SE-LAB-SET2

QI. Software Requirement Specification (SRS)

You are assigned with a task to build a modular web application that serves as an online event management system. It allows users to view upcoming events, register/log in, book seats, and receive confirmation emails.

Based on this scenario, answer the following: [10 Marks]

1. Draft a concise Abstract for MJP (scope, key modules) – 4M
2. List five functional requirements (e.g., user registration, seat booking, email notification) – 2M
3. List three non-functional requirements (e.g., performance, error handling, maintainability) – 2M
4. Identify the primary user roles (e.g., admin, registered user, guest) and briefly describe their privileges – 2M

Here's a detailed and concise answer based on your scenario:

---

1. Abstract (Scope & Key Modules) – 4M

The Online Event Management System (OEMS) is a modular web application designed to facilitate seamless event discovery, registration, and seat booking for users. It enables users to view upcoming events, register or log in, book seats, and receive confirmation emails. The system also allows administrators to manage events, track bookings, and monitor user activity. Key modules include **User Management**, **Event Catalog**, **Seat Booking**, **Payment Integration**, and **Notification Service**.

---

2. Functional Requirements – 2M

1. **User Registration/Login:** Users can create accounts and securely log in to access personalized services.
2. **Event Browsing:** Users can view a list of upcoming events with details like date, time, venue, and availability.
3. **Seat Booking:** Registered users can select seats and book them for specific events.
4. **Email Notification:** Users receive confirmation emails for registration, booking, and cancellations.
5. **Admin Event Management:** Administrators can add, update, or remove events and monitor bookings.

---

3. Non-Functional Requirements – 2M

1. **Performance:** The system should handle multiple concurrent users without significant delays.
2. **Error Handling:** Proper validation and user-friendly error messages must be implemented for failed operations.
3. **Maintainability:** The application should be modular and easy to update or enhance with minimal downtime.

---

4. Primary User Roles & Privileges – 2M

Role	Privileges
Guest	Browse events, view event details, and access registration page.
Registered User	Book seats, manage bookings, receive notifications, and update profile.
Admin	Add/update/delete events, view all bookings, manage users, and generate reports.

If you want, I can also **draw a simple module diagram** for this system that visually shows how all modules interact—it's perfect for SRS submission. Do you want me to do that?

QII: Maven Java Project Development [30M]

Work with the Maven project available at: https://github.com/savram674/MJP.git

1. Clone the repository, run a clean install, to correct and compile all the errors in POM.xml– 10M
2. Add dependency for mysql-connector latest version from maven central repository and after clear build show it in the local .m2 maven local repository – 5M
3. Explain the archetype used for maven java project and the folder structure – 3M
4. Write the dependency snippet to include MySQL Connector/J 8.0.33 in your project. – 3M
5. How can you create an executable JAR with a main method using Maven, and which plugin helps configure this behavior? -- 4M
6. Incorporate a plugin (e.g., Maven Surefire) to run unit tests during build; – 3M
7. What files are generated in the target/ folder after running mvn clean install? -- 2M

Here's a detailed and structured answer for your Maven Java Project Development question:

---

1. Clone Repository and Run Clean Install – 10M

- **Step 1:** Clone the repository

```
git clone https://github.com/savram674/MJP.git
cd MJP
```

- **Step 2:** Run Maven clean install to compile the project and resolve errors in `pom.xml`

```
mvn clean install
```

- Maven will download all required dependencies, compile the code, run tests, and package the application. Any errors in `pom.xml` (like version mismatch or missing dependencies) need to be corrected before a successful build.

---

2. Add MySQL Connector Dependency and Verify in Local Repo – 5M

- **Step 1:** Add the dependency to `pom.xml` (latest version from Maven Central, e.g., 8.0.33)
- **Step 2:** Run Maven build

```
mvn clean install
```

- **Step 3:** Verify in local Maven repository:

```
C:\Users\<username>\.m2\repository\mysql\mysql-connector-java\8.0.33
```

---

3. Maven Archetype & Folder Structure – 3M

- **Archetype:** A Maven archetype is a template used to generate a standard project structure. For Java projects, `maven-archetype-quickstart` is commonly used.
- **Folder Structure:**

```
MJP/
├─ src/
│  ├─ main/
│  │  ├─ java/       # Java source files
│  │  └─ resources/  # Configuration files
│  └─ test/
│     ├─ java/       # Test classes
│     └─ resources/  # Test resources
├─ pom.xml           # Project Object Model file
└─ target/           # Compiled output & JAR
```

---

4. Dependency Snippet for MySQL Connector/J 8.0.33 – 3M

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.33</version>
</dependency>
```

---

5. Create Executable JAR with Main Method – 4M

- **Maven Plugin:** Use **Maven Shade Plugin** or **Maven Assembly Plugin** to create an executable JAR.
- **Example using Maven Shade Plugin:**

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-shade-plugin</artifactId>
      <version>3.4.1</version>
      <executions>
        <execution>
          <phase>package</phase>
          <goals><goal>shade</goal></goals>
          <configuration>
            <transformers>
              <transformer implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
                <mainClass>com.example.MainClass</mainClass>
              </transformer>
            </transformers>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

- Run:

```
mvn clean package
```

- Result: `target/MJP-1.0-SNAPSHOT.jar` executable with main method.

---

6. Incorporate Maven Surefire Plugin for Unit Tests – 3M

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>3.1.2</version>
    </plugin>
  </plugins>
</build>
```

- Maven will automatically run tests during `mvn clean install`.

---

7. Files Generated in `target/` Folder – 2M

After `mvn clean install`, the `target/` folder contains:

- Compiled `.class` files (`target/classes/`)
- Packaged JAR or WAR (`MJP-1.0-SNAPSHOT.jar`)
- Test reports (`target/surefire-reports/`)

- Temporary build files

If you want, I can **draw a complete Maven workflow diagram** showing dependency resolution, compilation, testing, and JAR creation—it's very helpful for exams.

Do you want me to do that?

i will give you question , explain in detail , simple words and in detail say what that functon does QIII. – Git & GitHub Integration with Maven Project [30M]

Using the Maven project created above, perform the following Git tasks:

1. You created a Maven project and now want to put it under version control. How will you initialize a new Git repository and connect it with a remote GitHub repo – 4M
2. After adding and committing your project, you realize the commit message was incorrect. Since you haven't pushed yet, how can you fix the commit message?. – 2M
3. What is .gitignore. how will you use it to exclude target – 2M
4. View details of a commit in compact form – 2M
5. You are in the middle of editing a file in your Maven project but suddenly need to switch to another branch. How can you save your changes temporarily and restore them later?. – 3M
6. Create a feature branch gaming, include two files and then merge it with main – 3M
7. Simulate a merge conflict between gaming and main and resolve it. – 4M
8. You want to see the differences introduced by the last commit on your branch. – 2M
9. Demonstrate git pull with explanation of how it differs from git fetch. – 3M
10. While working on Calculator.java, you accidentally staged it with git add but aren't ready to commit. How can you unstage it without losing the edits? – 3M
11. Modify the commit message of the last commit.

Here's a detailed, step-by-step explanation of each Git & GitHub task in simple words, including what each command/function does:

---

1. Initialize a new Git repository and connect to GitHub – 4M

- **Step 1:** Initialize Git in your project folder. This creates a hidden `.git/` folder to track changes.

```
git init
```

- **Step 2:** Add all project files to staging.

```
git add .
```

- **Step 3:** Commit files with a message.

```
git commit -m "Initial commit"
```

- **Step 4:** Connect to remote GitHub repository. Replace URL with your repo.

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

- **Step 5:** Push changes to GitHub.

```
git push -u origin main
```

Explanation:

- `git init` → starts version control
- `git add .` → stages files for commit
- `git commit` → saves snapshot
- `git remote add` → links local repo to GitHub
- `git push` → uploads changes to remote

---

2. Fix incorrect commit message (before push) – 2M

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

Explanation:

- `--amend` lets you edit the last commit's message. Since it's not pushed, it safely replaces the old commit.

---

3. What is `.gitignore` and exclude `target/` – 2M

- `.gitignore` is a text file listing files/folders Git should ignore.
- To ignore the Maven `target/` folder, create `.gitignore` in project root:

```
target/
*.log
```

Explanation:

- Prevents unnecessary or generated files from being tracked, reducing repo size.

---

4. View commit details in compact form – 2M

```
git log --oneline
```

Explanation:

- Shows each commit in a single line (short hash + message), making it easier to read history quickly.

---

5. Save changes temporarily (stash) to switch branches – 3M

```
git stash
git checkout <other-branch>
```

- Restore changes later:

```
git stash pop
```

Explanation:

- `git stash` → temporarily stores uncommitted changes
- `git checkout` → switches branch
- `git stash pop` → reapplies saved changes

---

6. Create a feature branch `gaming`, add files, merge with main – 3M

```
git checkout -b gaming        # Create & switch to branch
git add file1 file2           # Stage files
git commit -m "Add gaming feature"  # Commit
git checkout main             # Switch back to main
git merge gaming              # Merge feature into main
```

Explanation:

- Branching isolates new features; merging combines it with main.

---

7. Simulate and resolve merge conflict – 4M

- **Step 1:** Make conflicting changes in `gaming` and `main`.
- **Step 2:** Merge branch:

```
git checkout main
git merge gaming
```

- If conflict occurs, Git marks conflicts in files with:

```
<<<<<<< HEAD
Code from main
=======
Code from gaming
>>>>>>> gaming
```

- **Step 3:** Edit file to resolve conflict, then:

```
git add <file>
git commit
```

Explanation:

- Conflicts happen when same lines are modified in two branches. You manually fix them and commit.

---

8. See differences from last commit – 2M

```
git show
```

Explanation:

- Shows changes introduced by the latest commit (added/modified/removed lines).

---

9. Demonstrate `git pull` vs `git fetch` – 3M

```
git fetch origin         # Downloads updates from remote
git merge origin/main    # Merge updates manually

git pull origin main     # Shortcut: fetch + merge
```

Explanation:

- **`git fetch`** → updates remote tracking branches only, does not touch local files
- **`git pull`** → fetches **and** merges into local branch automatically

---

10. Unstage a file without losing edits – 3M

```
git reset HEAD Calculator.java
```

Explanation:

- Removes file from staging area but keeps your changes in the working directory.

---

11. Modify last commit message – 3M

```
git commit --amend -m "New commit message"
```

Explanation:

- Edits the last commit message. Safe if not yet pushed.

If you want, I can also **draw a diagram showing Git workflow** for this Maven project—branching, staging, committing, merging, and pushing. It makes it extremely easy to remember for exams.

Do you want me to create that diagram?

QIV. Docker containerization for Maven Application- [20M]
1. Write a Dockerfile that uses Openjdk, copy the jar into the working directory and then run using java – 5M
2. Build Docker image mjp-app using CLI and list local images to confirm – 3M
3. Run a container from mjp-app named mjp-container, to see the project output – 3M
4. Use a Docker CLI command to view active and inactive containers – 2M

5. Stop and then remove mjp-container using CLI commands – 3M
6. Pull a redis image and run it as container – 2M
7. Remove the mjp-app image with the required steps and confirm the same

Here's a detailed, step-by-step explanation for Docker containerization of your Maven application, in simple words:

---

1. Dockerfile for Maven Application – 5M

Create a `Dockerfile` in your project root:

```
# Use official OpenJDK image as base
FROM openjdk:17-jdk-alpine

# Set working directory inside the container
WORKDIR /app

# Copy the JAR file from local machine to container
COPY target/MJP-1.0-SNAPSHOT.jar app.jar

# Command to run the JAR
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Explanation:

- `FROM openjdk:17-jdk-alpine` → Uses lightweight OpenJDK image.
- `WORKDIR /app` → Sets working directory in container.
- `COPY target/MJP-1.0-SNAPSHOT.jar app.jar` → Copies compiled JAR to container.
- `ENTRYPOINT ["java", "-jar", "app.jar"]` → Runs the app when container starts.

---

2. Build Docker Image and List Local Images – 3M

```
docker build -t mjp-app .
docker images
```

Explanation:

- `docker build -t mjp-app .` → Builds image named `mjp-app` using Dockerfile in current directory (`.`).
- `docker images` → Lists all local Docker images to confirm build.

---

3. Run Container from mjp-app – 3M

```
docker run --name mjp-container mjp-app
```

Explanation:

- `--name mjp-container` → Names the container for easy reference.
- Container runs the JAR and shows output in terminal.

---

4. View Active & Inactive Containers – 2M

```
docker ps -a
```

Explanation:

- `docker ps` → Shows only running containers.
- `docker ps -a` → Shows all containers (running + stopped).

---

5. Stop and Remove mjp-container – 3M

```
docker stop mjp-container
docker rm mjp-container
```

Explanation:

- `docker stop` → Stops a running container.
- `docker rm` → Deletes container from local system.

---

6. Pull Redis Image and Run Container – 2M

```
docker pull redis
docker run --name my-redis -d redis
```

Explanation:

- `docker pull redis` → Downloads latest Redis image.
- `-d` → Runs container in detached mode (background).

---

7. Remove mjp-app Image and Confirm – 2M

```
docker rmi mjp-app
docker images
```

Explanation:

- `docker rmi mjp-app` → Deletes Docker image.
- `docker images` → Confirm image is removed from local system.

If you want, I can also **draw a diagram showing Docker workflow for the Maven app** including image build, container run, stop, and removal—it's very handy for exams.

Do you want me to make that diagram?

QV. DOCKER COMPOSE [10M]
Create a multi-container setup for MJP:
1. Write a docker-compose.yml file that configures two services:
 app: build the dockerfile created in previous section
 db: uses a PostgreSQL, with volume for persistence – 6M

2. Using Docker Compose CLI:
 Bring services up (docker compose up -d) and confirm both are running – 2M
 Use a Docker Compose command to view the logs of the app service – 2M

Here's a detailed step-by-step answer for your Docker Compose question:

---

1. `docker-compose.yml` for Multi-Container Setup – 6M

Create a file `docker-compose.yml` in your project root:

```yaml
version: "3.9"

services:
  app:
    build: .
    container_name: mjp-app
    ports:
      - "8080:8080"        # Map host port to container port
    depends_on:
      - db
    environment:
```

```
      - SPRING_DATASOURCE_URL=jdbc:postgresql://db:5432/mjpdb
      - SPRING_DATASOURCE_USERNAME=postgres
      - SPRING_DATASOURCE_PASSWORD=postgres

  db:
    image: postgres:15
    container_name: mjp-db
    environment:
      POSTGRES_DB: mjpdb
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
    volumes:
      - pgdata:/var/lib/postgresql/data
    ports:
      - "5432:5432"

volumes:
  pgdata:
```

Explanation:

- `app` service builds the Maven Dockerfile (`build: .`) and depends on `db`.
- `db` service uses official PostgreSQL image with a persistent volume `pgdata` to save data.
- `depends_on` ensures the database starts before the app.
- `environment` sets database credentials and connection URL.
- `volumes` ensure PostgreSQL data persists even if the container stops.

---

2. Docker Compose CLI Commands – 4M

- **Bring up services in detached mode:**

```
docker compose up -d
```

- **Check running containers:**

```
docker compose ps
```

- **View logs of `app` service:**

```
docker compose logs app
```

Explanation:

- `up -d` → Starts all services in background.
- `compose ps` → Confirms which containers are running.
- `compose logs app` → Shows the logs for the app container to verify output or errors.

If you want, I can **draw a simple diagram showing the app and PostgreSQL container interaction with volume and network** for this Docker Compose setup—it makes it very easy to visualize for exams.

Do you want me to create that diagram?