# Milestone 4

Software design techniques

## Team members:

1. FINICHIU Eduard - Adelin
2. JERCĂU Hadasa - Ștefana

## Requirements for grade 10:

I. Three or more well-defined and independent services are implemented.
II. Inter-service communication is robust and efficiently implemented.
III. A comprehensive Postman collection covers all functionalities and edge cases.
IV. The Docker setup is clean, efficient, and the instructions in the Readme.md are clear and easy to follow.

### Proof of fulfillment for Requirement I

1. **Six distinct services** that demonstrate clear separation of concerns (a core tenet of microservices)
2. **Three functional services** that  operate independently regarding their database access (using separate H2 instances)
   a. User Service
   b. Job Service
   c. Bidding Service operate independently regarding their database access (using separate H2 instances)
3. **Well-defined and independent:** User Service owns identity, the Job Service owns job data, and the Bidding Service owns the proposals

### Proof of fulfillment for Requirement II

1. **Communication** is implemented **using** the standard **Spring Cloud** patterns essential for a scalable microservices architecture.
2. We utilized **Feign Clients** for synchronous communication (e.g., Job Service calling User Service to verify roles) **combined with Eureka Service Discovery** for dynamic lookups. This setup means services only need to know the name of the recipient (USER-SERVICE) rather than its specific IP address or port, ensuring the system is resilient to service location changes. Crucially, the solution correctly identified and

fixed the **Docker** networking issues (using internal container names instead of localhost), providing a robust communication layer.

## Proof of fulfillment for Requirement III

1. Postman Collection meets the standard for submission by covering the full API contract.
2. We successfully defined requests for all key **functionalities** (Create User/Job/Bid, Get All Users/Jobs) and implemented **Edge Cases** by intentionally triggering validation failures.

## Proof of fulfillment for Requirement IV

1. The **Docker deployment is professional and complete**. We successfully containerized all six components using separate Dockerfiles and orchestrated them efficiently using docker-compose.yml.
2. The finalization of the README.md with the single docker-compose up command provides a clear, efficient, and professional instructional guide.

# Walkthrough

## Services planned:

- [x] **api-gateway** (Entry point)
- [x] **discovery-service** (Essential for the gateway to find other services)
- [x] **user-service**
- [x] **bidding-service**
- [x] **job-service**
- [x] **review-service**
- [ ] **notification-service**
- [ ] **payment-service**

## About Config Server:

In a microservices environment, managing configurations (database credentials, port numbers, internal API URLs) across 6+ services becomes difficult and risky. The Config Server solves this by centralizing all settings in a Git repository.

## Development phase data related to ports used:

1. **Run Discovery Server (Eureka):** Should run on port **8761**.
2. **Run Config Server:** Should run on port **8888**.
3. **Run User Service:** Should run on port **8081**.
4. **Run API Gateway:** Should run on port **9090**.

## Observation about Job Service:

A Client cannot post a Job if the Client doesn't exist. The Job Service must "call" the User Service to validate the clientId. We will use Spring Cloud OpenFeign for this.

Usefull terminal commands for **docker-compose.yml** :
normal build:  docker-compose up --build
stop containers: docker-compose down
forced fresh start, forced clean build: docker-compose up --build --force-recreate
list containers processes: docker ps

needed due to race condition between containers:
docker-compose restart api-gateway

to get logs if anything bad happens::
docker-compose logs --tail=20 api-gateway

# Walkthrough

1. open Docker Desktop App

2. use **docker-compose down** in the project root to ensure all containers and the network are removed cleanly

3. in Terminal / Powershell go to root directory (Code-Microservices-Implementation)

4. **docker-compose up --build** or in case you want to rebuild them “docker-compose up --build --force-recreate”

5. Check **http://localhost:8761/**

6. http://localhost:8888/api-gateway/default Goal: Prove that the Gateway is using centrally managed configuration for its routing table (rather than hardcoded local files).

7. check **docker ps** if needed (see restarts, errors or runtime)

8. docker-compose restart api-gateway (to combate possible race condition)

## Functional Demo (The Postman Collection)

We run **Postman** in the following order to demonstrate the successful end-to-end flow.

- **A. Setup Data & Factory Pattern:**
  - Run **"POST Create Client"**. (Creates User 1 - Role: CLIENT).
  - Run **"POST Create Freelancer"**. (Creates User 2 - Role: FREELANCER).
  - *Goal:* Show the **Builder** and **Factory Method** patterns working by inspecting the resulting JSON bodies.
  - Run **"GET All Users"**.

- **B. Inter-Service Validation & Job Creation:**
  - Run **"POST Create Job"** (Requires Client ID 1).
  - *Goal:* Demonstrate the **Feign Client** working, as the **Job Service** must contact the **User Service** to validate the role.
  - Run **"GET Job by ID"**.

- **C. Triple Validation & Bidding Logic:**
  - Run **"POST Place Bid"**. (Requires Job ID 1 and Freelancer ID 2).
  - *Goal:* Demonstrate the **Bidding Service** successfully contacting *both* the Job and User services before saving the transaction.

- **D. Edge Case Test (Resilience/Validation):**
  - Run **"Edge Case"** requests for Job and Bid. (Attempts to post a job using Freelancer ID 2 / Attempts to bid using wrong data).
  - *Goal:* Demonstrate the **Application Logic** is functional by verifying the test fails with the expected **500 Internal Server Error**

Return to the terminal window running Docker Compose and press Ctrl + C.

Run: docker-compose down to cleanly stop and remove the containers.