# Solution for Campus Event Management System

**Q1) SRS  
a. Abstract**

This project aims to develop a centralized **Campus Event Management System** to streamline the organization, promotion, and participation in college events. The system will provide a unified platform for students, faculty, and administrators to manage the entire event lifecycle. Key functionalities will include event creation with detailed descriptions, user-friendly registration for attendees, a dynamic scheduling calendar, and a real-time notification system for updates and reminders. By offering an intuitive and accessible interface, the primary goal is to increase event attendance, enhance communication between organizers and participants, and improve the overall efficiency of managing campus activities.

**b. Functional Requirements**

Functional requirements define what the system must do. Based on the description, two key functional requirements are:

1. **Event Creation & Management:** The system must allow authorized users (e.g., administrators, faculty) to create, edit, and publish events. This includes adding details such as event title, description, date, time, location, and registration limits.
2. **User Registration for Events:** The system must allow users (primarily students) to view a list of upcoming events and register (RSVP) for those they wish to attend. The system should also allow them to view the events they have registered for.

**c. Non-Functional Requirements**

Non-functional requirements define the quality attributes of the system. Two key non-functional requirements are:

1. **Usability:** The system must have an **intuitive interface** that is easy to navigate for all user types (students, faculty, administrators) with minimal training.
2. **Performance:** The system must be responsive, especially the **real-time notification** feature, ensuring that alerts and updates are delivered to users promptly without significant delay.

**d. Identification of Users**

The scenario explicitly identifies three main types of users for the system:

1. **Students:** They are the primary audience who will use the system to discover, register for, and receive notifications about campus events.
2. **Faculty:** They can be event organizers, creating and managing departmental or academic events, as well as participants in general campus events.
3. **Administrators:** They have high-level control over the system, managing user accounts, overseeing all events, and ensuring the platform runs smoothly.

**Q2) Maven Java Application Development**

**1. Download the given repository and show the list of the files?**

**To download (clone) the repository:**  
Open your terminal or Git Bash and run the following command:

Bash

git clone https://github.com/KumbhamBhargavi75/CampusMgmtSystem.git

**To show the list of files:**  
After cloning, navigate into the project directory and list the files.

Bash

# Navigate into the newly created directory

cd CampusMgmtSystem

# List the files and directories (the -R flag lists them recursively)

ls -R

**Expected Output (a simplified version):**

text

.:

pom.xml

src/

./src:

main/

test/

./src/main:

java/

resources/

webapp/

./src/main/java/com/in/nit:

CampusMgmtSystemApplication.java

controller/

entity/

repository/

service/

...and so on for other directories and files.

**2. While executing getting an error as**-[ERROR] Failed to execute goal org.apache.maven.plugins:maven-compiler-plugin:3.8.1:compile**.**

This error message is a generic failure notification from the Maven compiler plugin. It simply means **"compilation failed"**. The *actual cause* of the error is usually printed just before this line in the logs.

**Common Causes and How to Debug:**

1. **Syntax Errors in Java Code:** There might be a typo, a missing semicolon, an incorrect import, or other language errors in one of your .java files.
   * **Solution:** Scroll up in the Maven output log. It will point to the exact file and line number containing the syntax error (e.g., [ERROR] /path/to/your/file.java:[10,5] ';' expected). Correct the code in that file.
2. **JDK Version Mismatch:** The project is configured to be compiled with Java 11 (as seen in pom.xml), but you might be running Maven with an older JDK (like JDK 8).
   * **Solution:** Ensure your JAVA\_HOME environment variable points to a JDK 11 (or newer) installation, and that your system's path includes its bin directory.

**To get more details, run Maven with the**-e**(show errors) flag:**

Bash

mvn clean install -e

**3. Source option 11 is not supported. How to resolve this? The project uses JUnit 3.8.1, but modern Java (21) requires JUnit 4/5. What changes would you make in pom.xml to upgrade JUnit?**

This question has two parts.

**Part A: "Source option 11 is not supported."**

* **Cause:** This error confirms the JDK version mismatch mentioned above. You are trying to compile a project that requires Java 11 using a JDK that is older (e.g., JDK 8).
* **Resolution:**
  1. Install a compatible JDK (version 11 or higher).
  2. Configure your system's JAVA\_HOME environment variable to point to the installation directory of the new JDK.
  3. If you are using an IDE like Eclipse or IntelliJ, make sure the project's configured JDK is also set to version 11 or higher.

**Part B: Upgrade JUnit**

* **Note:** The repository actually uses JUnit **4.11**, not 3.8.1. We will upgrade from version 4 to 5 (Jupiter).

To upgrade to **JUnit 5**, you need to replace the old JUnit 4 dependency with the JUnit 5 Jupiter API and Engine.

**1. Remove the old JUnit 4 dependency:**

XML

<!-- REMOVE THIS BLOCK -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.11</version> <!-- or 3.8.1 -->

<scope>test</scope>

</dependency>

**2. Add the JUnit 5 (Jupiter) dependencies:**  
The spring-boot-starter-test dependency already includes JUnit 5 by default. You just need to make sure you are not excluding it and that the old JUnit 4 is removed. The project's pom.xml already has the correct starter, but it also has the old junit:junit dependency, which should be removed.

The correct dependency block for testing with Spring Boot and JUnit 5 is:

XML

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

This single dependency transitively brings in junit-jupiter-api, junit-jupiter-engine, and other testing libraries like Mockito.

**4. In the dependency section, the version of JUnit is left out like this:**<dependency><groupId>junit</groupId><artifactId>junit</artifactId></dependency>**. What will Maven do? How can you ensure a proper version is used?**

**What Maven Will Do:**  
If a version is omitted, Maven will try to resolve it using **Dependency Management**. It searches for a version defined in a <dependencyManagement> section in either:

1. The current project's pom.xml.
2. The parent pom.xml (and its parents, recursively).

In this project, the parent is spring-boot-starter-parent. This parent POM defines managed versions for hundreds of common libraries, including JUnit. Therefore, **Maven will inherit the version of JUnit specified by the**spring-boot-starter-parent**POM.**

**How to Ensure a Proper Version is Used:**  
There are two primary ways to explicitly set a version:

1. **Add a**<version>**tag directly:** This is the most straightforward way and will override any version defined in the parent POM.

XML

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13.2</version> <!-- Explicitly setting the version -->

<scope>test</scope>

</dependency>

1. **Use a property:** This is a best practice for managing versions across multiple dependencies. Define a property in the <properties> section and reference it.

XML

<properties>

<java.version>11</java.version>

<junit.version>4.13.2</junit.version> <!-- Define the version as a property -->

</properties>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>${junit.version}</version> <!-- Reference the property -->

<scope>test</scope>

</dependency>

</dependencies>

**5. Suppose you misspelled**<artifactId>cms</artifactId>**in the**<dependency>**block. What kind of error will Maven throw during**mvn install**? How do you debug such XML mistakes?**

Let's assume you misspelled the artifactId of a known dependency, for example, junit as junitt.

**Kind of Error Thrown:**  
Maven will fail during the **dependency resolution phase** of the build lifecycle. The build will stop, and you will see an error message similar to this:

text

[ERROR] Failed to execute goal on project CampusMgmtSystem: Could not resolve dependencies for project com.in.nit:CampusMgmtSystem:war:0.0.1-SNAPSHOT: The following artifacts could not be resolved: junit:junitt:jar:4.12: Could not find artifact junit:junitt:jar:4.12 in central (https://repo.maven.apache.org/maven2) -> [Help 1]

The key phrases are **"Could not resolve dependencies"** and **"Could not find artifact"**.

**How to Debug:**

1. **Read the Error Message:** The message explicitly tells you the groupId, artifactId, and version it failed to find (junit:junitt:jar:4.12).
2. **Check for Typos:** Carefully compare the artifact coordinates in your pom.xml with the correct ones. This is the most common cause.
3. **Search Maven Central:** Go to a repository browser like [search.maven.org](https://search.maven.org/) and search for the dependency. This will confirm the correct groupId, artifactId, and available versions.
4. **Check Repository Configuration:** If the dependency is not in Maven Central, ensure you have correctly configured the custom repository in your pom.xml or settings.xml file where the artifact is hosted.

**6. You written like this**<paking>**tag in the POM. What does Maven assume by default, and what impact will it have on this web project?**

* **The Error:** The correct tag is <packaging>. The tag <paking> is an invalid XML tag and would be ignored by Maven, or in a stricter validation, cause a parsing error. Assuming it's simply ignored or omitted...
* **Maven's Default Assumption:** If the <packaging> tag is omitted, Maven assumes the default value, which is jar.
* **Impact on this Web Project:**
  1. **Incorrect Artifact:** The project will be built as a .jar file instead of the intended .war file (Web Application Archive).
  2. **Deployment Failure:** A WAR file has a specific structure (WEB-INF/, WEB-INF/lib, etc.) required by servlet containers like Apache Tomcat. A JAR file lacks this structure. Therefore, the resulting .jar file cannot be deployed to a standard external servlet container.
  3. **Different Build Lifecycle:** Maven uses a different set of build lifecycle plugins for jar packaging versus war packaging. For example, the maven-war-plugin goals will not execute, and JSP/static files might not be packaged correctly.

**7. Currently, only JUnit is included. If the project needs to connect to MySQL, what dependency would you add?**

* **Note:** The provided repository *already includes* the dependency for MySQL. However, if it were missing, this is what you would add.

You need to add the **MySQL Connector/J** dependency to your pom.xml.

XML

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<scope>runtime</scope> <!-- 'runtime' is appropriate as it's needed at runtime, not compile time -->

</dependency>

*Note: Spring Boot's parent POM will manage the version, so you often don't need to specify it.*

**8. How can I add the JQuery 3.6.1 dependency to your project. Check how it works?**

jQuery is a front-end JavaScript library. You don't add it like a typical Java dependency. There are two main methods:

**Method 1: Using WebJars (The "Maven" Way)**  
WebJars package client-side web libraries into JAR files. Spring Boot has excellent auto-configuration for them.

1. **Add the WebJars dependency to**pom.xml**:**

XML

<dependency>

<groupId>org.webjars</groupId>

<artifactId>jquery</artifactId>

<version>3.6.1</version>

</dependency>

1. **How it works:**
   * Maven downloads a JAR containing the jQuery files.
   * Spring Boot automatically configures a resource handler that serves files from any WebJars JAR on the classpath under the /webjars/\*\* path.
   * You can then reference it in your HTML or JSP file like this:

HTML

<script src="/webjars/jquery/3.6.1/jquery.min.js"></script>

**Method 2: Manually Adding to Static Resources**

1. Download the jquery-3.6.1.min.js file from the official jQuery website.
2. Place it in the static resources directory of your project: src/main/resources/static/js/.
3. Reference it in your HTML or JSP with a relative path:

HTML

<script src="/js/jquery-3.6.1.min.js"></script>

**9. Identify if the**</artifactId>**is written correct or not. How will this affect your generated jar, and how would you fix it?**

This question seems to be asking what happens if you have an XML syntax error, like a typo in a closing tag (e.g., <artifactId>my-app</artfactId>).

* **Is it correct?** No, an XML tag must have a matching opening and closing tag. </artfactId> does not match <artifactId>.
* **Effect on the Generated JAR:**  
  The pom.xml file would be **invalid XML**. Maven parses this file before it does anything else. The build will **fail immediately** during the "project validation" phase with an XML parsing error.  
  **No JAR will be generated at all.** The build process will not even start the compilation or packaging phases.
* **How to fix it:**  
  Correct the typo in the closing tag to make the XML well-formed.  
  **Incorrect:** <artifactId>my-app</artfactId>  
  **Correct:** <artifactId>my-app</artifactId>

**10. & 11. You are required to change the default build output directory from**target/**to**build\_output/**. How would you configure it in pom.xml?**

These two questions ask the same thing. The default build directory in Maven is ${project.basedir}/target. To change it, you need to add a <directory> tag inside the <build> section of your pom.xml.

**Configuration:**  
Add the following snippet inside the <project> tags in your pom.xml:

XML

<build>

<!-- This sets the output directory for build artifacts -->

<directory>${project.basedir}/build\_output</directory>

</build>

* ${project.basedir} is a Maven property that refers to the project's root directory (where the pom.xml is located). This ensures the path is always relative to the project root.
* After this change, running mvn clean install will create a build\_output/ directory instead of target/ and place all compiled classes and the final artifact (.war file) inside it.

**12. Add the central dependency of JUnit - 4.13-beta-2 to your existing project. check the complete pom.xml file and run it.**

To do this, you need to find the existing junit dependency in the pom.xml and update its version.

1. **Locate the dependency block:**

XML

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.11</version> <!-- This is the old version -->

<scope>test</scope>

</dependency>

1. **Update the version:** Change <version>4.11</version> to <version>4.13-beta-2</version>.

XML

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13-beta-2</version> <!-- This is the new version -->

<scope>test</scope>

</dependency>

1. **Run it:** To verify the change and run the project's tests, execute the following Maven command from your terminal in the project's root directory:

Bash

mvn clean test

This command will clean previous build artifacts, compile the code, and run all tests using the newly specified JUnit version.

**13. Push the updated project into GitHub of your account using maven gitbash environment and check?**

This is a Git workflow task. Assuming you have forked the repository to your own GitHub account and have cloned it locally:

**Steps in Git Bash:**

1. **Check Status:** First, see what files you have changed.

Bash

git status

*(This will show pom.xml as modified).*

1. **Stage Changes:** Add the modified pom.xml to the staging area.

Bash

git add pom.xml

*(Or use git add . to stage all changed files).*

1. **Commit Changes:** Commit the staged files with a descriptive message.

Bash

git commit -m "Update pom.xml with new dependencies and build configuration"

1. **Push to GitHub:** Push your committed changes to your remote repository on GitHub.

Bash

# 'origin' is the default name for your remote repository.

# 'main' or 'master' is the name of the branch you are working on.

git push origin main

1. **Check:** Go to your GitHub repository in your web browser. You will see the new commit with your message, and you can view the pom.xml file to confirm that the changes are live.

**Q3. Git & GitHub Integration with Maven Project (30M)**

Here are the Git commands and workflows to perform the required tasks.

**1. You want to start tracking your Campus Event Management System project in Git for the first time.**

To initialize a new Git repository in your existing project directory, navigate to the project's root folder in your terminal and run:

Bash

git init

**2. You want to save a snapshot of the current project with a message "Added event registration feature".**

This is a two-step process: first, you stage the files you want to save, and then you commit them with a message.

Bash

# Stage all new and modified files in the project

git add .

# Commit the staged files with the specified message

git commit -m "Added event registration feature"

**3. You want to see the current status of modified and staged files.**

To check which files are modified, staged for the next commit, or untracked, use:

Bash

git status

**4. You want to view the commit history of your project to check what changes were made.**

To see a detailed list of all commits, including the author, date, and commit message, use:

Bash

git log

For a more compact, one-line view, you can use:

Bash

git log --oneline

**5. You cloned a repository but want to see all branches in it.**

To list all branches, including local branches and remote-tracking branches (those on the remote server like GitHub), use:

Bash

git branch -a

**6. You created a new feature branch called event-scheduler.**

To create a new branch, you can use git branch. A common practice is to create the branch and switch to it in one command.

Bash

# Creates the new branch and immediately switches to it

git checkout -b event-scheduler

*Alternatively, you could do it in two steps: git branch event-scheduler followed by git checkout event-scheduler.*

**7. You received a .patch file from a teammate fixing bugs in the event notification module.**

You can apply changes from a .patch file using the git apply command.

Bash

# First, check if the patch can be applied cleanly without errors

git apply --check name-of-the.patch

# If there are no errors, apply the patch

git apply name-of-the.patch

**8. You want to create a new branch for online feedback functionality, make multiple commits, and then merge it cleanly into main.**

This describes a standard feature branching workflow.

Bash

# 1. Create and switch to the new feature branch

git checkout -b online-feedback

# 2. Work on the feature, making multiple commits

# ... make your first set of changes ...

git add .

git commit -m "feat: Add UI for feedback form"

# ... make your second set of changes ...

git add .

git commit -m "feat: Implement backend for feedback submission"

# 3. Switch back to the main branch

git checkout main

# 4. Merge the feature branch into main

git merge online-feedback

# 5. (Optional but good practice) Delete the feature branch

git branch -d online-feedback

**9. You encounter merge conflicts when merging event-scheduler with main.**

Here is the process to resolve merge conflicts:

1. **Attempt the merge**, which will fail and report the conflicts.

Bash

git merge event-scheduler

# Output will show: "Automatic merge failed; fix conflicts and then commit the result."

1. **Identify and resolve conflicts**. Open the files listed as conflicted in your code editor. You will see markers like <<<<<<< HEAD, =======, and >>>>>>> event-scheduler. Edit the file to keep the correct code and remove these markers.
2. **Stage the resolved files**. After fixing a file, you must stage it to mark it as resolved.

Bash

git add <path/to/conflicted-file.java>

1. **Commit the merge**. Once all conflicts are resolved and staged, create the merge commit to complete the process.

Bash

git commit

# A pre-populated commit message will open in your editor. Save and close it.

**10. You want to collaborate using a fork-and-pull request workflow to contribute a new feature.**

This is the standard workflow for contributing to open-source or team projects on GitHub.

1. **Fork the Repository:** On the GitHub page of the original project, click the "Fork" button to create a personal copy under your account.
2. **Clone Your Fork:** Clone the repository from *your* account to your local machine.

Bash

git clone https://github.com/YourUsername/CampusMgmtSystem.git

1. **Create a Feature Branch:**

Bash

git checkout -b my-new-feature

1. **Commit and Push:** Make your changes, commit them, and push the new branch to *your fork*.

Bash

git add .

git commit -m "feat: Implement awesome new feature"

git push origin my-new-feature

1. **Create a Pull Request (PR):** Go to your forked repository on GitHub. You will see a banner to "Compare & pull request". Click it, write a clear title and description for your changes, and submit the PR to the original repository.

**11. You want to see what changes are present between your local branch and main on GitHub.**

First, ensure your local repository is aware of the latest changes on GitHub, then use git diff to compare.

Bash

# 1. Fetch the latest updates from the remote repository (named 'origin')

git fetch origin

# 2. Compare your current branch with the remote main branch

git diff origin/main

*If you are on a different branch, specify it: git diff my-feature-branch origin/main.*

**12. Your teammate pushed a new branch event-feedback to GitHub. You want to test it locally.**

To check out a branch that exists on the remote but not yet locally:

Bash

# 1. Fetch all remote branches and updates

git fetch origin

# 2. Switch to the desired branch. Git will automatically create a local

# branch that tracks the remote one.

git checkout event-feedback

**13. You want to see the differences introduced by the last commit on your branch.**

To view the changes (the "diff") made in the most recent commit on your current branch, use:

Bash

git show

This command displays the commit details (author, date, message) and the code changes for the latest commit (HEAD). An alternative, more explicit command is:

Bash

git diff HEAD~1 HEAD

**Q4. Docker containerization for Maven JAVA Application (20M)**

This section guides you through the process of containerizing a Maven-based Java web application, managing the Docker image and containers.

**Prerequisites:**

1. Docker must be installed and running on your machine.
2. You have a Docker Hub account.
3. You have cloned the repository: git clone https://github.com/Kumbhambhargavi75/CampusMgmtSystem.git
4. You have Maven installed to build the project locally, or you can use a multi-stage Dockerfile which is the recommended approach.

**Solution for Docker Containerization Questions:**

**a, b, c. Writing the Dockerfile**

This task is directly addressed in question 4. A multi-stage Dockerfile is the best practice for a Maven project as it keeps the final image clean without including the Maven build tools.

Place this Dockerfile in the root directory of the cloned CampusMgmtSystem project.

**Dockerfile**

Dockerfile

# Stage 1: Build the application using Maven

FROM maven:3.8-jdk-11 AS builder

WORKDIR /app

COPY pom.xml .

COPY src ./src

# Build the project and create the WAR file

RUN mvn clean package

# Stage 2: Create the final image using Tomcat

FROM tomcat:9.0-jdk11-openjdk-slim

# Remove the default Tomcat welcome page

RUN rm -rf /usr/local/tomcat/webapps/\*

# Copy the WAR file from the builder stage to Tomcat's webapps directory

# Note: The WAR file will likely be named CampusMgmtSystem.war

COPY --from=builder /app/target/CampusMgmtSystem.war /usr/local/tomcat/webapps/ROOT.war

# Expose port 8080

EXPOSE 8080

# The base Tomcat image already has a CMD to start the server

* **Explanation:** This Dockerfile first uses a Maven image to build the .war file. Then, it uses a lightweight Tomcat image and copies *only* the built .war file into it. By naming it ROOT.war, the application will be accessible at the root URL (e.g., http://localhost:8080/) instead of http://localhost:8080/CampusMgmtSystem/.

**1. Check the docker version and list all the images existing**

* **To check the Docker version:**

Bash

docker --version

or for more details:

Bash

docker version

* **To list all existing Docker images on your system:**

Bash

docker images

**2. How can you pull the hello-world image from locally and run it.**  
*(Note: "from locally" is a bit ambiguous. The standard workflow is to pull from a registry like Docker Hub, which then stores it locally for running.)*

* **To pull the image from Docker Hub:**

Bash

docker pull hello-world

* **To run the image:**

Bash

docker run hello-world

(If the image is not present locally, this command will automatically pull it before running.)

**3. How do you Pull the image ubuntu and work interactive mode?**

* **To pull the Ubuntu image:**

Bash

docker pull ubuntu

* **To run it in interactive mode (which gives you a command prompt inside the container):**

Bash

docker run -it ubuntu /bin/bash

* + -i stands for interactive (keeps STDIN open).
  + -t allocates a pseudo-TTY (gives you a shell-like interface).

**4. Write a Dockerfile to create an image name as**CSIMAGE**for your Maven-based CampusMgmtSystem application. Ensure the image can build and run your Java application consistently.**

* **Dockerfile:** Use the multi-stage Dockerfile provided at the beginning of this answer.
* **Build the image:** Navigate to the project's root directory (where the Dockerfile is) and run:

Bash

docker build -t csimage .

* + -t csimage tags the image with the name csimage.
  + . indicates the build context is the current directory.
* **Run the container from the image:**

Bash

docker run -d -p 8080:8080 --name campusapp csimage

* + -d runs the container in detached mode (in the background).
  + -p 8080:8080 maps port 8080 of your host machine to port 8080 of the container.
  + --name campusapp gives the container a memorable name.

**5. pull the official image from Docker Hub, run it in a container and list of containers that are running.**

* **Let's use the**nginx**image as an example.**

Bash

# Pull the official nginx image

docker pull nginx

# Run it in a container

docker run -d -p 8888:80 --name mynginx nginx

# List all RUNNING containers

docker ps

**6. How can you start and stop a running container?**

* **To stop a container (using the name from the previous step):**

Bash

docker stop mynginx

* **To start a stopped container:**

Bash

docker start mynginx

**7. How can you check a list of all containers, including stopped ones.**

* **Use the**-a**or**--all**flag:**

Bash

docker ps -a

**8. How can you stop a running container?**

* This is a repeat of question 6. You use the docker stop command followed by the container's name or ID.

Bash

docker stop <container\_name\_or\_id>

Example: docker stop campusapp

**9. Push your custom Docker image (with the deployed application) to your Docker Hub account and make it available publicly.**

* **Step 1: Log in to Docker Hub**

Bash

docker login

(Enter your Docker Hub username and password when prompted.)

* **Step 2: Tag the image**  
  You must tag your image with your Docker Hub username. Replace your-dockerhub-username with your actual username.

Bash

docker tag csimage your-dockerhub-username/csimage:latest

* **Step 3: Push the image**

Bash

docker push your-dockerhub-username/csimage:latest

* **Step 4: Make it public**  
  By default, new repositories on Docker Hub are public. If it's private, you can make it public by going to your repository on the Docker Hub website (hub.docker.com) and changing its visibility in the "Settings" tab.

**10. How do you check whether a container exited cleanly or crashed?**

* You can inspect the container's exit code. An exit code of 0 means it exited cleanly. Any non-zero code indicates an error.

Bash

docker inspect <container\_name\_or\_id>

Look for the ExitCode under the "State" section in the JSON output.

* You can also check the container's logs to see the application's output and any error messages.

Bash

docker logs <container\_name\_or\_id>

**Q5. DOCKER COMPOSE (10M)**

This section requires you to use Docker Compose to manage multiple containers. Based on the question, you need to run two containers: a generic Tomcat server and your custom Campus Management System application.

**1, 2, 3. Create a**docker-compose.yml**file to run two containers.**

Create a new directory for your compose setup, and inside it, create a file named docker-compose.yml with the following content. Replace your-dockerhub-username with your actual Docker Hub username.

**docker-compose.yml**

YAML

version: '3.8'

services:

# Container 1: A standard Tomcat server

TomcatServer:

image: tomcat:9.0-slim

container\_name: plain-tomcat

ports:

- "8086:8080" # Access Tomcat at port 8086 on your host machine

# Container 2: Your Campus Management System application

CampusMgmtSystemproject:

image: your-dockerhub-username/csimage:latest # Use the image you pushed to Docker Hub

container\_name: campus-management-app

ports:

- "7007:8080" # Access your app at port 7007 on your host machine

**How to Run:**

1. Save the file as docker-compose.yml.
2. Open a terminal in the same directory as the file.
3. Run the following command to start both containers in the background:

Bash

docker-compose up -d

**How to Access:**

* **Container 1 (TomcatServer):** Open your web browser and go to http://localhost:8086. You will see the default Tomcat landing page.
* **Container 2 (Campus Management System):** Open your web browser and go to http://localhost:7007. You will see your Java application.