Docker Cheat Sheet Spring Boot

Create an entire cheat sheet explaining all the concepts of docker in simple but detail.
Also explain about topics that can be asked in interview.
Basically in context of spring boot app with docker.
Add question that can be asked.

Docker Cheat Sheet for Spring Boot Developers

Docker is a platform that enables developers to package applications into containers, ensuring they run smoothly across different computing environments. In the context of Spring Boot applications, Docker is particularly useful for packaging, deployment, and scaling microservices.

1. What is Docker?

- Docker is a containerization tool used to run applications in isolated environments called containers.
- A container bundles the application code with dependencies, libraries, and configurations, making it portable and consistent across various environments.

2. Key Docker Concepts

- **Image:** A read-only template that defines the container. For example, a Spring Boot application image contains the app's code and its runtime environment (JVM, OS, etc.).
- **Container:** A runnable instance of an image. You can start, stop, and restart containers.
- **Dockerfile:** A text file that contains instructions to build a Docker image. It includes the base image, application source code, and necessary dependencies.
- **Docker Hub:** A cloud repository for sharing and distributing Docker images.
- **Volume:** A mechanism for persisting data generated by containers so that it is not lost when containers stop or are removed.
- **Network:** Allows containers to communicate with each other. By default, Docker containers are isolated but can be connected through networks.

3. Dockerfile Structure for Spring Boot

A `Dockerfile` is used to build a Docker image for a Spring Boot app.

Dockerfile

```
# 1. Base image (a minimal OS image with Java)
FROM openjdk:17-jdk-alpine

# 2. Set working directory in the container
WORKDIR /app

# 3. Copy JAR file from local to the container
```

```
COPY target/myapp.jar /app/myapp.jar

# 4. Expose the port Spring Boot runs on
EXPOSE 8080

# 5. Command to run the application
ENTRYPOINT ["java", "-jar", "myapp.jar"]
```

Commands Explanation:

- `**FROM**`: Specifies the base image (Java 17 in this case).
- `**WORKDIR**`: Defines the working directory inside the container.
- `**COPY**`: Copies the app's compiled JAR file into the container.
- `**EXPOSE**`: Opens port 8080 for the app to listen on.
- `**ENTRYPOINT**`: Defines the default command to run the Spring Boot app inside the container.

4. Building and Running the Docker Image

- **Build the Docker image:**

  ```bash
  docker build -t my-spring-boot-app .
  ```

- **Run the Docker container:**

  ```bash
  docker run -p 8080:8080 my-spring-boot-app
  ```

Explanation:

- `**-t my-spring-boot-app**`: Tags the image with a name.
- `**-p 8080:8080**`: Maps port 8080 on your machine to port 8080 in the container.

5. Docker Compose

`**docker-compose**` simplifies multi-container setups. For instance, a Spring Boot app with a PostgreSQL database can be defined in a `**docker-compose.yml**` file.

```yaml
version: '3'
services:
  app:
    image: my-spring-boot-app
    ports:
      - "8080:8080"
    depends_on:
      - db
  db:
    image: postgres:13-alpine
    environment:
      POSTGRES_USER: user
```

```
      POSTGRES_PASSWORD: password
      POSTGRES_DB: mydb
    ports:
      - "5432:5432"
```

Commands to Run:

- Start services:

```bash
docker-compose up
```

- Stop services:

```bash
docker-compose down
```

6. Persisting Data with Volumes

In production, data should persist even when containers are stopped. Docker volumes allow for this.

Example of adding a volume for the PostgreSQL container in `docker-compose.yml`:

```yaml
volumes:
  db_data:
services:
  db:
    image: postgres:13-alpine
    volumes:
      - db_data:/var/lib/postgresql/data
```

7. Docker Networks

Containers can communicate over Docker networks. By default, containers within the same network can communicate using their service names as hostnames.

In the `docker-compose.yml`, the `app` service can communicate with the `db` service by using `db` as the hostname.

8. Best Practices for Docker in Spring Boot

- **Multi-stage builds:** Use multi-stage builds in your Dockerfile to reduce the image size by separating the build and run stages.

  Example:

```
Dockerfile
```

```
FROM maven:3.8.6-jdk-17 as builder
WORKDIR /app
COPY . .
RUN mvn clean package -DskipTests

FROM openjdk:17-jdk-alpine
WORKDIR /app
COPY --from=builder /app/target/myapp.jar /app/myapp.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "myapp.jar"]
```

- **Health checks:** Ensure your application is running with health checks.

```
yaml
```

```
app:
  image: my-spring-boot-app
  ports:
    - "8080:8080"
  healthcheck:
    test: ["CMD", "curl", "-f", "http://localhost:8080/actuator/health"]
    interval: 30s
    timeout: 10s
    retries: 3
```

9. Docker Interview Questions for Spring Boot Developers

1. **What is Docker, and how does it differ from virtual machines?**
   - Docker is lightweight compared to VMs as it shares the host OS kernel and only requires the application and its dependencies.
2. **Explain the difference between Docker Image and Docker Container.**
   - A Docker image is a template, while a container is a running instance of that image.
3. **How do you build and run a Docker image for a Spring Boot application?**
   - Build using `` `docker build -t <image-name> .` `` and run using `` `docker run -p <host-port>:<container-port> <image-name>` ``.
4. **What is a Dockerfile?**
   - A Dockerfile is a script that contains instructions to build a Docker image.
5. **How do you use Docker Compose with a Spring Boot app and a database?**
   - Define services (app and db) in a `` `docker-compose.yml` `` and run `` `docker-compose up` `` to bring up both containers.
6. **How do you persist data in Docker containers?**
   - Use volumes in Docker, which allows data to be saved even if the container is removed or stopped.

7. **How can you reduce the size of a Docker image?**
    - Use multi-stage builds and Alpine base images to reduce image size.
8. **What are some best practices for running a Spring Boot application in Docker?**
    - Use a lightweight base image, multi-stage builds, and health checks for monitoring the container's status.
9. **What are Docker volumes, and why are they important?**
    - Volumes are used to persist data outside the lifecycle of a container. They're essential for databases or any stateful application.
10. **How do you troubleshoot issues with Docker containers?**
    - Use `` `docker logs <container_id>` `` to view logs, `` `docker exec` `` to run commands inside the container, and `` `docker inspect` `` to check container details.

10. Common Docker Commands

- **List running containers:** `` `docker ps` ``
- **Stop a container:** `` `docker stop <container_id>` ``
- **Remove a container:** `` `docker rm <container_id>` ``
- **Remove an image:** `` `docker rmi <image_name>` ``
- **View logs of a container:** `` `docker logs <container_id>` ``
- **Run a command inside a container:** `` `docker exec -it <container_id> <command>` ``

By understanding Docker and how to containerize Spring Boot applications, you'll be well-prepared for both practical usage and interview questions regarding Docker in Spring Boot contexts.

Explain the instructions written in docker file in detail and explain some others as well

Detailed Explanation of Dockerfile Instructions

A `` `Dockerfile` `` is a script containing a series of instructions to build a Docker image. Each instruction in the Dockerfile is executed in sequence to create a final image that can be used to run containers. Let's dive into the most common instructions and a few others that are often used.

1. `` `FROM` ``

- **Description:** This instruction sets the base image for your Docker image. The base image is a minimal OS or runtime that your application depends on.
- **Syntax:**

```
Dockerfile

FROM <image>:<tag>
```

Example:

```
Dockerfile

FROM openjdk:17-jdk-alpine
```

- `openjdk`: The name of the base image, in this case, an image that includes Java Development Kit (JDK).
- `17-jdk-alpine`: The specific version tag of the image, in this case, JDK 17 on Alpine Linux (a lightweight Linux distribution).

Other common examples:

- `FROM ubuntu:20.04`: Ubuntu base image with version 20.04.
- `FROM node:16`: Node.js base image with version 16.

2. `WORKDIR`

- **Description:** Sets the working directory inside the container. It's the directory where subsequent commands like `COPY`, `RUN`, or `CMD` will be executed.
- **Syntax:**

```
Dockerfile

WORKDIR /path/to/directory
```

Example:

```
Dockerfile

WORKDIR /app
```

This command creates the `/app` directory if it doesn't already exist and sets it as the working directory for the container.

3. `COPY`

- **Description:** Copies files from your local file system (host) to the Docker container's file system.
- **Syntax:**

```
Dockerfile

  COPY <source> <destination>
```

Example:

```
Dockerfile

  COPY target/myapp.jar /app/myapp.jar
```

- `**target/myapp.jar**`: Refers to the JAR file located on your host machine.
- `**/app/myapp.jar**`: Refers to the destination path inside the container where the file is copied.

4. `**RUN**`

- **Description:** Executes commands in the container during the image build process. This is typically used to install dependencies or perform system configurations.
- **Syntax:**

```
Dockerfile

  RUN <command>
```

Example:

```
Dockerfile

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

This command updates the package lists and installs `**curl**` inside the container.

Other common examples:

- `**RUN mvn clean package**`: In a Maven-based Java project, this command would build the project during image creation.

5. `**CMD**`

- **Description:** Specifies the default command to run when a container is started. You can only have one `**CMD**` per Dockerfile, and it's often used to define the main process of the container.
- **Syntax:**

```
Dockerfile

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

Example:

```
Dockerfile

CMD ["java", "-jar", "myapp.jar"]
```

- This command runs the Spring Boot application (`**myapp.jar**`) inside the container.

Note: If `**CMD**` is overridden when running the container (for example, `**docker run myapp echo "Hello"**`), the default command is replaced.

6. `**ENTRYPOINT**`

- **Description:** Similar to `**CMD**`, but it configures a container to run as an executable. It is often used to define a command that can't be overridden, with parameters passed to the container as additional arguments.
- **Syntax:**

```
Dockerfile

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

Example:

```
Dockerfile

ENTRYPOINT ["java", "-jar", "myapp.jar"]
```

Difference between `CMD` and `ENTRYPOINT`:

- `**ENTRYPOINT**` is primarily used for containers that act like executables, where additional parameters can be appended.
- `**CMD**` can be easily overridden when starting a container, while `**ENTRYPOINT**` is more rigid.

7. `**EXPOSE**`

- **Description:** This instruction informs Docker that the container will listen on a specified network port at runtime. It doesn't actually publish the port, but serves as documentation for users who run the container.
- **Syntax:**

```Dockerfile
EXPOSE <port>
```

Example:

```Dockerfile
EXPOSE 8080
```

- Exposes port 8080, which is typically used by a Spring Boot app. To make it accessible from the host machine, you must publish it using `**docker run -p 8080:8080**`.

8. `**ENV**`

- **Description:** Sets environment variables within the container.
- **Syntax:**

```Dockerfile
ENV <key>=<value>
```

Example:

```Dockerfile
ENV SPRING_PROFILES_ACTIVE=prod
```

This sets the `**SPRING_PROFILES_ACTIVE**` environment variable to `**prod**`, which can be used to specify the active profile in a Spring Boot application.

9. `**VOLUME**`

- **Description:** Creates a mount point with a volume, enabling persistent storage between container runs.
- **Syntax:**

```Dockerfile
VOLUME ["/path/in/container"]
```

Example:

```
Dockerfile

  VOLUME ["/var/lib/postgresql/data"]
```

- This instruction defines a volume for a PostgreSQL container to store its data, making it persistent across container restarts.

10. `ADD`

- **Description:** Similar to `COPY`, but with additional features. It can handle tar files (automatically extracting them) and can download files from URLs.
- **Syntax:**

```
Dockerfile

  ADD <source> <destination>
```

Example:

```
Dockerfile

  ADD https://example.com/file.zip /app/file.zip
```

- This downloads `file.zip` from the URL and adds it to the `/app` directory inside the container.

Note: Prefer `COPY` over `ADD` unless you specifically need the extra features.

11. `ARG`

- **Description:** Defines a build-time variable that can be passed to Docker during the build process.
- **Syntax:**

```
Dockerfile

  ARG <variable_name>
```

Example:

```
Dockerfile

  ARG JAR_FILE=target/myapp.jar
  COPY ${JAR_FILE} /app/myapp.jar
```

- This allows you to build with a specific JAR file by passing an argument when building:

```bash
docker build --build-arg JAR_FILE=target/myapp-other.jar -t myapp .
```

12. `USER`

- **Description:** Specifies the user to run commands as inside the container.
- **Syntax:**

```Dockerfile
USER <username>
```

Example:

```Dockerfile
USER root
```

- This ensures that all subsequent commands in the Dockerfile are run as the `root` user.

13. `HEALTHCHECK`

- **Description:** Tells Docker how to check if your container is healthy by running a command. If the command exits with a 0 status, the container is healthy; otherwise, it is unhealthy.
- **Syntax:**

```Dockerfile
HEALTHCHECK [OPTIONS] CMD <command>
```

Example:

```Dockerfile
HEALTHCHECK CMD curl --fail http://localhost:8080/actuator/health || exit 1
```

- This will check the health of a Spring Boot app by calling its health endpoint (`/actuator/health`).

14. `LABEL`

- **Description:** Adds metadata to an image, such as version, description, or maintainer info.
- **Syntax:**

```
Dockerfile

  LABEL <key>=<value>
```

Example:

```
Dockerfile

  LABEL version="1.0"
  LABEL description="My Spring Boot Application"
```

Other Important Docker Concepts

- **Multi-Stage Builds:** This allows you to create more efficient Docker images by separating the build environment from the runtime environment, reducing image size.

  Example:

```
Dockerfile

  # Build stage
  FROM maven:3.8.6-jdk-17 AS builder
  WORKDIR /app
  COPY . .
  RUN mvn clean package -DskipTests

  # Runtime stage
  FROM openjdk:17-jdk-alpine
  WORKDIR /app
  COPY --from=builder /app/target/myapp.jar /app/myapp.jar
  ENTRYPOINT ["java", "-jar", "myapp.jar"]
```

  - This approach reduces the final image size by copying only the compiled JAR file from the build stage to the runtime stage.

Additional Interview Questions on Docker

1. **What is the difference between `COPY` and `ADD` in Docker?**
   - `COPY` is used to copy files from the host to the container. `ADD` can do the same but also allows downloading from URLs and extracting tar files.
2. **Explain the use of multi-stage builds in Docker.**
   - Multi-stage builds allow you to separate build and runtime environments, reducing the size of the final image by only including the necessary runtime artifacts.
3. **What is the purpose

ChatGPT can make mistakes. Check important info.