Step 1: Maven Setup and Compilation

1. **Developer defines dependencies in `pom.xml`** – The `pom.xml` file is the core of Maven's dependency management. The developer specifies required dependencies, such as:

   ```
   <dependencies>
       <dependency>
           <groupId>org.springframework.boot</groupId>
           <artifactId>spring-boot-starter-web</artifactId>
           <version>3.1.0</version>
       </dependency>
   </dependencies>
   ```

   Here, the `spring-boot-starter-web` dependency is defined, allowing the Spring Boot application to support web functionalities.
2. **Maven fetches all required dependencies** – Once dependencies are defined, running `mvn clean install` will automatically download required libraries (like Spring Boot, Lombok, Hibernate, etc.) from the Maven Central Repository and store them in the local repository (`.m2` folder).
3. **Maven Compiler compiles the Java code** – Maven compiles all Java classes and checks for errors. The compilation ensures that all dependencies and Java files work together correctly before running the application.

Step 2: Testing

4. **Developer runs tests using `mvn test`** – This command runs unit tests to verify that the code behaves as expected.
5. **Maven Test executes unit and integration tests** – Maven automatically picks up test files (typically named `*Test.java`) and runs them.
   Example of a simple unit test in Spring Boot:

```
@Test
public void testServiceLayer() {
    assertEquals(5, someService.calculateSomething());
}
```

These tests ensure that components such as repositories, services, and controllers work properly before deployment.

Step 3: Packaging

6. **Developer runs `mvn package`** – After successful compilation and testing, the developer packages the application into a deployable artifact.
7. **Maven Package creates a JAR/WAR file** – Based on the `pom.xml` configuration, Maven generates either:
    - A **JAR file** (for standalone applications)
    - A **WAR file** (for applications deployed in servlet containers like Tomcat)

Example: After running `mvn package`, the generated file might be:

```
target/myapp-1.0.0.jar
```

Step 4: Deployment

8. **The JAR file is deployed to the server** – The packaged file is then deployed to a server such as:
    - **AWS EC2** (using SSH and SCP commands to transfer and run the JAR)
    - **Kubernetes** (by creating a Docker container and deploying via Helm)
    - **Bare Metal Server** (running `java -jar myapp-1.0.0.jar`)

Example command to deploy on a Linux server:

```
scp target/myapp-1.0.0.jar user@server:/opt/myapp/
ssh user@server "java -jar /opt/myapp/myapp-1.0.0.jar"
```

Step 5: Frontend Build (Angular)

9. **Developer writes Angular code** – Angular components, services, and modules are written in TypeScript.
10. **Angular components call services to fetch data** – A service makes API calls to the backend.
    Example of an Angular service:
    ```
    getData(): Observable<DataModel> {
        return this.http.get<DataModel>('http://backend-api.com/data');
    }
    ```
11. **Developer runs `ng build` to generate the frontend** – This creates a production-ready version of the application.
12. **The `dist` folder is created, containing the compiled Angular app** – The build process optimizes code, removes unnecessary files, and stores everything in the `dist` folder.

Step 6: Connecting Angular to Backend

13. **Angular Service sends an API request, but first, it goes to Spring Security (Keycloak)** – Any request sent from Angular to the backend must go through authentication.
14. **Keycloak checks authentication** – The backend verifies if the user is authenticated by validating a JWT (JSON Web Token).
15. **If authentication is valid, Keycloak returns success** – If valid, Keycloak allows the request to proceed. Otherwise, it rejects unauthorized requests.
16. **The request is then passed to the Spring REST Controller** – The

controller handles the API request, for example:

```
@GetMapping("/data")
public ResponseEntity<Data> getData() {
    return ResponseEntity.ok(dataService.getData());
}
```

17. **The controller calls the Service Layer to process business logic** – This layer contains core logic, such as data processing and validation.
18. **The Service Layer calls the Repository Layer to fetch data** – The service interacts with the database layer via repositories.
19. **Repository queries the Database for requested data** – This uses JPA/Hibernate to fetch records, for example:

```
@Repository
public interface DataRepository extends JpaRepository<Data, Long> { }
```

20. **Database returns the requested data** – The database responds with the queried records.
21. **The data flows back up through the layers** – The repository sends data back to the service layer.
22. **The response reaches the Spring Controller** – The controller prepares the data for an HTTP response.
23. **The response is sent back to the Angular Service** – The API response is returned as JSON.
24. **Angular Component updates the UI with the new data** – The component receives the data and updates the frontend dynamically.
25. **The Developer (User) sees the updated data on the frontend** – The final data is displayed on the web page, completing the request-response cycle.
