**Server Interaction Project**

**Architecture diagram.**

**A diagram of a computer program

Description automatically generated**

**Working**

Explanation:

**Frontend**: This is the AngularJS + NodeJS application responsible for rendering the user interface and handling user interactions. It communicates with the Spring Boot backend using HTTP requests.

This is the main application that accepts user requests, data and communicates with the first server (API). It exposes an endpoint to receive requests for performing a simple operation like multiplying two numbers.

**Spring Boot Backend**: This is the core of the application and is built using Spring Boot. It includes the following components:

This server acts as an intermediary between the client application and the second server (API). It receives requests from the client, processes them, and forwards them to the second server for computation. It also receives responses from the second server and sends them back to the client.

Controllers: Handle incoming HTTP requests and delegate processing to services.

Services: Implement business logic, interact with the Second Server API, and handle data access through the repository.

Repository: Interact with the MySQL Database to store and retrieve data.

**Second Server API**: This is the second server, also implemented using Spring Boot. It exposes a RESTful API to perform simple operations like multiplying two numbers.

This server handles the actual computation, in this case, the multiplication of two numbers. It receives requests from the first server, performs the calculation, and sends the result back to the first server.

**Oracle Database**: The backend uses an MYSQL database to store calculated results.

The frontend communicates with the Spring Boot backend, which, in turn, communicates with the Second Server API and MySQL Database to perform the required operations and store/retrieve data.

A screenshot of a computer

Description automatically generated

**Exception Handling**:

Custom exceptions is included to provide more meaningful and context-specific information to the client and handle errors in a more structured manner.

A screenshot of a computer

Description automatically generated

**Logging:** The logging frameworks like SLF4J is used in backend applications to log important events, errors, and debug information.

**Data persistence:** Spring Data JPA is used to save the calculated numbers in the database like MYSQL.

**POSTMAN:** Postman is used to test the end-points.

**A screenshot of a computer

Description automatically generated**

**API Documentation:** API Documentation is done using tools like Swagger or Springfox to generate API documentation for the application.

**A screenshot of a computer

Description automatically generated**

**Design Pattern:**

**Factory Method Patter:** The Factory Method pattern is used to create instances of the CalculationService class. Instead of directly instantiating the CalculationService in the controller, we can define a factory method that abstracts the instantiation logic and provides instances of the service. This can make the code more flexible and easier to modify if we want to change the service implementation in the future.

**Repository Pattern**: This pattern separates the data access logic from the rest of the application, making it easier to switch between different data storage solutions (e.g., Oracle, MySQL, etc.) without changing the application's core logic.

**Deployment Strategy:**

1. Build the Project:

We can use a build tool like Apache Maven to build Spring Boot project. This will create an executable JAR file containing all the necessary dependencies.

2. Choose a Hosting Environment:

Decide on a hosting environment for the Server Integration application. There are several options, including traditional servers, cloud platforms like AWS, Azure, or Google Cloud, or container orchestration platforms like Kubernetes.

3. Containerization (Optional):

Containerizing of Spring Boot application using Docker. Containerization allows to package the application and its dependencies into a container, ensuring consistency across different environments. This also facilitates easy deployment and scaling.

4. Configuration Management:

Externalize the application configuration from the code using environment-specific properties files or environment variables. This allows to change the configuration without modifying the code.

5. Database Setup:

Ensure that the MYSQL database is set up and accessible from the deployed environment. Use appropriate database configuration properties in the application for connection.

6. Choose Deployment Method:

Depending on the hosting environment, choose an appropriate deployment method:

- Traditional Server: Copy the JAR file to the server and use a script or system service to run the application.

- Cloud Platform: Deploy the JAR file to the cloud platform and configure auto-scaling and load balancing as needed.

- Docker & Kubernetes: Create a Docker image from the JAR file and deploy it to a Kubernetes cluster

7. Security:

Ensure that the application is secure by implementing best practices like HTTPS, input validation, and proper access controls.

8. Continuous Integration & Continuous Deployment (CI/CD):

Set up a CI/CD pipeline to automate the build, testing, and deployment processes. Tools like Jenkins, GitLab CI/CD, or GitHub Actions can be used for this purpose.

9. Test in Staging Environment:

Before deploying to the production environment, test the application in a staging or test environment to ensure everything works as expected.

10. Production Deployment:

Finally, deploy the application to the production environment after thorough testing and verification. Regularly update the application and its dependencies to keep it secure and up-to-date.