WEEK -2

**Step 3: Prepare Your Project**

* Have a **multi-folder project** (e.g., a small website or code files in folders).
* Place it in one folder, e.g., C:\Users\YourName\MyProject.

**🔹 Step 4: Create a Repository on GitHub**

1. Click **New Repository** in GitHub.
2. Give it a name (MyProject).
3. Choose:
   * **Public** → anyone can see (for Task 1, teacher demo).
   * **Private** → only you can see (for Task 2, student task).
4. Don’t add README yet (since we’ll push from local).

**Step 5: Initialize Git in Your Project (Local)**

1. Open **Git Bash** inside your project folder:
2. cd /c/Users/YourName/MyProject
3. git init
4. This makes it a Git repository.

**Step 6: Configure Git (only once)**

1. Set your name and email (appears in commits):
2. git config --global user.name "Your Name"
3. git config --global user.email "your@email.com"

**🔹 Step 7: Connect Local Project to GitHub Repo**

After creating the repo, GitHub gives you a link like:

https://github.com/username/MyProject.git

Add it in Git Bash:

git remote add origin https://github.com/username/MyProject.git

Check if added:

git remote -v

**🔹 Step 8: Add & Commit Files**

git add .

git commit -m "Initial commit"

**🔹 Step 9: Push to GitHub**

If GitHub default branch is **main** (most common):

git push -u origin main

If it’s **master**:

git push -u origin master

Now your project is online 🎉

**🔹 Step 10: Practice Basic Git Commands**

You need to practice all the commands in your doc (like git status, git branch, git log, git diff, etc.) — try them inside your project.  
Example:

git status

git log --oneline

git branch -a

**🔹 Step 11: Solve Scenario-Based Questions**

For example:

* **Q1:** You made a mistake before staging →
* git restore filename
* **Q2:** You staged by mistake →
* git reset filename
* **Q3:** Wrong commit message →

git commit --amend -m "Corrected message"

Week 3

1. Create a **GitHub Organization** for team collaboration.
2. Work together on a **shared repository** (invite collaborators).
3. Learn **forking and pull requests** (for contributing to others’ repos).
4. Handle **merge conflicts** when two people edit the same file.
5. Create and apply a **patch file**.
6. Solve scenario-based Git problems.

**🚀 Step 1: Create an Organization on GitHub**

1. Go to **GitHub → Profile menu → Your organizations → New organization**
2. Click **Create a free organization**.
3. Enter:
   * Organization name
   * Email address
   * Verification
   * Click **Next**
4. Add members (your classmates/friends). They’ll receive invites.
5. In **Settings → Member privileges**, set **Base permissions → Write** (so they can push code).
6. Inside the organization, **create a repository** (can be private).

👉 Now your repo belongs to the organization, and everyone added can work together.

**🚀 Step 2: Collaborator Setup (Shared Repo)**

* **Collaborator 1 (owner):**
  + Go to repo → **Settings → Collaborators → Add people** → Invite.
* **Collaborator 2 (you, student):**
  + Accept invite.
  + Clone repo to local machine:
  + git clone https://github.com/orgname/repo-name.git
  + cd repo-name

**🚀 Step 3: Work on Branches**

Always work in **separate branches** (not directly in main):

git checkout -b feature/my-feature

* Make changes →
* git add .
* git commit -m "Added new feature"
* git push origin feature/my-feature
* Owner (or teammates) can merge your branch into main.

**🚀 Step 4: Forking & Pull Requests (when repo is not yours)**

* Go to someone else’s repo → click **Fork**.
* GitHub copies it into your account.
* Clone fork:
* git clone https://github.com/yourname/forked-repo.git
* Make changes → commit → push → open **Pull Request** back to the original repo.
* Repo owner reviews and merges.

**🚀 Step 5: Handling Merge Conflicts**

Conflicts happen when **two people change the same line**.

1. Pull changes:
2. git pull origin main
3. Git marks conflict inside file:
4. <<<<<<< HEAD
5. Your changes
6. =======
7. Other collaborator’s changes
8. >>>>>>> branch-name
9. Edit file → keep one version or merge both → remove markers.
10. Mark as resolved:
11. git add filename
12. git commit
13. Push resolved changes.

**🚀 Step 6: Creating & Applying a Patch**

A patch = file containing changes.

* Create patch (from one commit):
* git format-patch -1 <commit-hash>

→ generates 0001-commit-message.patch

* Send patch file to teammate.
* Apply patch:
* git apply 0001-commit-message.patch
* # or
* git am 0001-commit-message.patch

**🚀 Step 7: Scenario Questions (examples)**

1. Error: *“rejected – non-fast-forward”* → Run:
2. git pull origin main --rebase
3. git push origin main
4. Push feature branch without affecting main:
5. git push origin feature/my-branch
6. Keep local repo updated:
7. git fetch origin
8. git pull origin main
9. Delete remote branch:
10. git push origin --delete feature/test

**Week 4: Maven Java + Web Project + Push to GitHub**

**📌 Part 1: Setup Before Starting**

1. **Install Eclipse IDE for Enterprise Java & Web**
   * Download: https://www.eclipse.org/downloads/
   * Choose *Eclipse IDE for Enterprise Java and Web Developers*.
2. **Install Apache Tomcat (for Web Project)**
   * Download: https://tomcat.apache.org/download-90.cgi
   * Extract to a folder (e.g., C:\apache-tomcat-9).
3. **Install Git** (if not done in Week 2 & 3)
   * Download: <https://git-scm.com/downloads>
   * Verify with:
   * git --version

**📌 Part 2: Create Maven Java Project (Console App)**

1. Open **Eclipse** → File → New → Maven Project.
2. In “Filter”, type quickstart.
3. Select **maven-archetype-quickstart** (version 1.4).
4. Fill details:
   * **Group ID (TeamID):** SE
   * **Artifact ID:** MavenJava  
     → Finish.

👉 Eclipse will generate a sample Java project with pom.xml.

1. Open file:  
   src/main/java/SE/MavenJava/App.java  
   → It has **Hello World** code.
2. Run build steps:
   * Right-click App.java → **Run As → Maven Clean**
   * Right-click App.java → **Run As → Maven Install**
   * Right-click App.java → **Run As → Maven Test**
   * Right-click App.java → **Run As → Java Application**

👉 Output should show **Hello World** ✅

**📌 Part 3: Create Maven Web Project**

1. File → New → Maven Project
2. In “Filter”, type webapp.
3. Select **maven-archetype-webapp** (version 1.4).
4. Fill details:
   * **Group ID (TeamID):** SE
   * **Artifact ID:** MavenWeb  
     → Finish.

👉 Eclipse generates a **web project** with pom.xml.

1. Open file:  
   src/main/webapp/index.jsp → contains sample **Hello World JSP**.
2. Add **Servlet API dependency** to pom.xml:
   * Go to <https://mvnrepository.com>
   * Search: *Servlet API* → copy latest dependency code.
   * Paste inside <dependencies> in pom.xml.
3. Setup **Tomcat Server** in Eclipse:
   * Window → Show View → Servers
   * Add **Apache Tomcat v9.0**
   * Configure ports (e.g., HTTP → 8085).
   * Add user in tomcat-users.xml file:
   * <role rolename="manager-gui,admin-gui,manager-script"/>
   * <user username="admin" password="1234" roles="manager-gui,admin-gui,manager-script"/>
4. Run build steps (like Java project):
   * **Maven Clean**
   * **Maven Install**
   * **Maven Test**
   * **Maven Build** (goals: clean install test)
5. Run on server:
   * Right-click index.jsp → **Run As → Run on Server**
   * Choose Tomcat v9.0
   * Open browser → http://localhost:8085/MavenWeb/

👉 Output: **Hello World Webpage** ✅

**📌 Part 4: Push Both Projects to GitHub**

**Java Project**

1. Create **new repo on GitHub** → name it MavenJava.
2. In Eclipse: Right-click MavenJava → **Show in Local Terminal → Git Bash**.
3. Run:
4. git init
5. git branch -M main
6. git remote add origin https://github.com/yourusername/MavenJava.git
7. git add .
8. git commit -m "Maven Java push"
9. git push -u origin main
10. Refresh GitHub → Project uploaded 🎉

**Web Project**

1. Create **new repo on GitHub** → name it MavenWeb.
2. In Eclipse: Right-click MavenWeb → **Show in Local Terminal → Git Bash**.
3. Run:
4. git init
5. git branch -M main
6. git remote add origin https://github.com/yourusername/MavenWeb.git
7. git add .
8. git commit -m "Maven Web push"
9. git push -u origin main
10. Refresh GitHub → Project uploaded 🎉

WEEK5

**🔹 Part 1: Working with Redis (Docker CLI)**

**Step 1 – Pull Redis image**

docker pull redis

**Step 2 – Run a Redis container**

docker run --name my-redis -d redis

**Step 3 – Check running containers**

docker ps

**Step 4 – Access Redis CLI**

docker exec -it my-redis redis-cli

Inside Redis:

SET name "Alice"

GET name

exit

**Step 5 – Stop the container**

docker stop my-redis

**Step 6 – Restart the container**

docker start my-redis

**Step 7 – Remove the container**

docker rm my-redis

**Step 8 – Remove the image**

docker rmi redis

**🔹 Part 2: Working with Dockerfile (Custom Redis)**

**Step 1 – Create folder**

mkdir C:\DockerProjects\Redis

cd C:\DockerProjects\Redis

**Step 2 – Create Dockerfile**

notepad Dockerfile

Paste:

FROM redis:latest

CMD ["redis-server"]

**Step 3 – Build the image**

docker build -t redisnew .

**Step 4 – Run container**

docker run --name myredisnew -d redisnew

**Step 5 – See running containers**

docker ps

**Step 6 – Stop container**

docker stop myredisnew

**🔹 Part 3: Docker Hub & Image Management**

**Login**

docker login

**Commit container → new image**

docker commit <container\_id> yourdockerhubusername/redis1

**Push to Docker Hub**

docker push yourdockerhubusername/redis1

**Pull image (on another machine)**

docker pull yourdockerhubusername/redis1

**Run pulled image**

docker run --name myredis -d yourdockerhubusername/redis1

**Access Redis**

docker exec -it myredis redis-cli

Inside Redis:

SET name "Abcdef"

GET name

exit

**🔹 Part 4: Useful Commands Recap**

* List all containers (including stopped):

docker ps -a

* List images:

docker images

* Remove container:

docker rm <container\_id>

* Remove image:

docker rmi <image\_name>

* Logout:

docker logout

WEEK\_6

**Step 1 – Create project folder**

In PowerShell:

cd C:\Users\SRIJA K

mkdir week6-docker

cd week6-docker

**Step 2 – Create Flask app**

Create folder and files:

mkdir flask-app

cd flask-app

notepad app.py

notepad requirements.txt

notepad Dockerfile

cd ..

Paste into app.py:

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route("/")

def hello():

return "Hello, Docker Compose!"

if \_\_name\_\_ == "\_\_main\_\_":

app.run(host="0.0.0.0", port=5000)

Paste into requirements.txt:

flask

Paste into Dockerfile:

FROM python:3.11-slim

WORKDIR /app

COPY requirements.txt requirements.txt

RUN pip install -r requirements.txt

COPY . .

CMD ["python", "app.py"]

**Step 3 – Create docker-compose.yml**

Back in the root folder (C:\Users\SRIJA K\week6-docker):

notepad docker-compose.yml

Paste:

services:

flask-api:

build: ./flask-app

ports:

- "5000:5000"

**Step 4 – Run with Docker Compose**

In PowerShell:

docker compose up -d --build

docker compose ps

Check in browser 👉 <http://localhost:5000>

-----------------------------------------------------------------------------------------------------------------------------------

Multi-Module Maven Project (Step by Step with Commands)

**🔹 Step 1: Create Parent Project**

**👉 Eclipse GUI Way**

1. File → New → Other → Maven → Maven Project
2. Select **Create a simple project (skip archetype selection)**
3. Fill:
   * GroupId → KMIT
   * ArtifactId → MultiModule
   * Packaging → pom
   * Name → Multimodule creation
   * Description → Sample multimodule with parent child hierarchy
4. Click **Finish**

**👉 Command Line Way**

mvn archetype:generate -DgroupId=KMIT -DartifactId=MultiModule -Dpackaging=pom -DinteractiveMode=false

This creates the **parent project** with packaging type **POM**.

**🔹 Step 2: Create First Child (Child1)**

**👉 Eclipse GUI Way**

1. Right-click on MultiModule → New → Maven Module
2. Select **Create a simple project (skip archetype selection)**
3. ArtifactId → MultiModuleChild1
4. Finish

**👉 Command Line Way**

cd MultiModule

mvn archetype:generate -DgroupId=KMIT -DartifactId=MultiModuleChild1 -DinteractiveMode=false

👉 This creates a new module (default JAR type).

**🔹 Step 3: Create Second Child (Child2 as WebApp)**

**👉 Eclipse GUI Way**

1. Right-click on MultiModule → New → Maven Module
2. ArtifactId → MultiModuleChild2
3. In filters search → maven-archetype-webapp
4. Choose version **1.1** or **1.5**
5. Finish (type **Y** if console asks)

**👉 Command Line Way**

cd MultiModule

mvn archetype:generate -DgroupId=KMIT -DartifactId=MultiModuleChild2 \

-DarchetypeArtifactId=maven-archetype-webapp -DarchetypeVersion=1.4 \

-DinteractiveMode=false

👉 This creates a **Web Application (WAR)** module.

**🔹 Step 4: Link Modules in Parent POM**

In **MultiModule/pom.xml**, add children inside <modules> tag:

<modules>

<module>MultiModuleChild1</module>

<module>MultiModuleChild2</module>

</modules>

👉 Now parent knows both child projects.

**🔹 Step 5: Add Dependency (Child2 depends on Child1)**

If **Child2 needs Child1**, open MultiModuleChild2/pom.xml and add inside <dependencies>:

<dependency>

<groupId>KMIT</groupId>

<artifactId>MultiModuleChild1</artifactId>

<version>0.0.1-SNAPSHOT</version>

</dependency>

**🔹 Step 6: Build Projects in Correct Order**

**👉 Correct Build Order**

1. Parent
2. Child1
3. Child2

**👉 Eclipse Way**

* Right-click → Run As → Maven Build

**👉 Command Line Way**

# Build parent first

cd MultiModule

mvn clean install

# Then build Child1

cd MultiModuleChild1

mvn clean install

# Finally build Child2

cd ../MultiModuleChild2

mvn clean install

✅ Now build will succeed, because dependencies are satisfied.

**📂 Final Folder Structure**

MultiModule (parent - pom.xml)

│── pom.xml (packaging = pom)

│

├── MultiModuleChild1

│ └── pom.xml (packaging = jar)

│

└── MultiModuleChild2

└── pom.xml (packaging = war, webapp)

⚡ Summary:

* **Parent** controls structure (pom).
* **Child1** = JAR project.
* **Child2** = WebApp (WAR) project.
* **Child2 depends on Child1** via <dependency>.
* **Build order matters** → Parent → Child1 → Child2.

Got it—here’s a clean, do-this-next guide for every item in your sheet. I’ll assume your Dockerfile is in the current folder and the app inside the image listens on port **8080** (adjust if yours differs).

**Docker CLI (Q1–Q10)**

1. **Build image named hospital-mgmt**

docker build -t hospital-mgmt:latest .

1. **List all Docker images**

docker images

# or

docker image ls

1. **Run and map *container* port 8080 → *host* port 9090**

docker run -d --name hospital-mgmt -p 9090:8080 hospital-mgmt:latest

* Visit: [http://localhost:9090](http://localhost:9090/)

1. **Pull official Redis, run it, list running containers, verify**

docker pull redis:latest

docker run -d --name redis -p 6379:6379 redis:latest

docker ps # shows running containers

# verify Redis responds:

docker exec -it redis redis-cli ping # expect: PONG

Note: Redis isn’t an HTTP server, so “[http://localhost:8080”](http://localhost:8080%E2%80%9D/) doesn’t apply.  
If your teacher really wants a browser check on :8080, use nginx instead:

docker run -d --name web -p 8080:80 nginx:latest

# now open http://localhost:8080

1. **Tag and push your app image to Docker Hub (yourusername/hms)**

docker login

docker tag hospital-mgmt:latest yourusername/hms:1.0

docker push yourusername/hms:1.0

1. **You forgot -p, can’t access the app — stop & rerun correctly**

docker ps # find the container name/ID

docker stop hospital-mgmt

docker rm hospital-mgmt

docker run -d --name hospital-mgmt -p 9090:8080 hospital-mgmt:latest

Port mappings can’t be added to an existing container; recreate it.

1. **Container crashes immediately — view logs & debug**

docker logs hospital-mgmt # last logs

docker logs -f --tail 200 hospital-mgmt # follow logs

docker ps -a # check status/exit code

docker inspect hospital-mgmt # env, ports, mounts, etc.

1. **Make your custom image public on Docker Hub**

* Steps are the same as #5 (login → tag → push).
* Ensure the repo yourusername/hms is **Public** on Docker Hub.

1. **Open a shell inside a running container**

docker exec -it hospital-mgmt /bin/bash # Debian/Ubuntu base

# or, if bash isn’t present (Alpine etc.)

docker exec -it hospital-mgmt /bin/sh

1. **List “removed” containers**

* Once a container is truly **removed**, Docker doesn’t keep it in docker ps.
* What you *can* list:

docker ps -a # all (running + stopped)

docker ps -a --filter status=exited # ones you could remove

# To see recent deletions (event stream):

docker events --since 24h --filter event=destroy

**5. Docker Compose (10M)**

**Goal:** App + MongoDB. App runs on host port **7079** and uses Mongo as DB.

Create docker-compose.yml in an empty folder:

version: "3.8"

services:

app:

image: yourusername/hms:1.0 # the image you pushed

container\_name: campusmgmt-app

ports:

- "7079:7079" # if your app listens on 7079 inside

# If your app listens on 8080 inside, use: "7079:8080"

environment:

# Adjust to what your app expects (examples):

- MONGO\_URL=mongodb://mongo:27017/campusdb

- MONGO\_DB=campusdb

depends\_on:

- mongo

mongo:

image: mongo:6

container\_name: campusmgmt-mongo

restart: always

ports:

- "27017:27017"

volumes:

- mongo-data:/data/db

volumes:

mongo-data:

**Run it:**

docker compose up -d

**Useful follow-ups:**

docker compose ps # show both containers

docker compose logs -f app # app logs

docker exec -it campusmgmt-mongo mongosh # open Mongo shell

# In mongosh:

# use campusdb

# db.createCollection("test")

**Tear down (keeping data volume):**

docker compose down

**Tear down and remove volumes (DB wiped):**

docker compose down -v

MAVEN questions:

Perfect 👌 I’ll break down **step-by-step solutions** for every question in your image (Q7–Q13, Q3 Git & GitHub Integration, and Q4 Docker containerization).

**Q7–Q13: Maven Questions**

**7. Add Java Servlet API dependency in pom.xml**

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

<version>4.0.0-b01</version>

<scope>provided</scope>

</dependency>

* Add to <dependencies> block in pom.xml.
* Run: mvn clean install.

**8. Developer removes <dependencies> completely – issues?**

* Build fails because no required libraries are available.
* Tests fail since dependencies (JUnit, Servlet, etc.) are missing.
* Compilation errors: *“package javax.servlet does not exist”*.

**9. Maven error – JRE instead of JDK**  
Error:

Compilation failure: [ERROR] tools.jar not found

* Occurs because Maven needs **JDK** not **JRE**.  
  ✅ Fix:
* Install JDK.
* Update JAVA\_HOME to JDK path:

export JAVA\_HOME=/usr/lib/jvm/java-17-openjdk

**10. WAR final name in pom.xml**

<build>

<finalName>foodSystem</finalName>

</build>

Deploys as:

http://localhost:8080/foodSystem

**11. WAR packaging**

<packaging>war</packaging>

* Needed to deploy on Tomcat.
* Without it → default jar, which Tomcat cannot deploy.

**12. Missing <url> in pom.xml**

<url>http://maven.apache.org</url>

* Purpose: Provides project homepage.
* Used in generated Maven site/docs.

**13. Push Maven project to GitHub**

git init

git add pom.xml src/

git commit -m "Initial Maven project"

git branch -M main

git remote add origin https://github.com/<username>/<repo>.git

git push -u origin main

**Q3: Git & GitHub Integration (30M)**

**1. Discard unstaged changes**

git checkout -- filename

# or (all files)

git restore .

**2. Fix last commit message**

git commit --amend -m "Correct commit message"

**3. View commit history (pretty)**

git log --oneline --graph --decorate

**4. Create & switch branch**

git checkout -b Feature/patientand

**5. Push new branch to remote**

git push -u origin Feature/patientand

**6. Push commits from local main**

git push origin main

**7. See all branches**

git branch -a

**8. Fetch + merge remote changes**

git fetch origin

git merge origin/main

**9. Push first time (set upstream)**

git push -u origin branchName

**10. Configure new remote**

git remote add neworigin <url>

git push -u neworigin branchName

**11. Local branch behind remote**

git pull --rebase

**12. Delete branch on remote**

git push origin --delete patient

**13. Apply .patch file**

git apply file.patch

git add .

git commit -m "Applied teammate patch"

**Q4: Docker Containerization (20M)**

**Task: Maven → Docker with Tomcat**

**1. Create Dockerfile**

# Use Maven to build WAR

FROM maven:3.9.5-eclipse-temurin-17 AS builder

WORKDIR /app

COPY . .

RUN mvn clean package -DskipTests

# Deploy WAR to Tomcat

FROM tomcat:9.0-jdk17

COPY --from=builder /app/target/\*.war /usr/local/tomcat/webapps/hospitalmgmt.war

EXPOSE 8080

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

**2. Build & run**

docker build -t hospital-mgmt .

docker run -d -p 8080:8080 hospital-mgmt

Visit: <http://localhost:8080/hospitalmgmt>

**3. Push to DockerHub**

docker tag hospital-mgmt yourusername/hospitalmgmt:1.0

docker push yourusername/hospitalmgmt:1.0

**Q1 — Software Requirement Specification (SRS) for Hospital Management System (10M)**

Write short, exam-style answers for each part.

**a) Abstract (4M) — what to write (3–6 sentences)**

* One-line purpose: *“The Hospital Management System (HMS) is a web application to manage patients, appointments, staff, billing and medical records.”*
* Key functions in a sentence: *“It stores patient demographics, schedules appointments, records diagnosis/prescriptions, and produces billing & reports.”*
* Constraints/benefit: *“Provides role-based access, secure data storage and improves operational efficiency.”*  
  **Example (pasteable):**

**Abstract:** The Hospital Management System (HMS) is a web-based application designed to manage patient records, appointments, medical staff schedules, and billing. It centralizes patient data, supports role-based access control (admin/doctor/receptionist), and produces reports for hospital administration. The system ensures confidentiality and auditability and reduces manual paperwork to improve operational efficiency.

**b) Functional Requirements (2M) — short bullet list**

Give 5–7 clear items:

* Patient registration (create/read/update/delete)
* Appointment booking/cancellation
* Doctor schedule management
* Prescription & medical records storage
* Billing & invoice generation
* Search patients by ID/name

**c) Non-Functional Requirements (2M)**

Bulleted measurable items:

* Security: HTTPS, role-based authentication, encrypted DB fields
* Performance: < 2s response for key pages under normal load
* Availability: 99% uptime (or “deployable on Tomcat with DB replication”)
* Scalability: App must support N concurrent users by scaling horizontally
* Usability: Responsive UI

**d) Identification of Users (2M)**

List major user roles and a one-line responsibility each:

* **Admin** — manage users, system configuration, reports
* **Doctor** — view patient history, add diagnosis/prescriptions
* **Nurse** — view schedules, update vitals
* **Receptionist** — register patients, book appointments, billing clerk

**Q2 — Maven Java Application Development (30M)**

You’re asked to clone the repo, fix pom.xml issues and run the build. Steps + exact commands and pom.xml edits below.

**Preliminaries (install/check)**

1. Install Java (JDK) and Maven.
2. Verify:

java -version

mvn -version

**1) Download the repository and list files (3M)**

**Commands:**

# clone

git clone https://github.com/Kumbhambhargavi75/HospitalMgmtSystem.git

cd HospitalMgmtSystem

# list files on Linux/macOS

ls -la

# or on Windows PowerShell

dir

If you want a tree:

# Linux

tree -a

Answer: show output of ls -la (or attach screenshot).

**2) mvn CLEAN package giving “unknown lifecycle phase” — why? (2M)**

**Cause:** Maven lifecycle phases and goals are case-sensitive — you used CLEAN instead of clean.  
**Fix / Correct command:**

mvn clean package

(Also ensure you’re in project root where pom.xml exists.)

If you still get unknown lifecycle phase, check mvn -v and ensure Maven installed.

**3) Add dependency servlet-api 2.5 (2M)**

Open pom.xml and add in <dependencies> **with scope provided** (Tomcat supplies servlet API):

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId>

<version>2.5</version>

<scope>provided</scope>

</dependency>

Why provided? So the servlet classes are not bundled inside the WAR (Tomcat already provides them).

After edit, run:

mvn clean package

**4) Build fails with JDK 21 — cause and pom.xml fix (2M)**

**Common cause(s):**

* maven-compiler-plugin or <maven.compiler.source>/<target> not set to match JDK (or set to older keyword), or plugin version incompatible with Java 21.
* The project expects an older Java (e.g., source/target 1.8) but you compile with 21 — or vice versa.

**Fix:** set compiler properties and (optionally) use release or update plugin. Add to pom.xml:

**Simple properties approach** (change 21 to the Java version you want to compile for — typically 17 or 21):

<properties>

<maven.compiler.source>21</maven.compiler.source>

<maven.compiler.target>21</maven.compiler.target>

<!-- or use release instead:

<maven.compiler.release>21</maven.compiler.release>

-->

</properties>

**Safer: explicit plugin config (recommended):**

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.11.0</version> <!-- choose a recent version -->

<configuration>

<release>21</release>

</configuration>

</plugin>

</plugins>

</build>

After editing:

mvn clean package

If you actually must build for an older Java (project code uses older APIs), install/use that JDK (set JAVA\_HOME) or change <release> to a matching version.

**5) Dependency has wrong groupId (SE:junit:4.6.0) — what will Maven do and how to fix? (2M)**

**What Maven does:** Maven attempts to resolve that coordinate (SE:junit:4.6.0) from remote repos. Because groupId is incorrect, it will likely fail to find the artifact → build error.

**Fix:** Use correct JUnit dependency and scope test. Example JUnit 4 (classic):

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13.2</version>

<scope>test</scope>

</dependency>

Or for JUnit 5:

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter</artifactId>

<version>5.9.3</version>

<scope>test</scope>

</dependency>

**6) WAR built as hospitalmgmtsystem-0.0.1-SNAPSHOT.war and you want HospitalMgmtSystem.war (3M)**

Maven names WAR as artifactId-version[-SNAPSHOT].war by default. To set a custom name, add <finalName> in <build>:

<build>

<finalName>HospitalMgmtSystem</finalName>

</build>

Then run:

mvn clean package

# target/HospitalMgmtSystem.war will be generated

(Alternatively change <artifactId> and <version> but finalName is simplest.)

**Quick sample pom.xml fragments to paste into your file**

**Add servlet dependency + compiler + finalName + junit fix (combined)**

<project ...>

...

<properties>

<maven.compiler.release>21</maven.compiler.release>

<!-- or set source/target:

<maven.compiler.source>21</maven.compiler.source>

<maven.compiler.target>21</maven.compiler.target>

-->

</properties>

<dependencies>

<!-- servlet API -->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId>

<version>2.5</version>

<scope>provided</scope>

</dependency>

<!-- junit for tests -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13.2</version>

<scope>test</scope>

</dependency>

<!-- other dependencies... -->

</dependencies>

<build>

<finalName>HospitalMgmtSystem</finalName>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.11.0</version>

<configuration>

<release>21</release>

</configuration>

</plugin>

</plugins>

</build>

</project>

**Final check list (commands to run end-to-end)**

# 1. clone & list

git clone https://github.com/Kumbhambhargavi75/HospitalMgmtSystem.git

cd HospitalMgmtSystem

ls -la

# 2. fix pom.xml as above

# 3. build

mvn clean package

# 4. confirm WAR

ls -la target

# you should see target/HospitalMgmtSystem.war