NOTE: 1. To open git in eclipse right click on project -> show in local terminal -> Git bash

2. You can type docker commands here only in this open git bash in eclipse. Give docker version to see client and server is displayed.

3. Note first open docker desktop and docker hub

4. To run web project, see Server is started status:

to start server in eclipse do :Right click on server tab-> start

Note everything from eclipse:

* To run Maven web project clean,install,test, build(goal =package), Run on server.
* Open Git bash in eclipse to run git commands
* Docker commands and docker login in opened git bash only
* Dockerfile and docker-compse.yml files create in maven project floder(in eclipse) opened from given link
* Dockerfile and docker-compse.yml files should not be under src or target. Create them above pom.xml

**Q2.** Clone the Train Booking System repository to your eclipse software from the URL below and execute the project by changing the required changes in the POM.xml file and make it a successful build.  **-30 Marks**

URL: eg. <https://github.com/archanareddyse/labinternal-1.git>

1. Resolve dependencies using pom.xml
2. Build the project to generate the WAR/JAR file
3. Verify the generated artifact in the target/ folder
4. **Open Eclipse**

**2. To Clone Repository do**

**Method I**

Go to the menu:  
 **File → Import → Git → Projects from Git (with smart import)** → Next.

Choose **Clone URI** → Next.

Enter the **GitHub repository URL** (SSH or HTTPS):

* + Put this URL (HTTPS):

<https://github.com/archanareddyse/labinternal-1.git>

Select the **branches** you want (usually main or master) → Next.

Choose the **directory** where Eclipse will store the repo → Finish

Or **Method II**

In git bash Clone URL <https://github.com/archanareddyse/labinternal-1.git>

Open the cloned project folder in eclipse(File 🡪 Open Projects from File system -> Import source specify path

POM.xml file read

The XML snippet you provided is a \*\*Maven Project Object Model (POM)\*\* file.

Maven is a build automation tool used primarily for Java projects, and the POM file is essential for defining project configuration, dependencies, build process, and more. Let's go through the key components and their meanings in this POM file:

### Breakdown of the POM File

1. \*\*XML Declaration:\*\*

```xml

<?xml version="1.0" encoding="UTF-8"?>

```

- This line defines the XML version and the character encoding used in the document (UTF-8).

2. \*\*Project Element:\*\*

```xml

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

```

- The `<project>` tag is the root element for the POM file.

- It includes namespaces to define the schema version of the POM.

3. \*\*Model Version:\*\*

```xml

<modelVersion>4.0.0</modelVersion>

```

- This specifies the version of the POM model being used. For Maven 4.0.0 and higher, it generally remains fixed.

4. \*\*Group ID:\*\*

```xml

<groupId>SEGroup</groupId>

```

- Defines the group or organization that the project belongs to. It's used in the unique identification of artifacts.

5. \*\*Artifact ID:\*\*

```xml

<artifactId> ProjectNameYouGave </artifactId>

```

- This is the name of the project. It's a unique identifier for the artifact within the group.

6. \*\*Version:\*\*

```xml

<version>0.0.1-SNAPSHOT</version>

```

- Indicates the current version of the project. `SNAPSHOT` indicates that it is still in development and may change.

7. \*\*Packaging:\*\*

```xml

<packaging>war</packaging>

```

- Specifies the type of artifact to be built. In this case, it's a WAR (Web Application Archive) file suitable for web applications.

8. \*\*Project Name and URL:\*\*

```xml

<name> ProjectNameYouGave Maven Webapp</name>

<url>http://www.example.com</url>

```

- The `<name>` element provides a human-readable name for the project.

- The `<url>` element indicates the project's website (to be updated appropriately).

9. \*\*Properties:\*\*

```xml

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<maven.compiler.source>8</maven.compiler.source>

<maven.compiler.target>8</maven.compiler.target>

</properties>

```

- This section defines properties that can be reused throughout the POM.

- `project.build.sourceEncoding` sets the source file encoding.

- `maven.compiler.source` and `maven.compiler.target` set the Java version used for compiling the code to version 8.

10. \*\*Dependencies:\*\*

```xml

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13.1</version>

<scope>test</scope>

</dependency>

</dependencies>

```

- This section lists dependencies required by the project.

- Here, JUnit is included as a dependency for testing purposes only (denoted by `<scope>test</scope>`), meaning it won't be included in the final WAR file.

11. \*\*Build Section:\*\*

```xml

<build>

<finalName> ProjectNameYouGave</finalName>

```

- The `<build>` section contains settings related to building the project.

- `finalName` specifies the name of the artifact generated (WAR file).

12. \*\*Plugin Management:\*\*

```xml

<pluginManagement>

<plugins>

<plugin>

<artifactId>maven-clean-plugin</artifactId>

<version>3.4.0</version>

</plugin>

...

</plugins>

</pluginManagement>

```

- This section allows for the management of Maven plugins and their versions to ensure consistent builds.

- Several plugins are listed:

- `maven-clean-plugin`: Cleans the target directory.

- `maven-resources-plugin`: Handles copying resources.

- `maven-compiler-plugin`: Compiles the project source code.

- `maven-surefire-plugin`: Runs the unit tests.

- `maven-war-plugin`: Packages the web application into a WAR file.

- `maven-install-plugin`: Installs the package into the local repository.

- `maven-deploy-plugin`: Deploys the built artifacts to a remote repository.

### Summary

This POM file acts as the central configuration file for a Maven project. It outlines how the project is structured, what dependencies it has, how it should be built, and what plugins are used. By properly configuring the POM file, developers can manage their Java projects effectively, ensuring consistency in building, testing, and deploying applications. If you have specific questions about any parts of this POM or would like to know more, feel free to ask!

DOCKER

NOTE: To open git in eclipse right click on project -> show in local terminal -> Git bash

You can type docker commands here only in this open git bash in eclipse. Give docker version to see client and server is displayed.

Note first open docker desktop and docker hub

Code to create Docker image for Maven web project of given link in Q2.

\*\*Dockerfile\*\* that describes how to package a Java web application as a Docker image using Apache Tomcat as the servlet container.

# Use Tomcat 9 as base image

FROM tomcat:9.0

# Copy your built WAR into Tomcat’s webapps folder

COPY target/\*.war /usr/local/tomcat/webapps/ROOT.war

# Start Tomcat

CMD ["catalina.sh", "run"]

Below, I’ll explain each line of the Dockerfile in detail.

### Breakdown of the Dockerfile

1. \*\*Base Image Declaration:\*\*

```dockerfile

FROM tomcat:9.0

```

- \*\*FROM\*\*: This instruction sets the base image for the Docker image being built. Here, it specifies that Apache Tomcat version 9.0 should be used as the base image.

- By using an official Tomcat image, you start with a pre-configured environment that includes Tomcat server, Java, and other necessary components to run Java web applications.

2. \*\*Copying the WAR file:\*\*

```dockerfile

COPY target/\*.war /usr/local/tomcat/webapps/ROOT.war

```

- \*\*COPY\*\*: This instruction copies files or directories from the host machine into the Docker image being built.

- `target/\*.war`: This specifies the source location of the WAR file that has been built by your Java project. This is usually the output of a build process (like Maven or Gradle) that compiles the code and packages it into a WAR (Web Application Archive) file.

- `/usr/local/tomcat/webapps/ROOT.war`: This specifies the destination path within the Docker image. Tomcat uses this directory (`/usr/local/tomcat/webapps`) to deploy web applications. By copying the WAR file to this location with the name `ROOT.war`, you configure Tomcat to serve this application as the default web application, accessible at the root context (e.g., `http://localhost:8080/`).

3. \*\*Starting Tomcat:\*\*

```dockerfile

CMD ["catalina.sh", "run"]

```

- \*\*CMD\*\*: This instruction specifies the command that should be executed when a container based on the image starts. It can be overridden by providing a different command when running the container.

- `["catalina.sh", "run"]`: This is the default command to start the Tomcat server. `catalina.sh` is the script that controls Tomcat server startup processes, and the `run` argument tells it to start the server in the foreground. Running Tomcat in the foreground is essential in Docker containers because it allows Docker to manage the process properly. If the main process in a container exits, the container will stop.

### Summary

This Dockerfile is designed to create a container image for a Java web application that is deployed on an Apache Tomcat server.

- It uses the official Tomcat 9.0 image as the base image.

- It copies a previously built WAR file from the host machine into the webapps directory of Tomcat to make it accessible as a web application.

- Finally, it specifies the command to start the Tomcat server when a container is launched from this image.

This setup allows developers to easily deploy and run a Java web application in a consistent environment using Docker. If you have any further questions or need clarification on any part of this Dockerfile, feel free to ask!

**Q Build custom Docker image**

**Task:** Containerize your Maven project using Docker.

1. **Dockerfile Creation** 
   * Ensure it copies the WAR/JAR and runs on Tomcat (or relevant base image).

**Dockerfile**

# Use Tomcat 9 as base image

FROM tomcat:9.0

# Copy your built WAR into Tomcat’s webapps folder

COPY target/\*.war /usr/local/tomcat/webapps/ROOT.war

# Start Tomcat

CMD ["catalina.sh", "run"]

1. **Image Building** 
   * Build the Docker image using docker build -t <image\_name> .
   * Commands docker build –t yogflightimage **.**
   * docker images
   * docker run --name yogflightcontainer –d yogflightimage
   * docker ps
   * docker stop yogflightcontainer
   * docker ps
2. **Push to Docker Hub** 
   * Tag and push the created image to your Docker Hub account.
   * Commands
   * docker login
   * tag image as
   * docker commit <containername / id> <docker hub name>**/**<new image name>

example-> docker commit yogflightcontainer shradha2022/flightimageyog

* + docker images // shows image flightimageyog
  + docker push shradha2022/flightimageyog // pushes image in docker hub
  + docker rm yogflightcontainer
  + docker rmi shradha2022/flightimageyog

archanas ->docker push shradha2022/libimagearchana ---in hub

1. SBQ

**Q Docker Compose for a multi-container setup**

Write a **docker-compose.yml** file for a multi-container setup.

1. **Container 1** – Use the Docker image you pushed to Docker Hub i.e. flightimageyog
2. **Container 2** – Configure a database container (MySQL/PostgreSQL/MongoDBEnsure both containers are running together

Code here docker-compose.yaml file

version: "3.9"

services:

app:

build: . # build from local Dockerfile

image: flightimageyog # libimagearchana

container\_name: tomcat-app

ports:

- "8080:8080"

depends\_on:

- db

environment:

DB\_HOST: db

DB\_PORT: 3306

DB\_NAME: mydb

DB\_USER: myuser

DB\_PASS: mypassword

db:

image: mysql:8.0

container\_name: mysql-db

restart: always

environment:

MYSQL\_DATABASE: mydb

MYSQL\_USER: myuser

MYSQL\_PASSWORD: mypassword

MYSQL\_ROOT\_PASSWORD: rootpassword

ports:

- "3307:3306"

volumes:

- db\_data:/var/lib/mysql

volumes:

db\_data:

1. Demonstrate service startup with docker-compose up

Shows => [+] Running 4/4

✔ Network labinternal-1\_default Created 0.2s

✔ Volume "labinternal-1\_db\_data" Created 0.0s

✔ Container mysql-db Started 2.3s

✔ Container tomcat-app Started 2.1s

Next stop running container by command:

docker-compose down