База по 8 лабе

Содержание

1 Основы

В современном мире разработки программного обеспечения технологии, которые облегчают процесс создания, развертывания и управления приложениями, становятся все более важными. Одной из таких технологий является контейнеризация, которая позволяет разработчикам упаковывать приложения вместе со всеми их зависимостями в стандартизированные, самодостаточные единицы, которые можно легко передвигать между различными средами. Docker - это одна из самых популярных платформ контейнеризации. Рассмотрим, как это работает.

Контейнеризация - это технология, которая позволяет изолировать приложение и его зависимости в самодостаточный, самостоятельный "контейнер". Этот контейнер можно запустить на любой машине, которая поддерживает технологию контейнеризации, без необходимости устанавливать все зависимости вручную.

Docker - это одна из самых популярных платформ контейнеризации. Он позволяет разработчикам создавать контейнеры, упаковывать их с их приложениями и зависимостями, а затем развертывать эти контейнеры в любой среде, которая поддерживает Docker.

Образ Docker - это шаблон, используемый для создания контейнеров Docker. Образы Docker включают в себя все, что необходимо для запуска приложения, включая код, среду выполнения, библиотеки, переменные окружения и файлы конфигурации. Образы Docker создаются из Dockerfile, который содержит набор инструкций, описывающих, как создать образ.

Контейнер Docker - это запущенный экземпляр образа Docker. Контейнеры Docker изолированы друг от друга и от хост-системы, они имеют собственные файловые системы и не могут взаимодействовать с процессами вне своего контейнера.

Dockerfile - это текстовый файл, который Docker может использовать для автоматического создания образа Docker. Dockerfile содержит инструкции для Docker, которые описывают, как настроить среду, установить зависимости, скопировать код приложения в образ и т.д.

Docker Compose - это инструмент Docker, который позволяет определить и управлять многоконтейнерными приложениями Docker. С его помощью можно определить, какие контейнеры должны быть запущены вместе, как они должны взаимодействовать, и т.д.

В целом, Docker и технология контейнеризации значительно упрощают процесс разработки, тестирования и развертывания приложений, обеспечивая консистентность между различными средами и упрощая управление зависимостями.

2 Основные команды

Docker предоставляет множество команд и флагов для управления контейнерами, образами, сетями и т.д. Вот некоторые из них:

- **docker images**: Показывает список всех образов Docker, которые доступны локально на вашей машине. Флаг -q вернет только идентификаторы образов.
- **docker pull**: Скачивает образ из реестра Docker. Например, `docker pull ubuntu:18.04` скачает образ Ubuntu версии 18.04.
- **docker run**: Запускает контейнер из образа. Например, `docker run -d -p 8080:80 -name my_container my_image` запустит контейнер с именем `my_container` из образа `my_image`, привяжет порт 8080 хоста к порту 80 контейнера и запустит контейнер в фоновом режиме (-d).
- **docker ps**: Показывает список работающих контейнеров. Флаг -a покажет все контейнеры (включая остановленные), а флаг -q вернет только идентификаторы контейнеров.
- **docker stop**: Останавливает один или несколько контейнеров. Например, `docker stop my_container` остановит контейнер с именем `my_container`.
- **docker rm**: Удаляет один или несколько контейнеров. Например, `docker rm my_container` удалит контейнер с именем `my_container`.
- **docker rmi**: Удаляет один или несколько образов. Например, `docker rmi my_image` удалит образ с именем `my_image`.
- **docker build**: Строит образ Docker из Dockerfile. Например, `docker build -t my_image .` построит образ с именем `my_image` из Dockerfile в текущем каталоге.
- **docker exec**: Запускает новую команду в работающем контейнере. Например, `docker exec -it my_container bash` запустит оболочку bash в контейнере `my_container`.
- **docker logs**: Получает логи контейнера. Например, `docker logs my_container` покажет логи контейнера `my_container`.
- **docker volume**: Управляет томами Docker. Например, `docker volume create my_volume` создаст новый том с именем `my_volume`.

- **docker network**: Управляет сетями Docker. Например, `docker network create my_network` создаст новую сеть с именем `my_network`.

3 Volume and Mount

Volume в Docker - это наиболее гибкий способ сохранения данных. Они полностью управляются Docker и хранятся в специальной области файловой системы хоста. Docker обеспечивает команды для создания, копирования и удаления томов.

Преимущества использования томов включают:

- Простота резервного копирования или миграции данных, поскольку данные хранятся в одном месте.
- Высокая производительность чтения и записи, поскольку данные хранятся непосредственно на файловой системе хоста.
- Возможность использования с различными драйверами хранения, что позволяет хранить данные на удаленных серверах или облачных провайдерах.

Основные команды:

- **docker volume create**: Создает новый том. Например, `docker volume create my_vol` создаст новый том с именем `my_vol`.
- **docker volume ls**: Показывает список всех томов.
- **docker volume rm**: Удаляет один или несколько томов. Например, `docker volume rm my_vol` удалит том с именем `my_vol`.
- **docker volume inspect**: Показывает подробную информацию о томе. Например, `docker volume inspect my_vol` покажет информацию о томе `my_vol`.
- **docker volume prune**: Удаляет все неиспользуемые тома. Например, `docker volume prune` удалит все тома, которые не привязаны к контейнерам.

Mount в Docker - это другой способ сохранения данных. Он позволяет привязать определенный каталог на файловой системе хоста к каталогу внутри контейнера. Это означает, что все данные, записанные в этот каталог внутри контейнера, фактически сохраняются на файловой системе хоста.

Преимущества использования монтирования включают:

- Простота обмена файлами между хостом и контейнером.
- Возможность использовать существующие каталоги и файлы на хосте внутри контейнера.
- Возможность использовать один и тот же каталог на хосте для нескольких контейнеров.

Однако использование команды mount в Docker может привести к нескольким проблемам. Во-первых, это может привести к проблемам с безопасностью, поскольку контейнеры могут получить доступ к файлам и директориям на хост-системе, что может быть потенциально опасно.

Во-вторых, это может привести к проблемам с переносимостью. Если вы используете mount для связывания директорий или файлов на хост-системе с контейнером, то эти директории или файлы должны быть доступны на всех системах, где будет запускаться контейнер, что может быть неудобно.

По этим причинам, вместо использования mount рекомендуется использовать Docker volumes. Docker volumes предоставляют более безопасный и удобный способ управления данными в Docker.

4 Сети в Docker

Сети в Docker позволяют контейнерам взаимодействовать друг с другом и с внешним миром. Docker предоставляет несколько типов сетей, каждый из которых предназначен для определенного использования. Вот некоторые из них:

- **bridge**: Стандартная сеть Docker, используемая, если ничего не указано. Каждый контейнер, подключенный к сети bridge, получает свой собственный IP-адрес в этой сети.
- **host**: Удаляет сетевую изоляцию между контейнером и хостом Docker, делая сеть контейнера идентичной сети хоста.
- **none**: Отключает все сетевые интерфейсы внутри контейнера.
- **overlay**: Позволяет создавать сети между несколькими хостами Docker, что полезно для приложений микросервисов.

Вот некоторые основные команды Docker для работы с сетями:

- **docker network create**: Создает новую сеть. Например, `docker network create -driver bridge my_network` создаст новую сеть bridge с именем `my_network`.

- **docker network ls**: Показывает список всех сетей.
- **docker network rm**: Удаляет одну или несколько сетей. Например, `docker network rm my_network` удалит сеть с именем `my_network`.
- **docker network connect**: Подключает контейнер к сети. Например, `docker network connect my_network my_container` подключит контейнер `my_container` к сети `my_network`.
- **docker network disconnect**: Отключает контейнер от сети. Например, `docker network disconnect my_network my_container` отключит контейнер `my_container` от сети `my_network`.
- **docker network inspect**: Показывает подробную информацию о сети. Например, `docker network inspect my_network` покажет информацию о сети `my_network`.

5 Dockerfile

Dockerfile - это текстовый файл, который содержит все команды для создания образа Docker. Используя `docker build`, пользователи могут создавать автоматизированные сборки, которые включают в себя все зависимости и параметры, необходимые для запуска приложения в контейнере.

Вот основные инструкции, которые обычно используются в Dockerfile:

FROM устанавливает базовый образ для последующих инструкций. В Dockerfile должна быть хотя бы одна инструкция FROM.

```
FROM ubuntu:18.04
```

RUN выполняет команду и создает новый слой образа. В основном используется для установки пакетов в контейнер.

```
RUN apt-get update && apt-get install -y git
```

CMD предоставляет значения по умолчанию, которые могут быть включены в команду, когда контейнер запускается. В Dockerfile может быть только одна инструкция CMD. Если указано несколько инструкций CMD, то будет использоваться последняя.

```
CMD ["executable","param1","param2"]
```

В этом примере, когда контейнер запускается из образа, будет выполнена команда `executable` с аргументами `param1` и `param2`.

CMD имеет две формы:

- `CMD ["executable "param1 "param2"]` (exec форма)
- `CMD command param1 param2` (shell форма)

Exec форма предпочтительнее, потому что она не вызывает командную оболочку. Это снижает нагрузку на систему и делает образ более безопасным.

Если Dockerfile также содержит инструкцию `ENTRYPOINT`, то `CMD` будет использоваться как аргументы для `ENTRYPOINT`.

```
ENTRYPOINT ["executable"]
CMD ["param1","param2"]
```

В этом примере, когда контейнер запускается из образа, будет выполнена команда `executable` с аргументами `param1` и `param2`.

Если при запуске контейнера указаны аргументы командной строки, то они заменят значения, указанные в `CMD`.

```
docker run -it image_name param3 param4
```

В этом примере, когда контейнер запускается, будет выполнена команда `executable` (из `ENTRYPOINT`), но с аргументами `param3` и `param4`, а не `param1` и `param2` (из `CMD`).

COPY копирует новые файлы или каталоги из исходного пути и добавляет их в файловую систему контейнера по пути назначения.

```
COPY . /app
```

ADD копирует файлы и каталоги в файловую систему контейнера. ADD имеет функциональность TAR-архивации и автоматического распаковывания.

```
ADD test.tar.gz /app
```

EXPOSE информирует Docker, что контейнер слушает указанный сетевой порт во время выполнения.

```
EXPOSE 8080
```

WORKDIR устанавливает рабочий каталог для следующих инструкций RUN, CMD, ENTRYPOINT, COPY и ADD.

```
WORKDIR /app
```

ENTRYPOINT позволяет контейнеру вести себя как исполняемый файл.

```
ENTRYPOINT ["executable"]
```

Dockerfile позволяет автоматизировать процесс создания образа Docker, что делает его ключевым инструментом для создания контейнеров.

6 Файл .dockerignore

Файл `.dockerignore` используется для исключения файлов и директорий из контекста сборки Docker. Это может быть полезно для ускорения процесса сборки, уменьшения размера образа и предотвращения включения нежелательных или чувствительных файлов в образ.

Файл `.dockerignore` работает аналогично файлу `.gitignore`. Он содержит список шаблонов, которые сопоставляются с файлами и директориями. Если файл или директория совпадает с одним из шаблонов, он исключается из контекста сборки.

Вот пример файла `.dockerignore`:

```
*.log
*.tmp
.DS_Store
node_modules/
```

В этом примере файлы с расширениями `.log` и `.tmp`, файл `.DS_Store` и директория `node_modules` будут исключены из контекста сборки.

Использование файла `.dockerignore` может существенно улучшить эффективность и безопасность процесса сборки Docker.

7 Docker Compose

Docker Compose - это инструмент для определения и запуска многоконтейнерных приложений Docker. С его помощью вы можете использовать файл YAML для конфигурации сервисов вашего приложения, а затем с помощью одной команды создать и запустить все сервисы из вашей конфигурации.

Вот некоторые основные команды Docker Compose:

- **docker-compose up**: Запускает все сервисы, описанные в файле docker-compose.yml. Например, `docker-compose up -d` запустит все сервисы в фоновом режиме.

- **docker-compose down**: Останавливает и удаляет контейнеры, сети, тома и сервисы, определенные в файле docker-compose.yml. Например, `docker-compose down` остановит и удалит все ресурсы, созданные командой `docker-compose up`.

- **docker-compose ps**: Показывает текущее состояние запущенных сервисов. Например, `docker-compose ps` покажет состояние сервисов для текущего проекта.

- **docker-compose build**: Собирает образы для сервисов, описанных в файле docker-compose.yml. Например, `docker-compose build` соберет образы для всех сервисов, которые еще не были собраны, или для которых файл Dockerfile был изменен.

Docker Compose делает работу с многоконтейнерными приложениями Docker проще и более удобной, позволяя вам управлять сложными структурами с помощью простых команд.

8 Структура файла Docker Compose

Файл Docker Compose обычно называется `docker-compose.yml` и использует формат YAML. Он содержит информацию о сервисах, сетях и томах, которые должны быть созданы и запущены.

Вот основные элементы файла Docker Compose:

- **version**: Указывает версию синтаксиса Docker Compose. Например, `version: '3'`.

- **services**: Определяет сервисы, которые будут запущены. Каждый сервис представляет собой приложение в контейнере, которое можно запустить вместе с другими сервисами.

- **networks**: Определяет сети, которые будут использоваться сервисами. Сети Docker позволяют контейнерам общаться друг с другом.

- **volumes**: Определяет тома, которые будут использоваться сервисами. Тома Docker позволяют сохранять данные вне жизненного цикла контейнеров.

Вот пример простого файла Docker Compose:

```
version: '3'
services:
  web:
    image: nginx:latest
    ports:
      - "80:80"
networks:
  default:
volumes:
  mydata:
```

В этом примере определен один сервис (`web`), который использует образ `nginx:latest` и пробрасывает порт 80 из контейнера на порт 80 хоста. Также определены стандартная сеть и том `mydata`.

9 Continuous Integration и Continuous Delivery (CI/CD)

Continuous Integration (CI) и Continuous Delivery (CD) — это методология разработки программного обеспечения, которая включает в себя регулярное слияние рабочих копий в общий репозиторий и автоматическую доставку обновлений программного обеспечения заинтересованным сторонам. Это позволяет командам разработки быстрее и эффективнее тестировать и выпускать новые версии программного обеспечения.

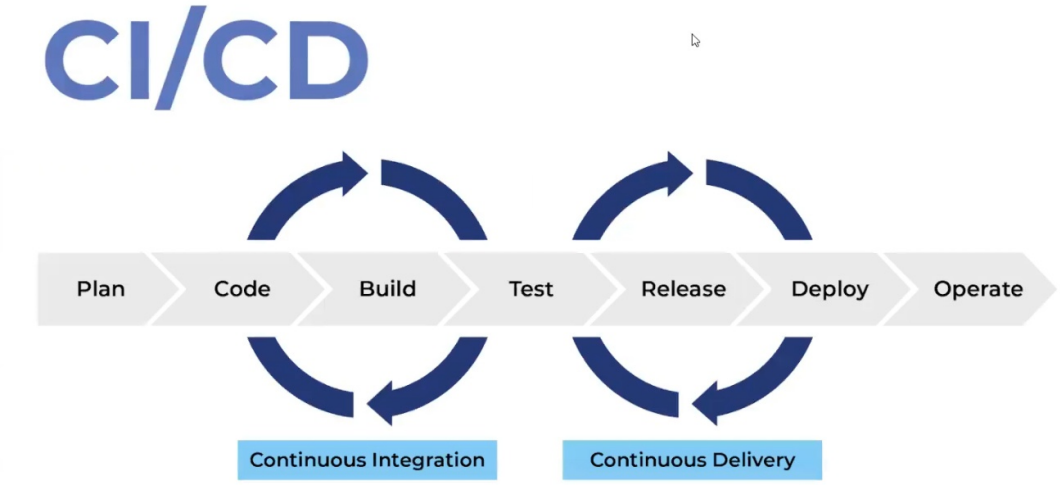
Continuous Integration (CI) — это практика разработки программного обеспечения, при которой разработчики регулярно объединяют свои изменения кода в общий репозиторий. После этого автоматически выполняются различные тесты и проверки, чтобы обеспечить корректность и совместимость нового кода.

Цель CI — обнаруживать и исправлять проблемы с кодом как можно раньше. Это уменьшает вероятность возникновения сложных конфликтов при слиянии кода и позволяет командам разработки быстрее и эффективнее работать вместе.

Continuous Delivery (CD) — это практика разработки программного обеспечения, при которой новые версии программного обеспечения автоматически готовятся к выпуску в продакшн. Это включает в себя автоматическую сборку, тестирование и развертывание кода в различных окружениях.

Цель CD — ускорить процесс доставки обновлений программного обеспечения пользователям. Это позволяет командам разработки быстрее реа-

гировать на изменения требований и улучшать продукт на основе обратной связи от пользователей.



CI/CD предлагает ряд преимуществ для команд разработки и бизнеса в целом. Он ускоряет процесс разработки и доставки новых версий программного обеспечения, улучшает качество продукта за счет раннего обнаружения и исправления ошибок, и позволяет командам быстрее реагировать на изменения требований. В результате, команды могут быстрее и эффективнее доставлять высококачественные продукты и услуги своим пользователям.