**WEEK-05 HANDSON** **SOLUTIONS**

**MICROSERVICES WITH SPRING BOOT 3 AND SPRING CLOUD**

**MICROSERVICES WITH API GATEWAY**

**EXERCISE:**

**CREATING MICROSERVICES FOR ACCOUNT AND LOAN**

In this hands on exercises, we will create two microservices for a bank.

One microservice for handing accounts and one for handling loans.

Each microservice will be a specific independent Spring RESTful Webservice maven project having it's own pom.xml.

The only difference is that, instead of having both account and loan as a single application, it is split into two different applications.

These webservices will be a simple service without any backend connectivity.

Follow steps below to implement the two microservices:

Account Microservice

• Create folder with employee id in D: drive

• Create folder named 'microservices' in the new folder created in previous step. This folder will contain all the sample projects that we will create for learning microservices.

• Open https://start.spring.io/ in browser

• Enter form field values as specified below:

o Group: com.cognizant

o Artifact: account

• Select the following modules o Developer Tools > Spring Boot DevTools o Web > Spring Web

• Click generate and download the zip file

• Extract 'account' folder from the zip and place this folder in the 'microservices' folder created earlier

• Open command prompt in account folder and build using mvn clean package command

• Import this project in Eclipse and implement a controller method for getting account details based on account number.

Refer specification below:

o Method: GET

o Endpoint: /accounts/{number}

o Sample Response. Just a dummy response without any backend connectivity. { number: "00987987973432", type: "savings", balance: 234343 }

• Launch by running the application class and test the service in browser Loan Microservice

• Follow similar steps specified for Account Microservice and implement a service API to get loan account details

o Method: GET o Endpoint: /loans/{number}

o Sample Response.

Just a dummy response without any backend connectivity.

{ number: "H00987987972342", type: "car", loan: 400000, emi: 3258, tenure: 18 }

• Launching this application by having account service already running

• This launch will fail with error that the bind address is already in use

• The reason is that each one of the service is launched with default port number as 8080.

Account service is already using this port and it is not available for loan service.

• Include "server.port" property with value 8081 and try launching the application

• Test the service with 8081 port Now we have two microservices running on different ports.

NOTE: The console window of Eclipse will have both the service console running. To switch between different consoles use the monitor icon within the console view.

**SOLUTION:**

As part of the microservices hands-on exercise,

I implemented two independent Spring Boot RESTful services — one for managing bank account information and another for handling loan details.

The objective was to simulate a microservice-based architecture by creating separate Maven-based applications that expose simple REST APIs without backend connectivity.

Each service was designed to run independently on different ports to reflect real-world service isolation in microservice ecosystems.

To begin with, I set up the workspace by creating a folder named after my employee ID (22501A4401) in the C: drive.

Within it, I created a subfolder called microservices to house both the account and loan projects.

Using https://start.spring.io, I initialized the account microservice by providing com.cognizant as the group and account as the artifact.

The dependencies selected included Spring Web and Spring Boot DevTools, which provided REST support and hot-reloading features.

The generated ZIP file was extracted into the microservices directory and built successfully using mvn clean package from the command line.

After importing the account project into Eclipse as a Maven project,

I implemented a REST controller named AccountController under the package com.cognizant.account.

This controller exposed a simple GET endpoint /accounts/{number} which returned hardcoded JSON data simulating account information.

For instance, accessing http://localhost:8080/accounts/00987987973432 would return account details including account number, type, and balance.

I verified the output using a web browser and confirmed that the service was working correctly.

Next, I followed a similar process to create the loan microservice.

The group was again set as com.cognizant and the artifact as loan, with the same dependencies included.

The project was extracted into the same microservices folder and built using Maven.

Since the default port 8080 was already in use by the account service, I added the property server.port=8081 in the application.properties file of the loan service to assign it a different port.

This change ensured that both microservices could run simultaneously without port conflicts.

In the loan microservice, I created a LoanController under the package com.cognizant.loan.

This controller defined the endpoint /loans/{number} and returned dummy loan data, including loan number, type, amount, EMI, and tenure.

For example, when the URL http://localhost:8081/loans/H00987987972342 was accessed, the service responded with static JSON data representing a car loan.

During development, I faced minor issues such as default port conflicts and initial endpoint mapping errors.

These were quickly resolved by assigning unique ports to each microservice and verifying the correct controller-to-endpoint mappings.

After making these adjustments, both services ran smoothly and responded as expected.

Both services were verified to be running correctly via browser and console logs.

The account microservice was accessible at <http://localhost:8080/accounts/00987987973432>

and the loan microservice at <http://localhost:8081/loans/H00987987972342>

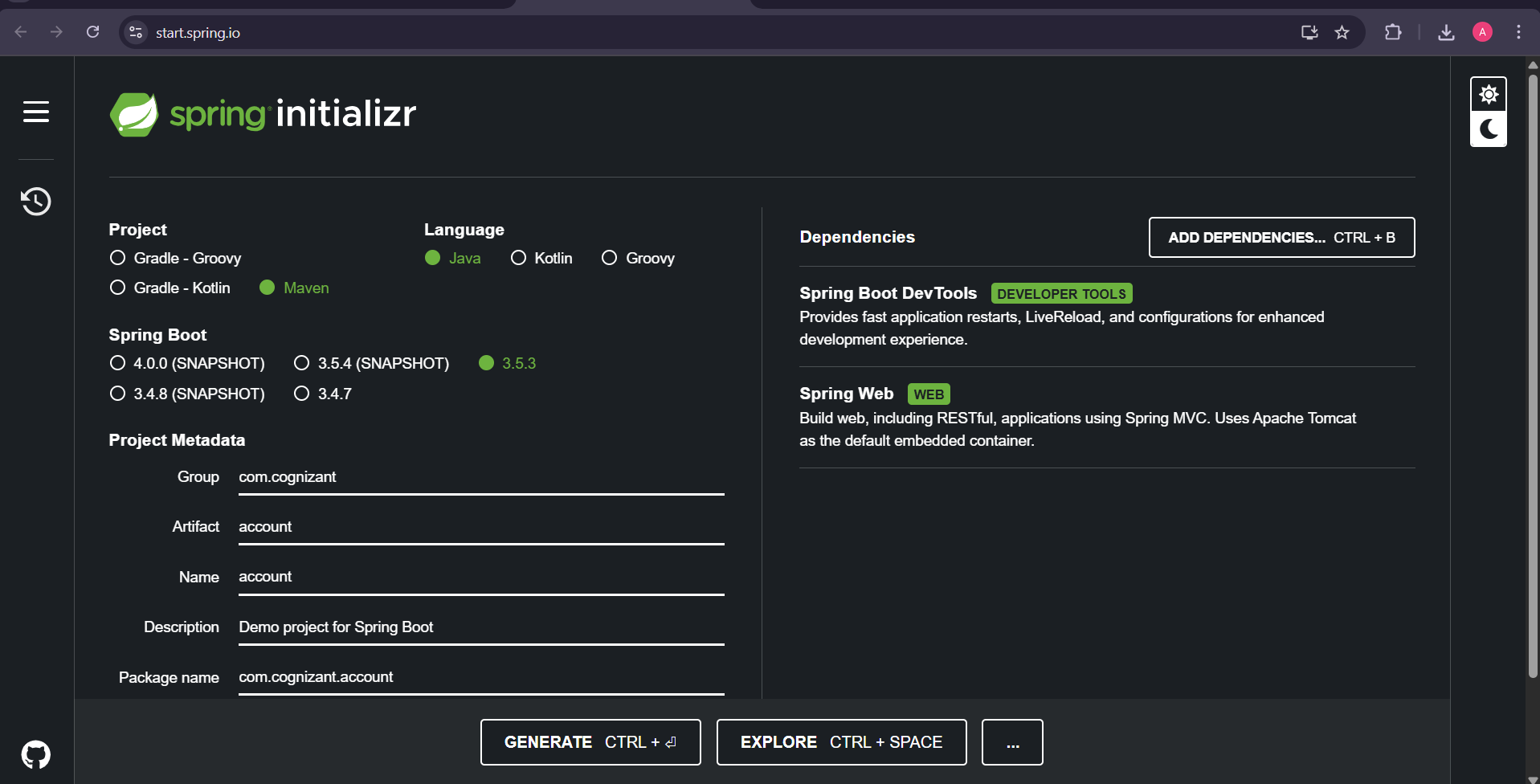
The console logs confirmed that both applications had started on their respective ports and were serving the expected REST endpoints.

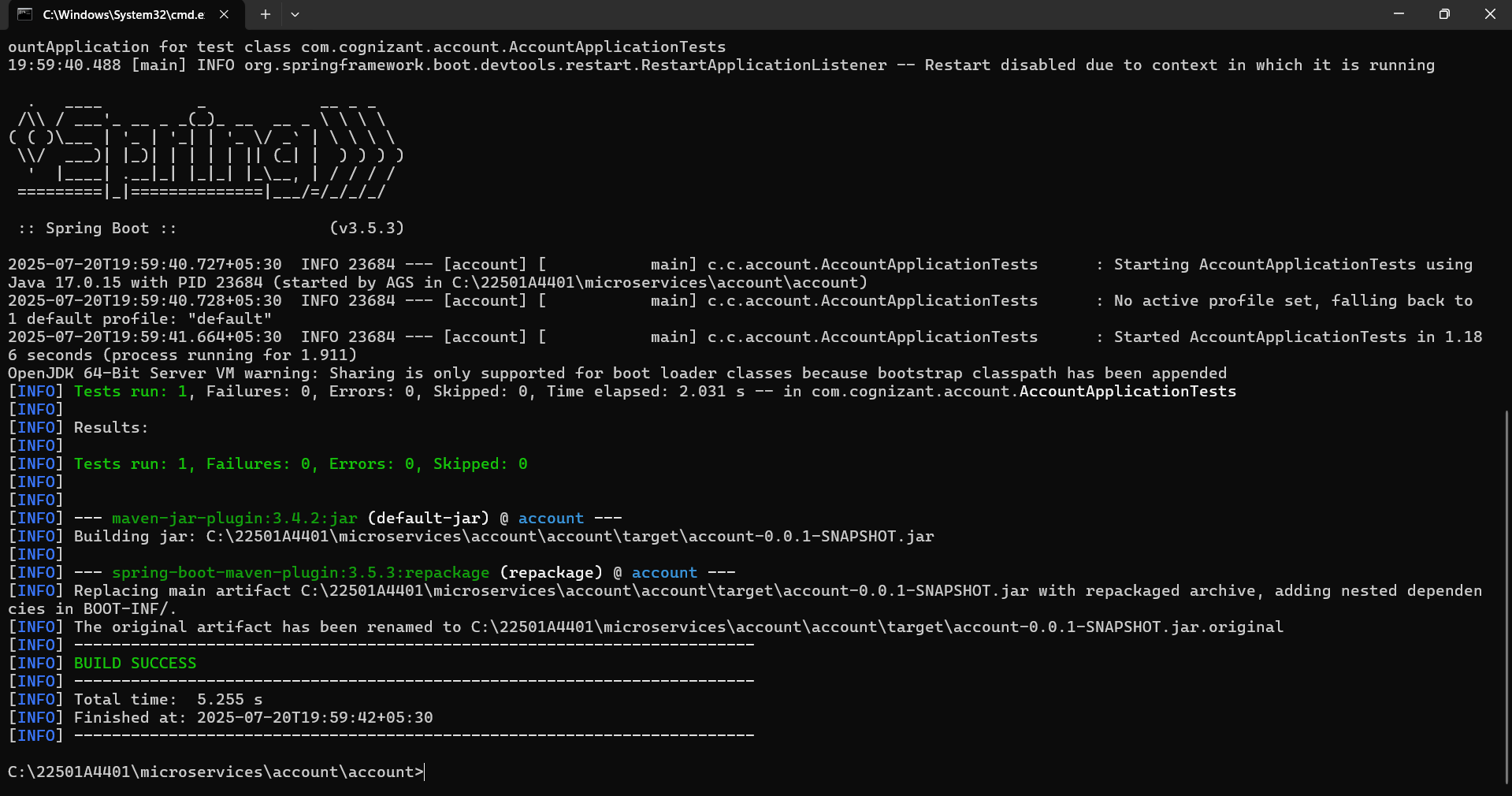
In conclusion, this exercise provided a clear and practical understanding of how to build, structure, and run basic Spring Boot microservices.

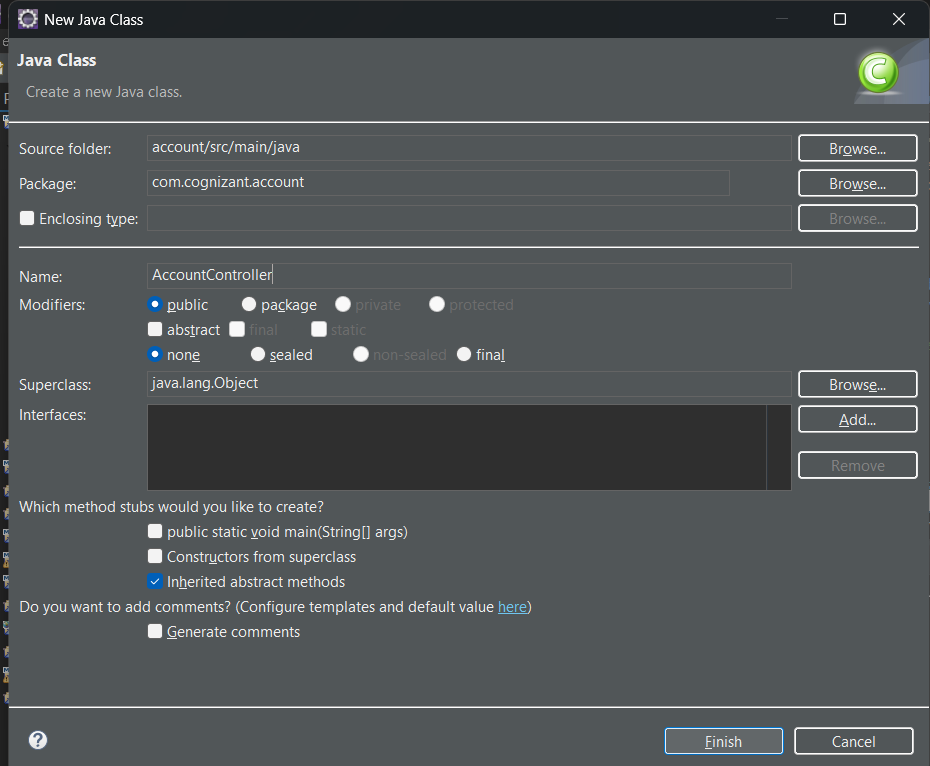
By separating concerns into two services and deploying them independently, I achieved service isolation and gained hands-on experience with real-world microservice principles like individual port configuration, RESTful design, and modular project structure.

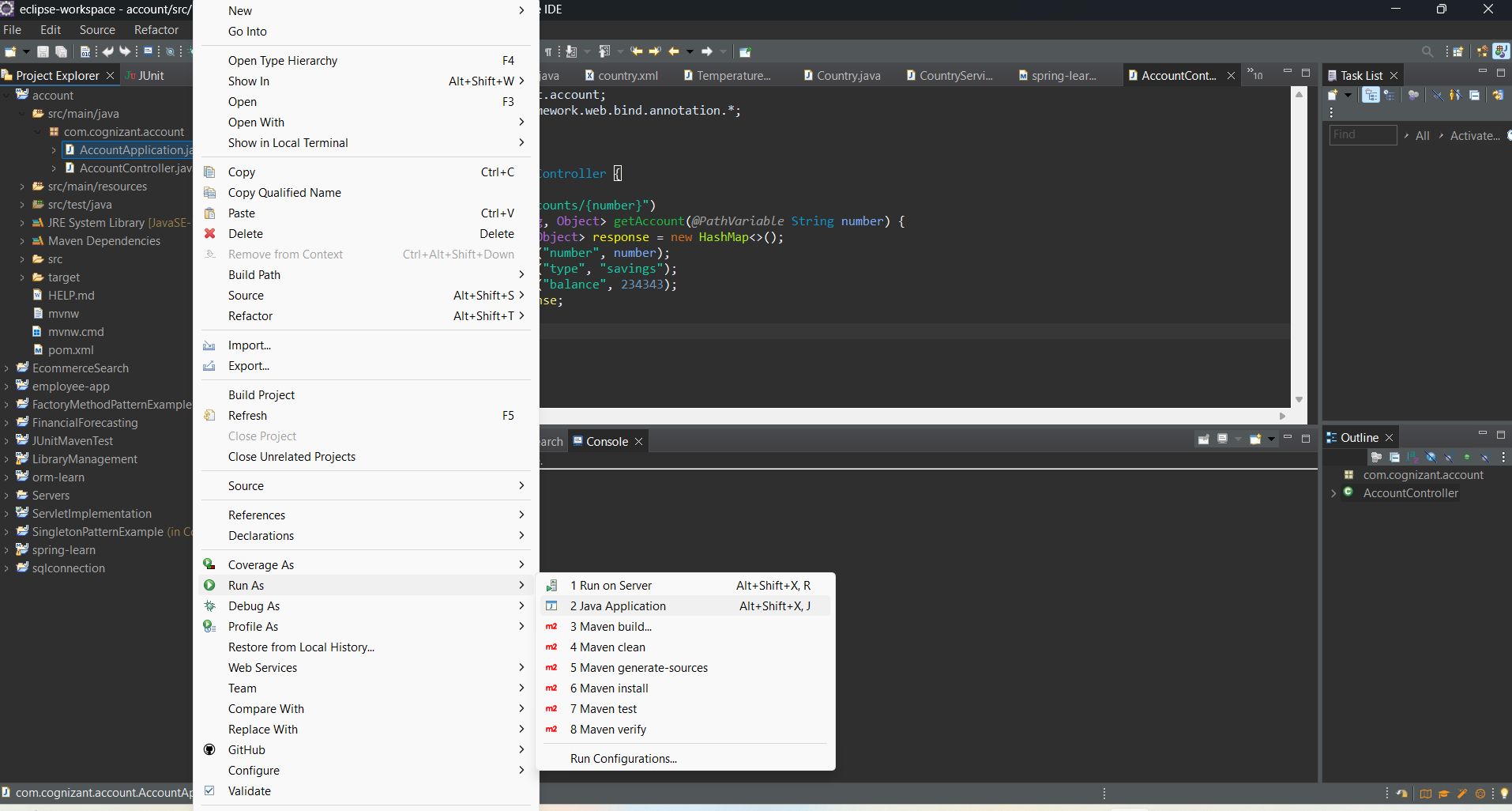
This successful implementation lays a solid foundation for developing more complex, database-integrated microservices in the future.

**IMPLEMENTATION OUTPUTS:**

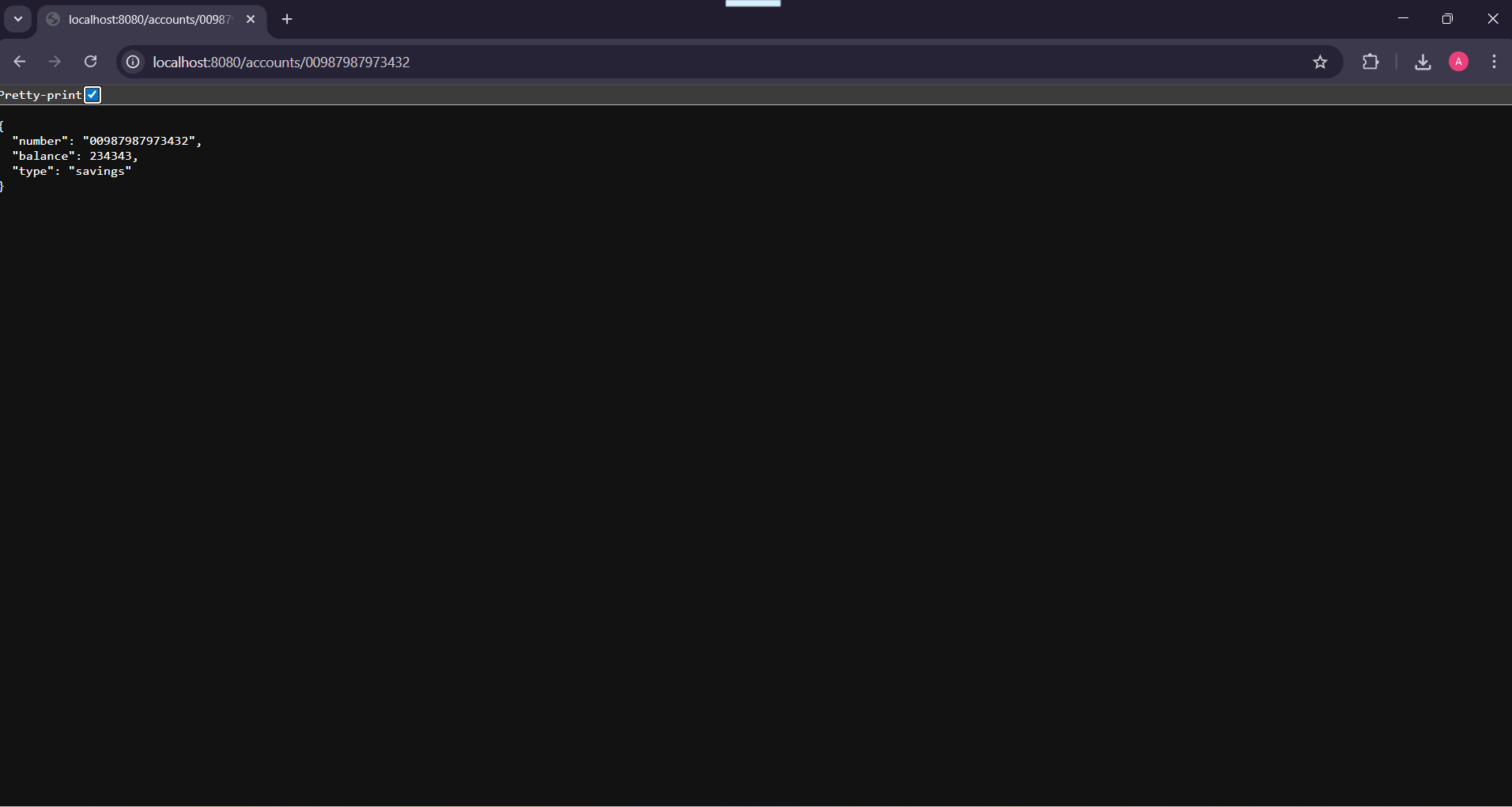




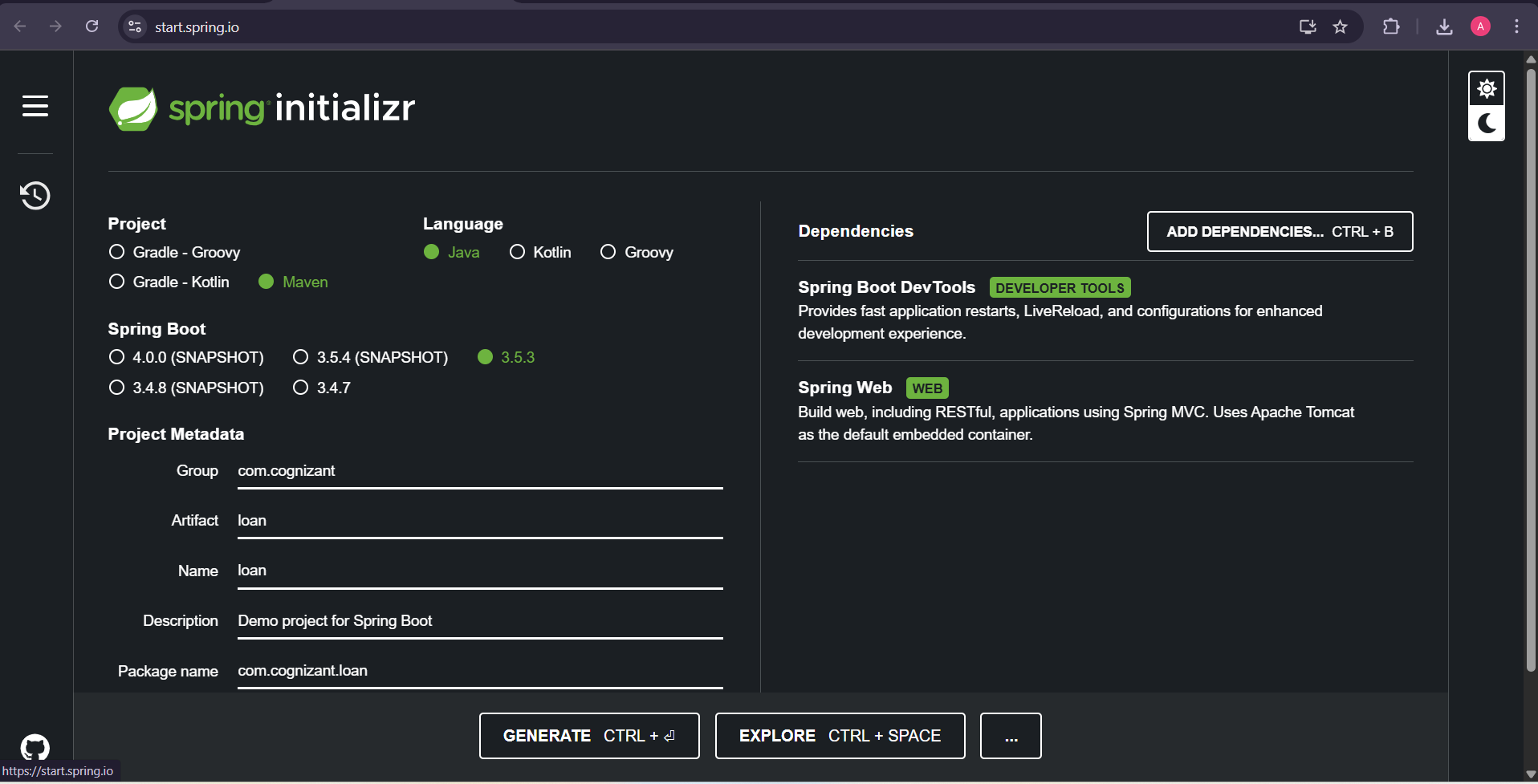


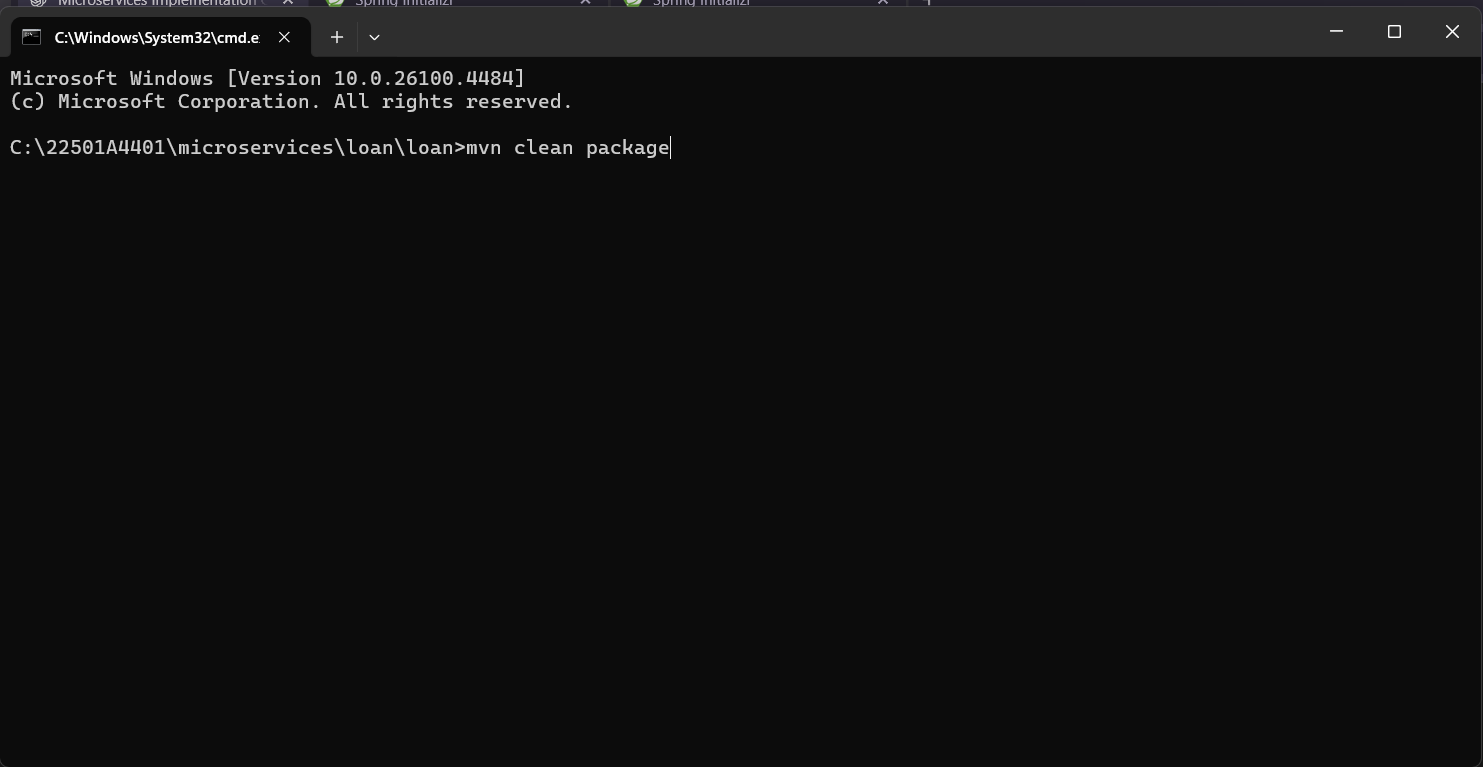


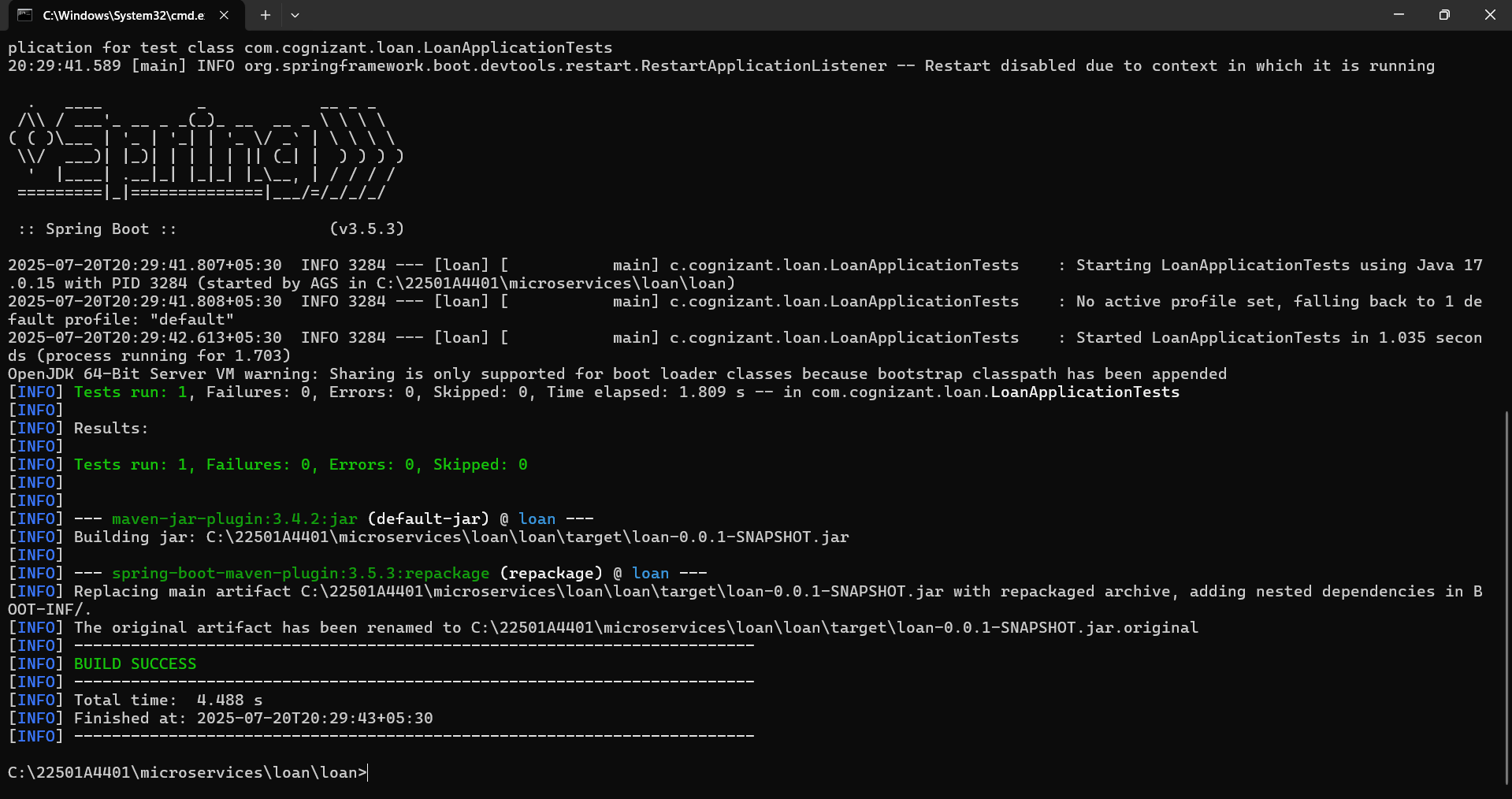
**OUTPUT:**

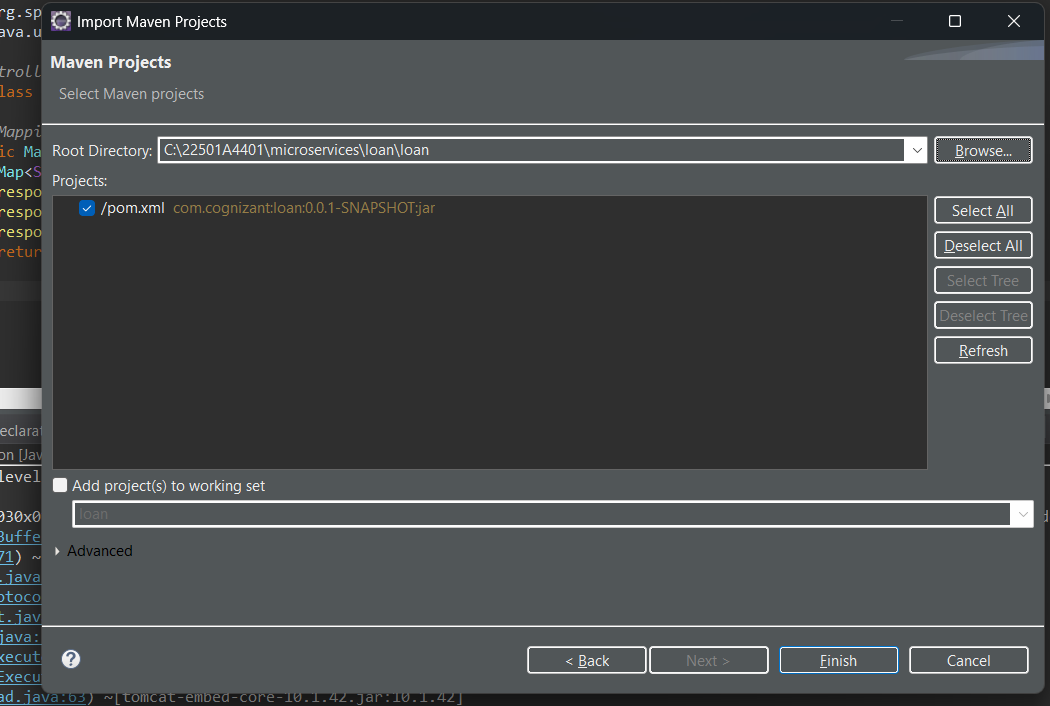


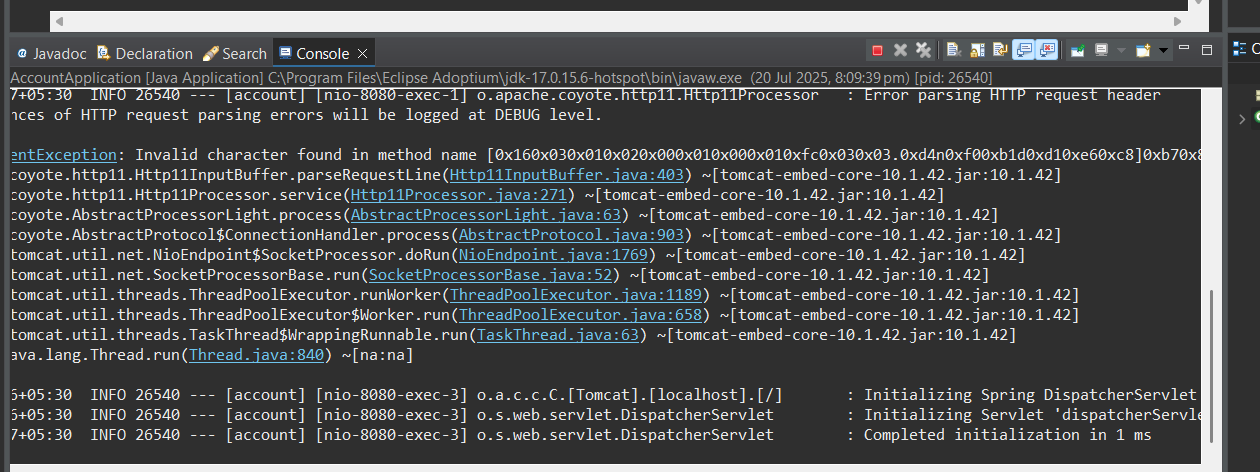
**IMPLEMENTATION OUTPUTS:**

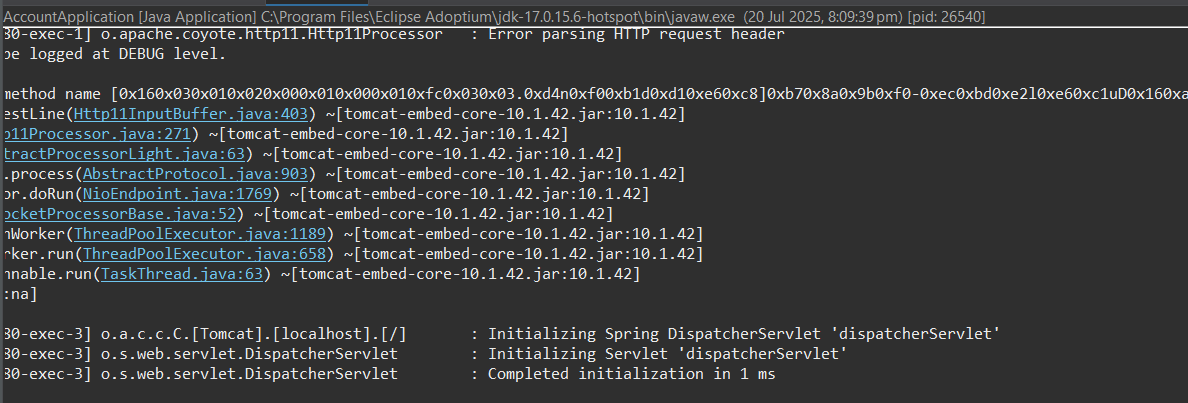


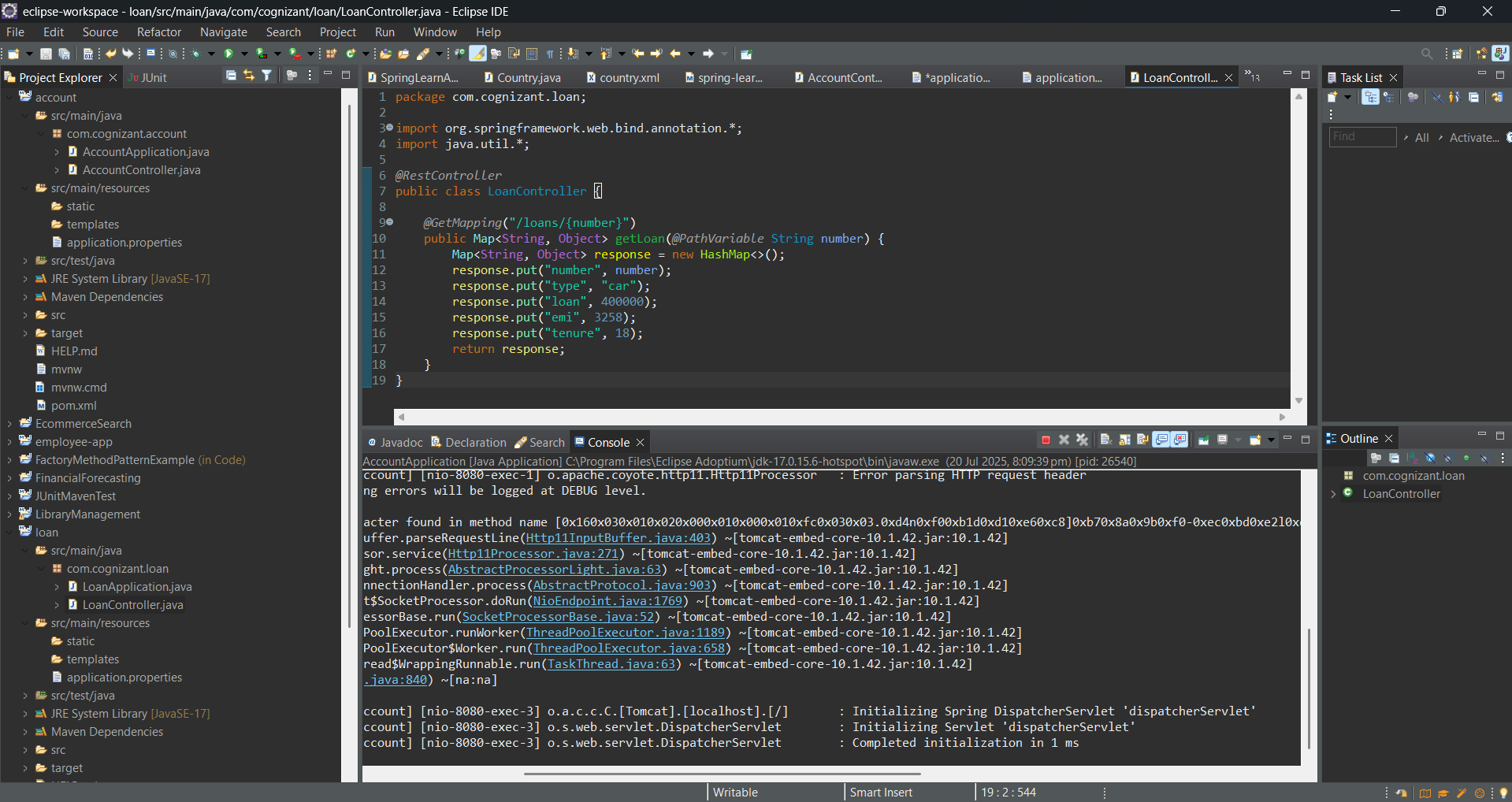


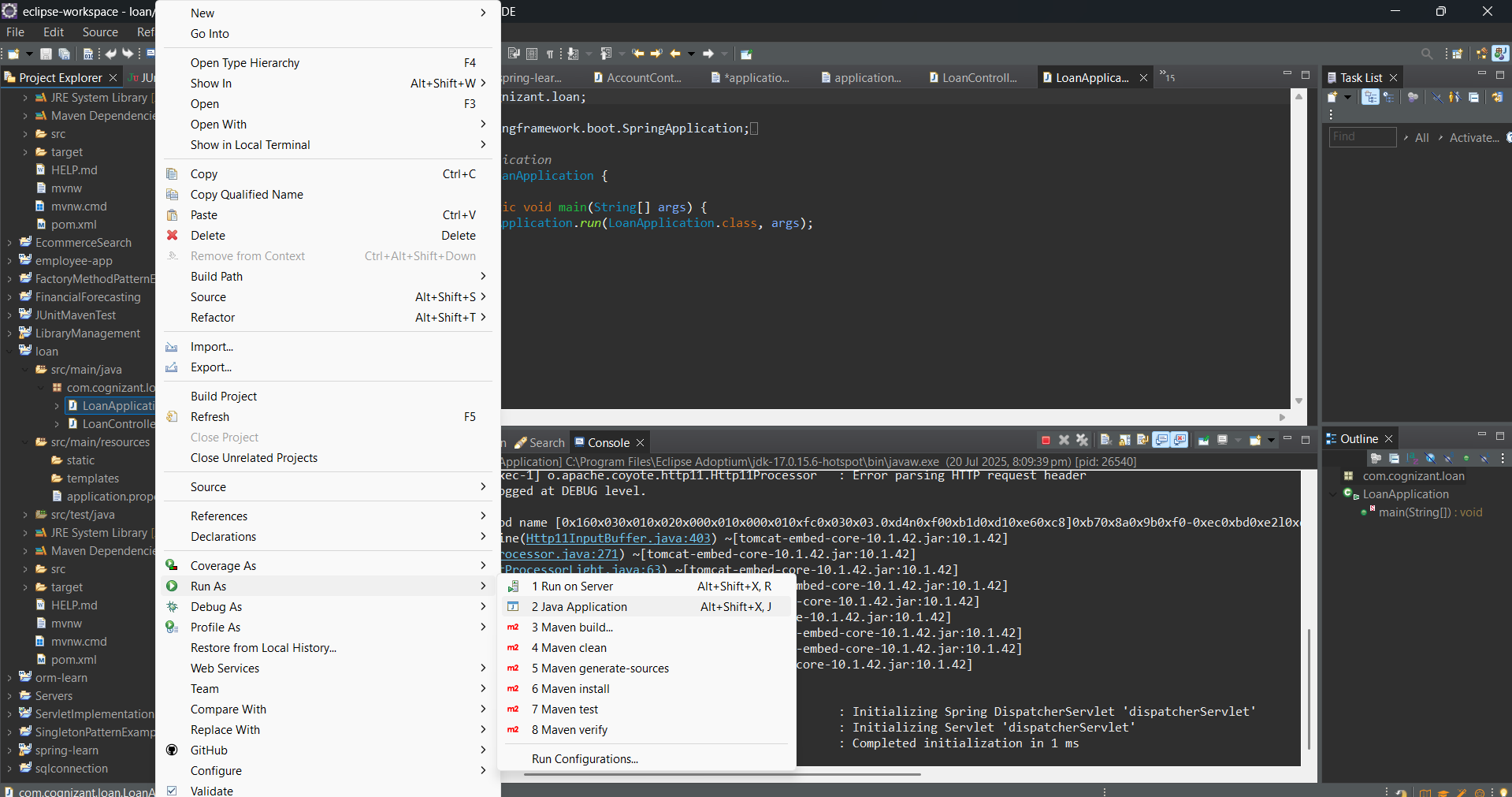


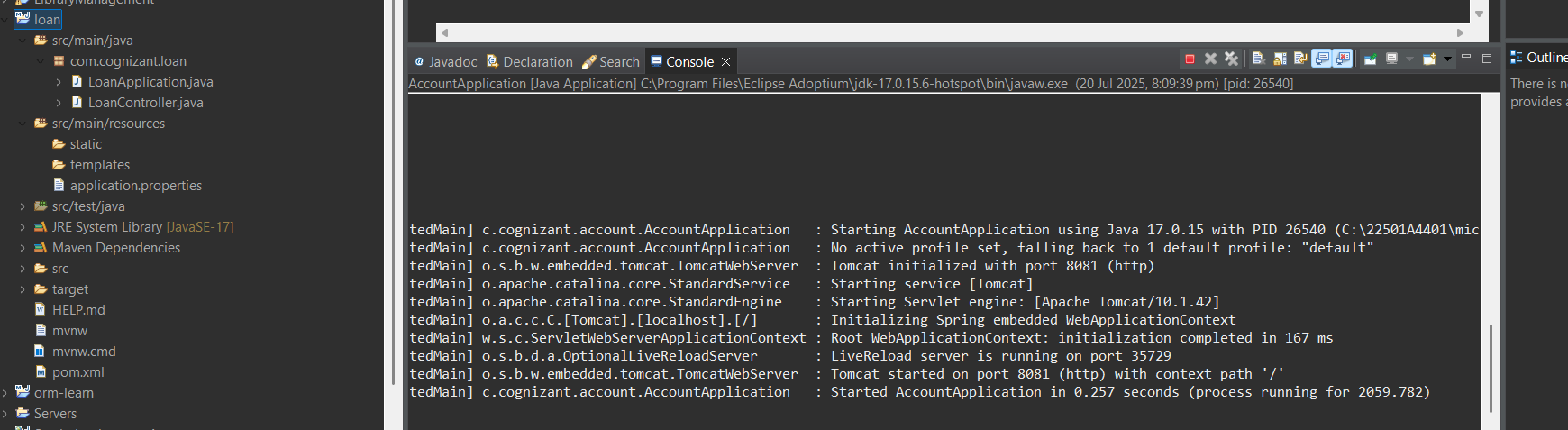












**OUTPUT:**

