля Pod. Поскольку мы коснулись концепции реплик, следующим логическим шагом является определение балансировки нагрузки.

Объекты Service в Kubernetes определяют абстрактный способ доступа к модулям Pod, описываемым в объектах Deployment, изображающих развертывания. В Compose мы можем получить доступ к службе напрямую по ее имени. В Kubernetes нужно определить объект Service, который обеспечит эту абстракцию внутри кластера Kubernetes, и задать в нем единое DNS-имя, используя которое, можно будет взаимодействовать с модулями Pod, составляющими приложение. Кроме того, при применении Service тра-

фик будет балансироваться между модулями Pod, перечисленными в развертывании Deployment.

Метки

В Compose мы можем использовать метки, пары ключ/значение, прикрепленные к компонентам Compose. Метки можно добавлять в службы, образы и ресурсы, определяемые в приложении Compose. Метки можно рассматривать как способ прикрепления метаданных к компонентам Compose. Kubernetes использует аналогичную концепцию меток, но делает еще один шаг вперед. В Kubernetes метки могут использоваться не только для прикрепления метаданных.

В Kubernetes метки можно прикрепить к любому компоненту. Но особенно важной отличительной чертой меток в Kubernetes является их использование. С помощью меток служба Service может идентифицировать модули Pod, в которые следует направлять трафик. С помощью меток можно определить правила, управляющие входящим и исходящим трафиками модулей Pod.

Сети Compose и сетевые политики Kubernetes

В Compose можно определять сети, представляющие каналы связи между службами. Если создать приложение Compose без настроек сетей, то все службы окажутся в одной сети и смогут взаимодействовать друг с другом без ограничений. Если в Compose определить сети, то службы должны находиться в одной сети, чтобы иметь возможность взаимодействовать друг с другом. В Kubernetes это делается с помощью объектов NetworkPolicy, определяющих сетевые политики. Определяя сетевую политику, можно указать правила для входящего и исходящего трафиков, используя блоки IP-адресов, пространства имен или селекторы модулей Pod.

Метки играют в этом ключевую роль, позволяя организовать маршрутизацию трафика между модулями Pod только на основе меток. С другой стороны, поскольку может понадобиться установить связь между несколькими приложениями в кластере Kubernetes, расположенными в разных пространствах имен, маршрутизацию трафика также можно определить с помощью селекторов пространств имен.

Теперь, получив представление о том, как связаны компоненты Compose и Kubernetes, преобразуем наше существующее приложение Compose в приложение Kubernetes.

Преобразование файлов с помощью Kompose

Преобразование существующего приложения Compose в приложение Kubernetes не должно вызвать затруднений. С другой стороны, приложения

Compose имеют гораздо более простую структуру, а развертывание в Kubernetes может стать намного более сложным, потому что в Kubernetes имеется множество своих особенностей и возможностей.

Для демонстрации возьмем приложение, созданное нами в главе 5 «Подключение микросервисов друг к другу», и адаптируем его для развертывания в Kubernetes.

Приложение Compose можно преобразовать в эквивалентный набор ресурсов Kubernetes вручную. С другой стороны, есть такой инструмент, как Kompose, с помощью которого можно сделать то же самое, но с меньшими усилиями.

Устанавливается Kompose просто – все необходимые инструкции вы найдете на странице https://kompose.io/installation/.

Прежде чем перейти к преобразованию приложения, внесем в него некоторые изменения.

Прежде всего добавим версию в файл Compose, как того требует Kompose:

```
version: '3'
```

Другой важный аспект служб Compose – экспорт портов. Экспортируя порты, мы позволяем Kompose определить, какие порты необходимо открыть при развертывании.

В Compose экспортируемый порт становится общедоступным и его можно развернуть в ECS и Azure.

В Kubernetes экспорт порта в развертывании несет также документирующую функцию, точно так же как `EXPOSE` в Dockerfile. Чтобы открыть доступ к модулю Pod в Kubernetes с балансировкой нагрузки, необходимо определить объект службы Service.

Добавим порт, чтобы открыть доступ к Redis:

```
services:
  redis:
    image: redis
    networks:
      - redis-network
    ports:
      - 6379: 6379
...
```

Теперь добавим порт, чтобы открыть доступ к службе определения местоположения:

```
services:
  location-service:
  ...
    ports:
      - 8080:8080
   ...
```

Еще одно необходимое изменение связано с параметром `start_period` механизма проверки работоспособности. В Kubernetes есть контекст провер-

ки работоспособности, а также контекст запросов о готовности. Чтобы не ждать момента, когда можно начинать проверять работоспособность, можно определить запрос для проверки готовности к работе, с помощью которого Kubernetes определит момент, когда приложение будет готово, и начнет производить проверки работоспособности.

Итак, давайте свяжем проверки работоспособности с проверкой готовности:

```
  healthcheck:
    test: ["CMD", "curl", "-f", "http://localhost:8080/ping"]
    interval: 10s
    timeout: 5s
    retries: 5
    start_period: 5s
```

Теперь можно сгенерировать файлы:

```
$ kompose convert
```

Если выполнить эту команду в каталоге, где находится файл Compose, то по ее завершении мы увидим сгенерированные файлы с определениями объектов трех типов:

- развертываниями Deployment;
- службами Service;
- сетевыми политиками NetworkPolicy.

Содержимое и конфигурация нашего приложения будет помещена в определение объекта развертывания Deployment.

Заглянув в файл `task-manager-deployment.yaml`, можно увидеть, что там определены переменные окружения и соответствующие проверки работоспособности. Также обратите внимание, что порт был открыт для доступа извне.

Следующий файл – определение службы Service. В отличие от Compose, где к приложению можно обращаться по имени службы без дополнительных настроек, в Kubernetes для этого необходимо поместить приложение в службу Service.

Следующий файл – определение сетевых политик. Если в Kubernetes установлен сетевой плагин, то управлять трафиком, поступающим из других модулей Pod, можно на основе их меток.

В сетевых политиках для Redis и службы определения местоположения входящий трафик маршрутизируется между модулями Pod с использованием их меток.

Теперь развернем приложение в кластере Kubernetes.

Введение в Minikube

Чтобы запустить и протестировать развертывание Kubernetes локально, можно использовать Minikube. Minikube – это локальный движок Kubernetes, на котором можно развернуть и протестировать приложение Kubernetes.

Minikube не является полнофункциональным кластером Kubernetes, поэтому мы можем протестировать далеко не все. Например, мы не сможем проверить масштабирование базовой группы узлов с увеличением рабочей нагрузки или распределить развертывание приложений по разным зонам доступности. Однако для нашего случая использования этого инструмента более чем достаточно.

В документации на сайте проекта (https://minikube.sigs.k8s.io/docs/start/) вы найдете все необходимые инструкции по установке для своей рабочей станции.

Если у вас уже установлен Minikube и вы хотите начать заново, то просто удалите предыдущий контейнер и запустите новый:

```
$ minikube stop && minikube delete
```

Для проверки сетевых политик требуется сетевой плагин. При его использовании политики вступят в силу, и мы сможем протестировать взаимодействия между службами и управление ими. По умолчанию Minikube не включает сетевой плагин. Чтобы включить наши сетевые политики, мы используем плагин Calligo.

Запустим Minikube:

```
$ minikube start --network-plugin=cni --cni=calico
```

Если в macOS произойдет сбой и вы увидите такое сообщение об ошибке[1]:

```
You might also need to resize the docker driver desktop.
```

– то попробуйте изменить размер образа диска в конфигурации Docker Desktop, как показано на рис. 13.1.

Давайте протестируем Minikube, применив развертывание:

```
$ kubectl create deployment nginx --image=nginx
deployment.apps/nginx created

$ kubectl get pod

NAME                                READY
STATUS    RESTARTS        AGE

nginx-8f458dc5b-z2ctk               1/1     Running   0
55s
```

Теперь можно установить сгенерированные файлы. Следующий шаг – запуск приложения диспетчера задач в Kubernetes.

[1] Перевод: «Измените размер устройства для Docker Desktop». – *Прим. перев.*

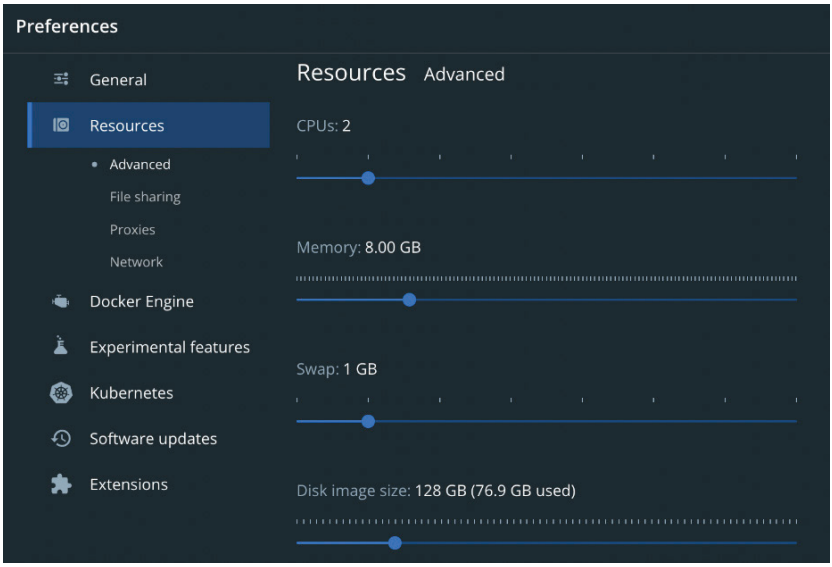

Рис. 13.1 ❖ Изменение размера образа диска

Развертывание в Kubernetes

Получив определение развертывания Deployment для Kubernetes, мы вплотную подошли к нашей главной цели – миграции приложения в Kubernetes. Однако есть еще кое-что, о чем следует позаботиться, и это связано с извлечением образов из реестра. Как мы видели, когда обсуждали ECS и ACI, очень важно, чтобы механизм оркестрации контейнеров мог извлекать образы из реестра. При использовании Minikube нет необходимости создавать реестр.

Мы можем создавать образы и сохранять их в локальном реестре Minikube. Для этого направим наши операции сборки в локальный реестр Minikube.

Мы сделаем это с помощью команды `docker-env`:

```
$ eval $(minikube docker-env)
```

Теперь создадим и развернем эти образы в реестре:

```
$ docker compose build
```

Наличие файлов, сгенерированных ранее, здорово упрощает нашу задачу.

Начнем с развертываний Redis:

```
kubectl apply -f redis-deployment.yaml
kubectl apply -f redis-service.yaml
kubectl apply -f redis-network-networkpolicy.yaml
```

Вслед за Redis развернем и запустим службу событий. Единственная зависимость службы событий – это Redis:

```
kubectl apply -f event-service-deployment.yaml
```

Следующая служба, которая также зависит от Redis, – это служба определения местоположения:

```
kubectl apply -f location-service-deployment.yaml
kubectl apply -f location-service-service.yaml
kubectl apply -f location-network-networkpolicy.yaml
```

И последняя, но не менее важная служба – собственно диспетчер задач:

```
kubectl apply -f task-manager-deployment.yaml
kubectl apply -f task-manager-service.yaml
```

Теперь наше приложение должно было быть полностью развернуто в Kubernetes.

Настроим переадресацию портов диспетчера задач:

```
$ kubectl port-forward svc/task-manager 8080:8080
```

Запустив приложение, создадим задачу, выполнив запрос с помощью `curl`:

```
$ curl --location --request POST '127.0.0.1:8080/task/' \
  --header 'Content-Type: application/json' \
  --data-raw '{
    "id": "8b171ce0-6f7b-4c22-aa6f-8b110c19f83a",
    "name": "A task",
    "description": "A task that need to be executed at the
timestamp specified",
    "timestamp": 1645275972000,
    "location": {
        "id": "1c2e2081-075d-443a-ac20-40bf3b320a6f",
        "name": "Liverpoll Street Station",
        "description": "Station for Tube and National Rail",
        "longitude": -0.081966,
        "latitude": 51.517336
    }
}'
{"created":true,
 "message":"Task Created Successfully",
 "task":{"id":"8b171ce0-6f7b-4c22-aa6f-8b110c19f83a",
         "name":"A task",
         "description":"A task that need to be executed at the timestamp specified",
         "timestamp":1645275972000,
         "location":{"
           id":"1c2e2081-075d-443a-ac20-40bf3b320a6f",
           "name":"Liverpoll Street Station",
           "description":"Station for Tube and National Rail",
           "longitude":-0.081966,
           "latitude":51.517336
         }
  }
}
```

Мы можем проверить работу сетевых политик, изменив настройки.

Теперь применим сетевую политику `deny-all` («запретить все»):

```
$ kubectl create -f - <<EOF
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: deny-all
spec:
  podSelector:
    matchLabels: {}
EOF
```

Удалив сетевую политику Redis, мы не сможем получить доступ к базе данных из модулей Pod:

```
$ kubectl delete -f ./redis-network-networkpolicy.yaml
```

Мы сделали это! Мы только что перенесли наше приложение в Kubernetes! Службы Compose завернуты в объекты Service и развертываются с помощью объектов Deployment. Кроме того, в нашем кластере действуют сетевые политики.

Итоги

В этой главе мы познакомились с фреймворком Kubernetes, некоторыми его базовыми компонентами и соответствиями между компонентами Compose и Kubernetes. Затем мы создали ресурсы Kubernetes, необходимые для работы нашего приложения Compose. Чтобы упростить создание файлов Kubernetes, был использован инструмент Kompose. Далее мы установили Minikube с сетевым плагином Calico. С помощью Minikube мы смогли развернуть наше приложение и протестировать его.

До сих пор мы широко использовали Compose для повседневной разработки, тестирования развертываний в промышленном окружении и в Kubernetes. На данный момент у вас есть все знания, чтобы уверенно использовать Compose в повседневной работе.

Я хотел бы поблагодарить вас за выбор этой книги. Теперь вам решать, где и как применять вновь полученные знания. Будет ли это ваше новое облачное приложение, окружение для заданий CI/CD или локальная среда для вашей команды – решать вам.

Вот и все! Это конец книги. А теперь идите вперед и творите!

Предметный указатель

D

E

G

H

I

K

З

И

К

Л

М

Н

О

П

Р

С

Т

У

Х

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38, оф. 10;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
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: **http://www.galaktika-dmk.com/**.

Эммануил Гадзурас

Docker Compose для разработчика

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Перевод *Яценков В. С.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 17,88. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**