Packaging Java for VMs and Containers

Packaging Java for VMs

- Java-приложения упакованы и развернуты на виртуальных машинах (JVM)
- JVM предоставляет среду для выполнения Java-байткода
- Опции упаковки Java-приложения:
  - JAR (Java Archive), WAR (Web Application Archive), EAR (Enterprise Archive)
    - Эти форматы обычно содержат скомпилированный байт-код Java с ресурсами, библиотеками и файлами конфигурации, необходимыми для приложения
- Управление зависимостями (Dependency Injection)
  - Зависимости приложения, такие как внешние библиотеки и т.д., обычно упаковывается в артефакт. Это гарантирует, что приложение имеет доступ ко всем необходимым зависимостям во время выполнения
- ‘Развертывание/разворачивание’ (Deployment)
  - Java-приложение обычно разворачивается через JAR-артефакт. Затем JVM выполняет байт-код Java и запускает приложение

Uber JARs (aka Fat JARs)

- Это гарантирует, что все необходимые зависимости объединены вместе для лёгкого развёртывания и выполнения приложения
- *Плюсы*
  - Легко разворачивать такие приложения в разных средах
  - Портативность – можно не полагаться на внешние зависимости, так как всё уже упаковано
- *Минусы*
  - Артефакт, включающий в себя не только скомпилированный байт-код Java-приложения, но и все его зависимости (внешние библиотеки, фреймворки и т.д.)
  - Страдает размер артефакта (JAR). Увеличение времени развертывания и увеличение требований к хранилищу
  - 0 динамики: если приложению требуется динамическая загрузка или обнаружение зависимостей во время выполнения, с fat JAR это невозможно
- Альтернативы
  - Контейнеры (Docker, Kubernetes)
  - Dependency Injection (Maven, Gradle, etc.)

Как создать Fat JAR

- Инструменты сборки (Maven, Gradle) – их плагины мержат зависимости в один большой артефакт, JAR-файл
- Maven Shade Plugin
  - Может перемещать (т. е. изменять имена пакетов) зависимости, чтобы избежать конфликтов, и объединять их в итоговый JAR. Например, таким образом можно разрешить конфликт версий одной и той же библиотеки
- Gradle Shadow Plugin

Packaging Java for Containers

- Контейнеры – изолированная и облегченная среда для работы приложения. Контейнеры инкапсулируют приложение и его зависимости
- Java-приложения для контейнеров обычно упаковываются с использованием технологий контейнеризации, как Docker.
  - Приложение вместе со своими зависимостями и средой выполнения объединено в образ контейнера. Образ контейнера включает в себя необходимые компоненты, такие как JVM, зависимости операционной системы, библиотеки и код приложения.
- Управление зависимостями
  - При контейнеризации зависимости приложения обычно управляются **отдельно** от самого приложения. Контейнеры используют реестры контейнеров (например, Docker Hub) для извлечения и включения необходимых зависимостей в процессе создания образа.
- Развертывание
  - Контейнерные приложения Java можно развернуть, запустив образ контейнера в среде выполнения контейнера, такой как Docker или Kubernetes. Среда выполнения контейнера выделяет необходимые ресурсы и выполняет приложение в изолированной среде контейнера.

Virtual Machines vs Containers

- Уровень изоляции
  - **VM** обеспечивают более надежную изоляцию между приложениями, поскольку каждая виртуальная машина имеет собственный экземпляр операционной системы.
  - **Контейнеры**, с другой стороны, совместно используют ядро операционной системы хоста, обеспечивая более легкую изоляцию, но большую эффективность использования ресурсов.
- Портативность
  - **Контейнеры** обеспечивают большую портативность
  - **VM** могут потребоваться дополнительные шаги для обеспечения портативности
- Расходы ресурсов
  - **VM** обычно тратят больше ресурсов из-за необходимости в отдельном экземпляре операционной системы
  - **Контейнеры** тратят меньше ресурсов, что делает их более эффективными с точки зрения использования ресурсов.
- Гибкость, лёгкость развертывания
  - **Контейнеры** обладают высокой гибкостью для развертывания и масштабирования приложений в различных средах, включая локальную разработку, тестирование и производство
  - **VM** часто используются в виртуализированной инфраструктуре и облачных средах

Как же выбрать?

- Выбор между VM или контейнерами зависит от факторов:
  - требования к развертыванию приложения
  - инфраструктура
  - потребности в масштабируемости
  - желаемый уровень изоляции для приложения

Docker, dockerfile

- dockerfile
  - файл с инструкциями о том, как построить docker-образ приложения. Может лежать в корне приложения.

```
# Use a base Java image
FROM openjdk:19

# Set the working directory inside the container
WORKDIR /your-project

# Copy the compiled Java application to the container
COPY src ./src

# Set the entry point for the container
ENTRYPOINT ["java", "/your/full/path/to/Main.java"]
```

dockerfile

```
# Use a base Java image
FROM openjdk:19

# Set the working directory inside the container
WORKDIR /your-project

# Copy the compiled Java application to the container
COPY src ./src

# Set the entry point for the container
ENTRYPOINT ["java", "/your/full/path/to/Main.java"]
```

dockerfile

- Docker предоставляет подробные данные в процессе сборки, отображая выполняемые шаги, загружая необходимые зависимости и регистрируя любые ошибки или предупреждения во время сборки:

```
docker build -t your-project-tag .
Sending build context to Docker daemon  18.43kB
Step 1/4 : FROM openjdk:19
 ---> 2e6f6690e479
Step 2/4 : WORKDIR /your-project
 ---> Running in d0438f3e3e1c
Removing intermediate container d0438f3e3e1c
 ---> 482a29e1a9fb
Step 3/4 : COPY src ./src
 ---> b49b8e41db7a
Step 4/4 : CMD ["java", "/your/full/path/to/Main.java"]
 ---> Running in 379b3050f551
Removing intermediate container 379b3050f551
 ---> 9f9e2576c283
Successfully built 9f9e2576c283
Successfully tagged your-project-tag:latest
```

docker build command

- docker build [OPTIONS] PATH | URL | -
  - Строит docker-образ из dockerfile
  - Основная команда в workflow docker
  - Позволяет создать кастомные образы, инкапсулирующее приложение и его зависимости
  - Docker читает Dockerfile и выполняет каждую инструкцию по порядку, создавая промежуточные контейнеры для каждой инструкции. Эти промежуточные контейнеры образуют слои, которые кэшируются для ускорения последующих сборок.

`docker build` command

- Ключевые опции
  - **`-t, --tag`** - Assigns a name and optionally a tag to the image being built.
  - **`-f, --file`** - Specifies the name of the Dockerfile (default: Dockerfile).
  - **`--build-arg`** - Sets build-time variables to be used in the Dockerfile.
  - **`--no-cache`** - Forces the build process to be performed without using cached intermediate images.
  - **`--network`** - Specifies the network mode for the build process.
  - **`--pull`** - Always attempts to pull the latest version of the base image before building.
  - **`--target`** - Sets the target build stage in the Dockerfile.

docker run command

- docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
- Создаёт и запускает новый контейнер на основе docker-образа
- Позволяет запускать контейнер и взаимодействовать с ним, задавая различные параметры и конфигурации
- Контейнер запускает указанную команду (или команду по умолчанию) и выполняет любые действия, определенные внутри

Your image distribution

- Docker Hub
- Private registry
- Export and import – сохранит в текущую директорию `.tar`-файл
  - `docker save -o your-image.tar your-current-image`
  - `docker load [OPTIONS] – чтобы загрузить образ приложения`

```
# Base image
FROM openjdk:11

# Set the working directory inside the container
WORKDIR /app

# Copy the application JAR file to the container
COPY myapp.jar /app/myapp.jar

# Install any necessary libraries or dependencies
RUN apt-get update && \
    apt-get install -y <library1> <library2> <library3>

# Set environment variables, if needed
ENV MY_ENV_VARIABLE=value

# Expose any necessary ports
EXPOSE 8080

# Define the command to run the application
CMD ["java", "-jar", "myapp.jar"]
```