Лабораторная работа № 7

Тема работы:

Создание контейнеров. Виртуализация в Docker.

Теоретическая часть:

Контейнеризация — метод виртуализации, при котором ядро операционной системы поддерживает несколько изолированных экземпляров пространства пользователя вместо одного. Эти экземпляры (обычно называемые контейнерами или зонами) с точки зрения выполняемых в них процессов идентичны отдельному экземпляру операционной системы. Для систем на базе Unix эта технология похожа на улучшенную реализацию механизма **chroot** (операция изменения корневого каталога в Unix-подобных операционных системах.) Ядро обеспечивает полную изолированность контейнеров, поэтому программы из разных контейнеров не могут воздействовать друг на друга. В отличие от аппаратной виртуализации, при которой эмулируется аппаратное окружение и может быть запущен широкий спектр гостевых операционных систем, в контейнере может быть запущен экземпляр операционной системы только с тем же ядром, что и у хостовой операционной системы (все контейнеры узла используют общее ядро). При этом при контейнеризации отсутствуют дополнительные ресурсные накладные расходы на эмуляцию виртуального оборудования и запуск полноценного экземпляра операционной системы, характерные при аппаратной виртуализации.

Существуют реализации, ориентированные на создание практически полноценных экземпляров операционных систем (**Solaris Containers**, контейнеры **Virtuozzo**, **OpenVZ**), так и варианты, фокусирующиеся на изоляции отдельных сервисов с минимальным операционным окружением (**jail**, **Docker**).

Docker — программное обеспечение для автоматизации развёртывания и управления приложениями в средах с поддержкой контейнеризации, контейнеризатор приложений. Позволяет «упаковать» приложение со всем его окружением и зависимостями в контейнер, который может быть развёрнут на любой Linux-системе с поддержкой контрольных групп в ядре, а также предоставляет набор команд для управления этими контейнерами. Программное обеспечение функционирует в среде **Linux** с ядром, поддерживающим контрольные группы и изоляцию пространств имён (namespaces); существуют сборки только для платформ **x86-64** и **ARM**.

Для экономии пространства хранения проект использует файловую систему **Aufs** (альтернативная версия UnionFS, вспомогательной файловой системы, выполняющей каскадно-объединённое монтирование – позволяющее одновременно монтировать несколько файловых систем как одну: для файловых систем Linux) с поддержкой технологии каскадно-объединённого монтирования: контейнеры используют образ базовой операционной системы, а изменения записываются в отдельную область. Также

поддерживается размещение контейнеров в файловой системе **Btrfs** (файловая система для Linux, основанная на структурах B-деревьев и работающая по принципу «копирование при записи» (copy-on-write)) с включённым режимом копирования при записи.

В состав программных средств входит демон — сервер контейнеров (запускается командой **`docker -d`**), клиентские средства, позволяющие из интерфейса командной строки управлять образами и контейнерами, а также API, позволяющий в стиле REST управлять контейнерами программно. Демон обеспечивает полную изоляцию запускаемых на узле контейнеров на уровне файловой системы (у каждого контейнера собственная корневая файловая система), на уровне процессов (процессы имеют доступ только к собственной файловой системе контейнера, а ресурсы разделены средствами **libcontainer**), на уровне сети (каждый контейнер имеет доступ только к привязанному к нему сетевому пространству имён и соответствующим виртуальным сетевым интерфейсам).

Набор клиентских средств позволяет запускать процессы в новых контейнерах (**`docker run`**), останавливать и запускать контейнеры (**`docker stop`** и **`docker start`**), приостанавливать и возобновлять процессы в контейнерах (**`docker pause`** и **`docker unpause`**). Серия команд позволяет осуществлять мониторинг запущенных процессов (**`docker ps`** по аналогии с **ps** в Unix-системах, **`docker top`** по аналогии с **top** и другие). Новые образы возможно создавать из специального сценарного файла (**`docker build`**, файл сценария носит название **Dockerfile**), возможно записать все изменения, сделанные в контейнере, в новый образ (**`docker commit`**). Все команды могут работать как с docker-демоном локальной системы, так и с любым сервером Docker, доступным по сети. Кроме того, в интерфейсе командной строки встроены возможности по взаимодействию с публичным репозиторием **Docker Hub**, в котором размещены предварительно собранные образы приложений, например, команда **`docker search`** позволяет осуществить поиск среди размещённых в нём образов, образы можно скачивать в локальную систему (**`docker pull`**), возможно также отправить локально собранные образы в Docker Hub (**`docker push`**).

Также Docker имеет пакетный менеджер **Docker Compose**, позволяющий описывать и запускать многоконтейнерные приложения; конфигурационные файлы для него описываются на языке **YAML**.

Источник: Википедия

Dockerfile

Docker может собирать образы автоматически, читая инструкции из Dockerfile. Dockerfile — это текстовый документ, содержащий все команды, которые пользователь может вызвать в командной строке для сборки образа. На этой странице описаны команды, которые можно использовать в Dockerfile.

Формат Dockerfile:

```
# Comment

INSTRUCTION arguments
```

Инструкции не чувствительны к регистру. Однако принято, чтобы они были в UPPERCASE, чтобы легче было отличить их от аргументов. Docker выполняет инструкции в Dockerfile по порядку. Dockerfile должен начинаться с инструкции **FROM**. Она может располагаться после директив парсера, комментариев и глобальных **ARG**. Инструкция FROM указывает родительский образ, из которого выполняется сборка. Инструкции FROM может предшествовать только одна или несколько инструкций ARG, которые объявляют аргументы, используемые в строках FROM в Dockerfile.

Docker рассматривает строки, начинающиеся с #, как комментарий, если только эта строка не является корректной директивой синтаксического анализатора. Маркер # в любом другом месте строки рассматривается как аргумент. Это позволяет использовать такие выражения, как:

```
# Comment

RUN echo 'we are running some # of cool things'
```

Инструкции Dockerfile:

- **FROM**

```
FROM [--platform=<platform>] <image> [AS <name>]

FROM [--platform=<platform>] <image>[:<tag>] [AS <name>]

FROM [--platform=<platform>] <image>[@<digest>] [AS <name>]
```

Инструкция FROM инициализирует новый этап сборки и задает базовый образ для последующих инструкций. Поэтому правильный Dockerfile должен начинаться с инструкции FROM. В качестве образа может использоваться любой подходящий образ - особенно удобно начинать с образа из публичных репозиториев.

- **RUN**

`RUN <command>` (shell-форма, команда выполняется в оболочке, которой по умолчанию является **/bin/sh -c** в Linux или **cmd /S /C** в Windows)

`RUN ["executable", "param1", "param2"]` (exec-форма)

Команда RUN выполнит все команды в новом слое поверх текущего образа и зафиксирует результаты. Полученный зафиксированный образ будет использован для следующего шага в Dockerfile.

`RUN --mount` позволяет создавать монтирования файловых систем, к которым сборка может получить доступ.

```
RUN --mount=[type=<TYPE>][,option=<value>[,option=<value>]...]
```

Типы монтирования:

bind (по умолчанию) – Bind-mount контекстных каталогов (только для чтения).

cache – Монтировать временный каталог для кэширования каталогов для компиляторов и менеджеров пакетов.

secret – Разрешить контейнеру сборки доступ к защищенным файлам, таким как закрытые ключи, не запекая их в образ.

ssh – Разрешить контейнеру сборки доступ к SSH-ключам через SSH-агенты, с поддержкой парольных фраз.

`RUN --network` позволяет контролировать, в какой сетевой среде выполняется команда.

`RUN --network=<TYPE>`

Типы сетей:

default (по умолчанию) – Запуск в сети по умолчанию.

none – Запуск без доступа к сети.

host – Запуск в сетевом окружении хоста.

- **CMD**

`CMD ["executable","param1","param2"]` (форма exec, эта форма является предпочтительной)

`CMD ["param1","param2"]` (как параметры по умолчанию для **ENTRYPOINT**)

`CMD command param1 param2` (shell-форма)

В Dockerfile может быть только одна инструкция CMD. Если перечислено более одной CMD, то в действие вступит только последняя CMD. Основное назначение CMD заключается в предоставлении настроек по умолчанию для исполняемого контейнера. Эти параметры по умолчанию могут включать исполняемый файл, а могут и не включать его, в этом случае необходимо также указать инструкцию ENTRYPOINT. Если CMD используется для предоставления аргументов по умолчанию для инструкции ENTRYPOINT, то и инструкция CMD, и инструкция ENTRYPOINT должны быть заданы в формате массива JSON.

- **COPY**

`COPY [--chown=<user>:<group>] [--chmod=<perms>] <src>... <dest>`

`COPY [--chown=<user>:<group>] [--chmod=<perms>] ["<src>",... "<dest>"]`

Инструкция COPY копирует новые файлы или каталоги из <src> и добавляет их в файловую систему контейнера по пути <dest>. Можно указать несколько ресурсов <src>, но пути к файлам и каталогам будут интерпретироваться как относительные к источнику контекста сборки.

- **EXPOSE**

`EXPOSE <port> [<port>/<protocol>...]`

Инструкция EXPOSE сообщает Docker, что контейнер прослушивает указанные сетевые порты во время выполнения программы. Можно указать, какой порт прослушивается - TCP или UDP, а если протокол не указан, то по умолчанию используется TCP.

Чтобы задействовать как TCP, так и UDP, включите две строки:

```
EXPOSE 80/tcp

EXPOSE 80/udp
```

Переведено с помощью www.DeepL.com/Translator (бесплатная версия)

Источник: https://docs.docker.com/engine/reference/builder/

Справочник по командам терминала Linux:

https://linuxjourney.com/lesson/the-shell

Основное задание:

Подготовить виртуальную машину и в ней запустить Вашу систему, разработанную в процессе выполнения предыдущих лабораторных работ.

- **Легкий вариант:** запустить разработанную Вами систему и продемонстрировать работу ее компонентов.
- **Средний вариант:** запустить разработанную Вами систему и продемонстрировать работу ее компонентов во взаимосвязи или реализацию задачи писателей и читателей.
- **Сложный вариант:** запустить разработанную Вами систему и продемонстрировать работу в реализации задачи спящего парикмахера.

В парикмахерской имеется 1 парикмахер и n стульев для ожидающих клиентов.

Если v - вариант, а s - номер группы, то задача будет иметь вид:

Если v<=5, то n= 2*v+5+2*s;

Если v>5, то n= 2*v-6+s;

- когда нет ожидающих клиентов, парикмахер сидит на стуле и спит
- когда появляется клиент, ему приходится будить парикмахера
- если в то время, когда парикмахер стрижет клиента, появляется другая клиентка, то ей придется подождать на стуле (если он свободен), если она не покинет парикмахерскую.
- как номер варианта брать самый большой номер человека из подгруппы

По завершению работы, составьте отчет, в котором должно быть – Ваша фамилия, имя, группа, тема работы, Ваш вариант для реализации задания, краткое описание реализации задания, ссылку на исходный код на GitHub. Исходный код push-ите в вашу ветку в соответствующем репозитории - https://github.com/FCIM-SO/Practice-Work-RU. Сохранить отчет в формате PDF и отправить на ELSE - https://else.fcim.utm.md/mod/assign/view.php?id=43459.