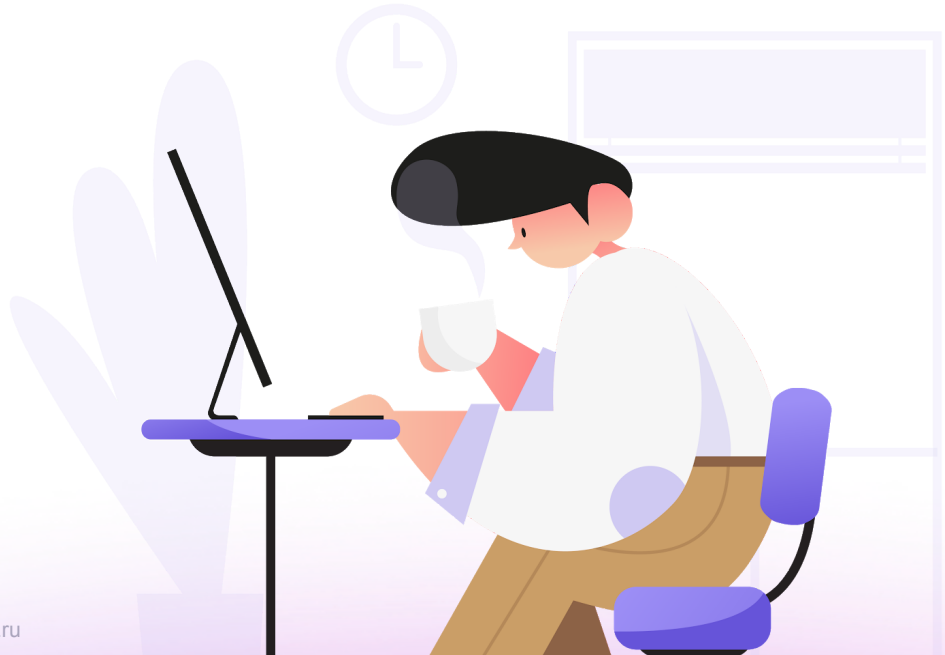
GeekBrains

Урок 1. Особенности докеризации Go-приложений



На этом уроке

1. Кратко повторим то, что мы уже знаем про работу с Docker-контейнерами.
2. Поговорим про разные способы описания Docker-контейнеров для Go-приложений.
3. Рассмотрим важные детали, которые необходимо учесть при докеризации Go-приложений.
4. Попробуем работу с контейнерами для запуска линтеров и тестов.
5. Попробуем инструмент docker-compose для поднятия локального окружения.

Оглавление

Что мы уже знаем про работу с Docker?

Для того, чтобы освежить знания о Docker, давайте повторим термины и понятия, с которыми мы уже сталкивались в предыдущих курсах.

Начнём с понятия образа контейнера.

Образ контейнера - это файл, который можно скачивать из реестра контейнеров, загружать в реестр контейнеров, и который используется для запуска экземпляра контейнера. В случае, если мы говорим о Docker-образах, мы можем утверждать, что каждый такой файл содержит в себе несколько **слоёв**. С физической точки зрения, если мы посмотрим на то, как выглядят Docker-образ, мы увидим, что каждый слой - это тоже отдельный **файл**, который хранится на машине, где запускаются

контейнеры. Интересно, что одни и те же слои могут использоваться разными контейнерами. Если вы уже видели, как выглядит Docker-файл и немного работали с Docker, вы могли заметить, что каждая команда Docker-файла создает один промежуточный слой.

Кроме непосредственно самих слоёв образ также содержит **метаданные** с описанием этих слоев. Каждый образ обычно начинается с базового образа или с базового слоя, это или слой операционной системы. В самом простом случае базовым слоем является, фактически, только ядро Linux и Docker-файл начинается с команды “FROM scratch”.

Также общими будут являться слои, связанные с языками программирования и средами, где мы хотим запускать приложения. Например, пусть у нас есть несколько приложений, написанных на Go. Допустим, компилировать приложения мы хотим тоже в контейнере. Возьмем за основу стандартный образ с DockerHub для языка Go. Для каждого нашего приложения у нас будет отдельный докерфайл, но все они будут использовать один и тот же базовый образ и, соответственно, слои, отвечающие за этот базовый образ, тоже будут общими.

Следующее понятие - это сам контейнер. **Контейнеры** - это запущенные экземпляры образов. Представим, что мы используем Linux. В таком случае контейнер будет представлять собой Linux-процесс. Этот процесс создается с помощью таких системных вызовов, как clone, а также к контейнерам применяются дополнительные правила изоляции за счет, например, использования механизма CGroups.

Ещё одно понятие - это **хост-машина** (или контейнерный хост, хостовая ОС), то есть та физическая или виртуальная машина, на которой запускаются контейнеры. Это может быть ваш ноутбук, виртуальная машина, предоставленная каким-то облачным провайдером, где вы хотите запускать контейнеры, это даже может быть физический сервер.

Давайте вспомним, как работает Docker. Здесь нам понадобятся три составляющих: docker daemon, docker client и реестр контейнеров.

Docker daemon запускается на хост-машине. Обычно его запуском занимается операционная система хост-машины. Docker daemon отвечает за создание и запуск контейнеров, за контроль того, как контейнеры работают, а также за создание и хранение образов. Docker daemon предоставляет HTTP API, поэтому с ним можно работать с помощью клиента, например, через командную строку или графический интерфейс. Вы даже можете написать свой собственный клиент и работать с демоном напрямую.

Docker client - это как раз тот cli-интерфейс, который общается с docker daemon. Когда мы вызываем такие команды, как запуск контейнера или создания образа, мы как раз общаемся с docker daemon с помощью клиента. При этом важно, что один клиент может быть сконфигурирован для работы с несколькими демонами, запущенными на разных машинах. Иногда это может быть очень удобно.

Третий компонент - это **реестр**, или docker registry. Он используется для хранения и распространения образов. Самый популярный реестр контейнеров - это, конечно, DockerHub, с которого часто скачиваются, например, все базовые контейнеры операционных систем. Но в целях безопасности и для устранения накладных расходов можно разместить свой собственный реестр где-то на инфраструктуре своей компании, или воспользоваться решением от облачного провайдера. Также специальные реализации докер-реестров могут содержать дополнительные “фичи”: дополнительные проверки безопасности, автоматизацию сборок и тому подобные удобства.

Давайте теперь чуть-чуть углубимся в работу docker deamon. Под капотом у Docker daemon - драйвер выполнения контейнеров runC. Этот драйвер тесно связан с двумя механизмами ядра Linux - это CGroups и Namespace.

CGroups, уже упомянутый в этом уроке, - это механизм, отвечающий за управление ресурсами, которые используются контейнером. Туда входят, например, процессор и оперативная память. Также этот механизм обеспечивает замораживание и размораживание контейнеров. Вызывая команду “docker pause”, мы можем увидеть, как это работает.

Второй механизм - **Namespace** - пространство имен. Этот механизм отвечает за изоляцию контейнеров. Пространство имен гарантирует, что файловая система или

имя хоста, или тип пользователя, который используется в системе, или всё, что касается сети и процессов контейнера, полностью отделены от хост-системы и от процессов и ресурсов другого контейнера.

Еще одна технология, которая нам здесь может быть важна - **UFS** (Union File System) - это файловая система, которая нам позволяет работать с разными слоями, когда мы работаем с образами и с самими контейнерами.

Образы Docker состоят из нескольких слоев, и каждый слой - это защищенная от записи файловая система, то есть файловая система доступная только для чтения. Для каждой инструкции, для каждой строчки Docker-файла в рамках работы с этой файловой системой создаётся свой слой, который размещается поверх предыдущих слоёв.

Когда мы запускаем контейнер из образа, то есть когда мы выполняем команду “docker run”, Docker выбирает нужный образ и на самом верхнем слое файловой системы этого образа добавляет еще одну файловую систему, ещё один уровень, на котором уже появляется возможность записи. Здесь также происходит настройка дополнительных параметров, таких как IP-адрес, имя, идентификатор контейнера и всё, что связано с ресурсами.

Вообще, контейнер может находиться в любой момент времени в одном из нескольких состояний:

- в процессе перезапуска
- в рабочем состоянии
- в состоянии паузы
- в остановленном состоянии

Надеемся, что в рамках предыдущих курсов вам уже удалось попрактиковаться с этими состояниями и работой с контейнерами вообще. А если нет, то в рамках текущего модуля будет возможность попробовать всё это как на своем локальном окружении, так и на окружении приближенном к продакшн.

“Моностейджинговые” сборки

Как правило, в случае продакшн-окружения мы работаем с виртуальными или физическими машинами на основе Linux. Это значит, что Docker-образ, который мы возьмём за основу для запуска Go-приложений, тоже должен основываться на Linux-дистрибутиве.

Давайте вспомним два важных свойства компилятора Go: кросс-компиляцию и возможность отключить использование библиотеки C-уровня (cgo).

Кросс-компиляция за счет указания переменных окружения GOOS и GOARCH позволяет нам скомпилировать бинарный файл под разные операционные системы и архитектуры независимо от того, в какой операционной системе и архитектуре мы находимся сейчас.

Задание переменной окружения CGO_ENABLED в значение 0 позволяет отключить использование cgo. В большинстве случаев для обычных контейнеризируемых Go-сервисов использование cgo как раз не понадобится. Таким образом, получаемый бинарный файл не будет зависеть от наличия библиотек уровня C в операционной системе, и бинарный файл в этом смысле будет включать в себя уже все необходимые библиотеки.

Что дают нам эти два свойства компилятора в плане докеризации? Если бинарный файл скомпилирован с параметрами GOOS=linux и CGO_ENABLED=0, мы можем скопировать и запустить его вообще в любом Docker-образе, основанном на Linux. Более того, даже образ собранный на основе самого минимального образа (FROM scratch), будет достаточно.

Типовая команда для компиляции в таком случае может выглядеть примерно так:

```
CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build \
     -o bin/myapp ./cmd/myapp
```

И при задании докерфайла нам только остается добавить бинарный файл в образ:

Dockerfile

```
FROM scratch
COPY bin/myapp /
CMD ["/myapp"]
```

Как видно из примера, всего трёх строк в докерфайле может быть достаточно для контейнеризации простейшего Go-приложения.

Обратите внимание на то, что контейнер, собранный таким образом, будет не только максимально легковесным, но и безопасным, так как в нём отсутствуют любые другие бинарные файлы. А значит, даже в случае если злоумышленник получит доступ к такому контейнеру, вряд ли он что-то сможет с ним сделать помимо запуска нашего бинарного файла.

Однако, в большинстве приложений, заточенных для реальной жизни, кроме непосредственно бинарного файла нам могут понадобиться и другие артефакты в образе контейнера. Например, если докеризуемое приложение выполняет https-запросы к каким-либо сервисам, в образе должны присутствовать SSL-сертификаты. Рантайм Go будет искать их по пути /etc/ssl/certs/.

Конечно, такие сертификаты можно подготовить и скопировать по аналогии с тем, как мы готовили и копировали в образ бинарный файл, но есть и более прагматичный способ, о нем мы поговорим в следующей части урока.

Мультистейджинговые сборки

Один и тот же докерфайл может содержать несколько конструкций FROM. Каждая такая конструкция будет открывать новый “стейдж” сборки образа, и сами такие сборки называются “мультистейджинговыми”.

Зачем нужны такие сборки? Представьте себе ситуацию, когда мы собираем контейнер из окружения, где не установлен компилятор Go. Например, на Continuous Integration системе, с которой мы работаем, компилятора может не быть, это нормальная ситуация. В таком случае подход, из предыдущего примера, когда мы заранее готовим бинарный файл, уже не сработает. Теперь нам необходимо включить компиляцию в один из этапов сборки докер-образа.

В таком случае докерфайл может выглядеть, например, как-то так:

Dockerfile

```
FROM golang:1.15

RUN mkdir -p /myapp
ADD . /myapp
WORKDIR /myapp

# Собираем бинарный файл
RUN GOOS=linux GOARCH=amd64 CGO_ENABLED=0 \
   go build -o /myapp ./cmd/myapp

CMD ["/myapp"]
```

Такой вариант сработает, и в результате мы получим готовый образ. Однако, этот образ гораздо более тяжеловесный и уже не такой безопасный, как вариант, собранный из scratch, поскольку помимо нашего бинарного файла в результующем образе будет весь набор приложений операционной системы, компилятор Go и другие полезные приложения.

К счастью, даже используя такую сборку, мы можем вернуться к легковесному scratch. На помощь здесь нам придут как раз мультистейджинговые образы. После

того, как мы описали процесс компиляции, мы можем добавить конструкцию FROM scratch ещё раз, и скопировать бинарный файл в результирующий контейнер.

При таком подходе удобно именовать промежуточные стейджи, для этого в строку FROM у нас добавится инструкция as: `FROM <образ> as <имя стейджа>`.

Результирующий докерфайл будет выглядеть уже так (новые конструкции выделены полужирным):

Dockerfile

```
FROM golang:1.15 as builder

RUN mkdir -p /myapp
ADD . /myapp
WORKDIR /myapp

# Собираем бинарный файл
RUN GOOS=linux GOARCH=amd64 CGO_ENABLED=0 \
   go build -o /myapp ./cmd/myapp

FROM scratch

COPY --from=builder /myapp /myapp

CMD ["/myapp"]
```

Всего пара дополнительных строк позволяет нам вернуться к легковесному образу. Промежуточные образы не будут включены в результирующий.

Помните, в предыдущей части урока мы говорили про SSL-сертификаты для возможности выполнять https-запросы? Давайте добавим их в результирующий образ:

Dockerfile

```
FROM golang:1.15 as builder

RUN mkdir -p /myapp
ADD . /myapp
WORKDIR /myapp
```

```
# Собираем бинарный файл
RUN GOOS=linux GOARCH=amd64 CGO_ENABLED=0 \
   go build -o /myapp ./cmd/myapp

FROM scratch

COPY --from=builder /myapp /myapp
COPY --from=builder /etc/ssl/certs/ /etc/ssl/certs/

CMD ["/myapp"]
```

Бывает, что в результирующем образе помимо нашего исходного приложения, могут всё-таки понадобиться утилиты операционной системы. В таких случаях можно воспользоваться легковесными базовыми образами, такими, как busybox.

В целом, образ, который мы получили, уже неплох, но с точки зрения безопасности можно улучшить его ещё немного. По умолчанию приложения в Docker-контейнере запускаются от привилегированного пользователя root. У такого пользователя может быть слишком много дополнительных прав, и при завладении контейнером, злоумышленник при определенных условиях сможет повредить даже хостовой машине. Если вам интересна тема безопасности, посмотрите доклад Liz Rice “Running with Scissors”, там она подробно демонстрирует такую ситуацию. К счастью, запуск контейнеров от пользователя root можно запретить на уровне политик системы управления контейнерами (например, в Kubernetes). В таком случае развертывание контейнера с пользователем root будет просто невозможным.

Наш текущий Dockerfile в данный момент не содержит ничего специального про определение пользователей, а значит результирующий контейнер запустится именно с пользователем root на борту. Давайте это исправим. На этапе сборки мы создадим нового пользователя myapp (при этом, в файл /etc/passwd будет записана информация о пользователе), а на самом последнем этапе зададим пользователя с помощью директивы USER.

Dockerfile

```
FROM golang:1.15 as builder
```

```
RUN mkdir -p /myapp
ADD . /myapp
WORKDIR /myapp

# Добавляем непривилегированного пользователя
RUN useradd -u 10001 myapp

# Собираем бинарный файл
RUN GOOS=linux GOARCH=amd64 CGO_ENABLED=0 \
   go build -o /myapp ./cmd/myapp

FROM scratch

# Не забываем скопировать /etc/passwd с предыдущего стейджа
COPY --from=builder /etc/passwd /etc/passwd
USER myapp

COPY --from=builder /myapp /myapp
COPY --from=builder /etc/ssl/certs/ /etc/ssl/certs/

CMD ["/myapp"]
```

Такой подход будет уже более безопасным, особенно, в случае, если результирующий образ выглядит сложнее, чем простая конструкция FROM scratch.

Давайте добавим теперь ещё немного оптимизации. Как вы уже знаете, при сборке докер кеширует слои образов, и в случае, если в слое не было изменений, повторная сборка на том же окружении будет проходить быстрее. Давайте воспользуемся этой особенностью для оптимизации работы с зависимостями приложения, объявленными с помощью файла go.mod. В наш докерфайл самым первым стейджем мы добавим загрузку модулей, а на втором этапе компиляции просто скопируем загруженные зависимости. В результате, до тех пор, пока файлы go.mod и go.sum не изменятся, наш первый стейдж останется закешированным, и не нужно будет подтягивать зависимости каждый раз при компиляции приложения. Мы просто будем копировать их из закешированного слоя. Вот так может выглядеть наш результирующий докерфайл:

Dockerfile

```
FROM golang:1.15 as modules
```

```
ADD go.mod go.sum /m/
RUN cd /m && go mod download

FROM golang:1.15 as builder

COPY --from=modules /go/pkg /go/pkg

RUN mkdir -p /myapp
ADD . /myapp
WORKDIR /myapp

# Добавляем непривилегированного пользователя
RUN useradd -u 10001 myapp

# Собираем бинарный файл
RUN GOOS=linux GOARCH=amd64 CGO_ENABLED=0 \
   go build -o /myapp ./cmd/myapp

FROM scratch

# Не забываем скопировать /etc/passwd с предыдущего стейджа
COPY --from=builder /etc/passwd /etc/passwd
USER myapp

COPY --from=builder /myapp /myapp
COPY --from=builder /etc/ssl/certs/ /etc/ssl/certs/

CMD ["/myapp"]
```

Такого докерфайла будет достаточно для большинства простых веб-сервисов. Тем не менее, если вам требуется что-то более сложное, вы всегда можете добавить дополнительные этапы. Однако, например, этапы, включающие в себя синтаксический анализ или тестирование, лучше не включать в докерфайл, описывающий процесс сборки и запуска приложения для продакшн. С точки зрения безопасности и консистентности, процесс проверки кода должен быть отделен от процесса запуска приложения в продакшн. Давайте посмотрим, как задать образ для запуска тестов и синтаксического анализа.

Docker-образы для запуска статического анализа и тестов

На своем локальном окружении мы обычно запускаем тесты и проверки непосредственно с окружениям, например, с помощью таких команд как go test или go lint. Однако, в случае работы с Continuous Integration мы снова можем столкнуться с тем, что нужны нам тулчейн не установлен в среде Continuous Integration. Здесь на помощь нам снова придёт Docker. Фактически, всю цепочку проверок, производимых на этапе сборки приложения, мы можем выполнять в рамках этапов сборки Docker-образа.

Например, простейший докерфайл для запуска тестов может выглядеть как-то так:

testing.dockerfile

```
FROM golang:1.15 as builder

RUN mkdir -p /myapp
ADD . /myapp
WORKDIR /myapp

RUN go test -v ./...
```

Мы можем запустить сборку образа с помощью команды типа *docker build -f testing.dockerfile .*

В случае, если один из тестов упадет, команда go test вернет ненулевой код возврата, и в свою очередь процесс сборки упадет, и от docker build тоже придет ненулевой код возврата. В логе сборки можно будет посмотреть информацию о выполнении команды go test.

Также и для запуска линтеров, например, для golangci-lint, тоже можно использовать аналогичный подход. Воспользуемся стандартным образом для golangci-lint. Кстати, мы можем объединить запуск линтеров и тестов в одну сборку:

testing.dockerfile

```
FROM golangci/golangci-lint:v1.31-alpine

RUN mkdir -p /myapp
ADD . /myapp
WORKDIR /myapp

RUN golangci-lint run --issues-exit-code=1 --deadline=600s ./...

RUN go test -v ./...
```

Еще раз обратите внимание на то, что при использовании этого подхода мы не работаем с контейнерами как таковыми, мы работаем только со сборкой образов. Кстати, этот подход также помогает быстро провести онбординг новых коллег: вместо того, чтобы долго объяснять, как вы запускаете тесты и проверки, и какие инструменты используете, достаточно просто показать такой докерфайл.

Команда docker compose для развертывания зависимостей проекта локально

Как правило, когда мы разрабатываем достаточно сложное приложение, нам приходится иметь дело с зависимостями и подключаемыми ресурсами: базой данных, очередями, хранилищами и другими механизмами. При локальной разработке, конечно, можно подключиться к удаленной базе данных или поставить все необходимые утилиты непосредственно на хостовую машину, но гораздо удобнее использовать для такого запуска инструмент docker compose. Он позволяет как бы “скомпоновать” несколько контейнеров в один запуск. При этом запускаемые контейнеры мы можем описать в конфигурационном файле, и нам не нужно помнить все тонкие параметры, которые нужно задать.

Удобнее всего разбирать работу docker compose на конкретных примерах. Давайте рассмотрим типичный файл для запуска двух контейнеров: приложения и подключаемой к нему базы данных. Создадим файл docker-compose.yml со следующим содержимым:

docker-compose.yml

```
version: "3"
networks:
 myapp:
   external: true

services:
 myapp:
   build:
     context: .
     dockerfile: Dockerfile
   ports:
     - 8080:8080
   environment:
     - DB_URL=postgres://user:pass@myapp-db:5432/myapp?sslmode=disable
     - PORT=8080
   depends_on:
     - myapp-db
   networks:
```

```
    - myapp

myapp-db:
  image: postgres:9.6
  volumes:
    - ./db/init.sql:/docker-entrypoint-initdb.d/init.sql
  ports:
    - 5432:5432
  environment:
    - POSTGRES_USER=user
    - POSTGRES_PASSWORD=pass
  healthcheck:
    test: PGPASSWORD='pass' psql -U user --command='SELECT 1'
    interval: 1s
    timeout: 2s
    retries: 5
  networks:
    - myapp
```

Здесь в группе services мы описали параметры запуска двух контейнеров: myapp и myapp-db. Контейнер myapp будет собран из локального докерфайла, это, собственно, и есть наше приложение. Контейнер myapp-db, база данных PostgreSQL, будет поднят из образа postgres:9.6. Мы также задали порты, которые будут доступны в контейнерах (8080 для приложения и 5432 для базы данных) и общую сеть myapp. Для сети мы поставили параметр external: true для того, чтобы иметь доступ к контейнерам и с хостовой машины тоже.

Обратите внимание на строку подключения к базе данных, заданную в переменной окружения DB_URL. Для контейнера myapp имя хоста базы данных будет совпадать с именем сервиса из файла docker-compose.yml, т.е. оно будет задано в виде myapp-db.

Для контейнера с базой данных мы задали конфигурацию [хелсчека](), успешное выполнение которого должно указывать на то, что “состояние здоровья” этого контейнера в норме. Аналогично можно указать его и для контейнера с сервисом. При конфигурации под продакшн (например, для Kubernetes), мы будем использовать более расширенный подход к хелсчекам. Тем не менее, уже сейчас имеет смысл подумать о том, по каким признакам можно понять, что приложение в порядке.

Здесь можно посмотреть пример приложения, собранного по схеме работы с docker compose. Обратите внимание, что в этом примере помимо контейнера для работы с базой данных мы также используем инструмент FlyWay для запуска миграций, который тоже запускается в отдельном контейнере. Кроме того, например, и веб-интерфейс pgAdmin для работы с базой данных мы тоже можем запускать в контейнере.

Аналогичным образом можно запускать локально с помощью docker-compose вообще любые нужные вам инструменты: стек ELK, Prometheus и Grapaha, Jaeger и так далее.

В этом примере мы рассмотрели основные и самые часто используемые параметры при задании конфигурации для docker compose, однако возможности этого инструмента весьма широки. Сам формат файла регулярно обновляется для поддержки новых фич докер. С полным набором возможностей конфигурации docker compose можно ознакомиться в официальной документации.

Важно при этом понимать, что такой механизм запуска контейнеров не будет надежным для продакшн-окружения. Допустимо использовать docker compose локально или на этапе тестирования приложений. Кстати, о том, как использовать docker compose при тестировании приложений мы ещё поговорим в одном из следующих уроков.

Задание для самостоятельной работы

Попробуйте “докеризировать” Go-приложения, написанные вами ранее. Обратите внимание на то, что в контейнеры можно положить не только сервисы, но и cli-приложения. Для лучшего закрепления навыков можно попрактиковаться и с теми, и с другими видами приложений.

Для приложений, которые работали с подключаемыми ресурсами (например, с базами данных) попробуйте использовать инструмент docker compose.

Если вы уже сталкивались с докеризацией Go-приложений, проведите аудит. Соответствуют ли ваши докерфайлы лучшим практикам? Все ли полезные параметры заданы в файлах конфигурации docker compose?

Дополнительные материалы

1. Статья по полезным особенностям кросс-компиляции Go-приложений
2. Демо проблем, возникающих при запуске контейнера от пользователя root
3. Официальная документация по мультистейджинговым сборкам
4. Официальная документация по Docker compose
5. Официальная документация по файлам конфигурации для Docker compose

Для глубокого погружения в Docker:

6. Книга Docker: Up & Running, 2nd Edition by Karl Matthias, Sean P. Kane - https://www.oreilly.com/library/view/docker-up/9781492036722/
7. Книга Using Docker by Adrian Mouat - https://www.oreilly.com/library/view/using-docker/9781491915752/
8. Книга Accelerating Development Velocity Using Docker: Docker Across Microservices by by Kinnary Jangla - https://www.oreilly.com/library/view/accelerating-development-velocity/9781484239360