Objective

Extend the Kafka RPC demo to create a **User Management Microservice** that handles CRUD operations via Kafka messaging.

Requirements

Part 1: Backend Service (5 points)

Modify consumer.js to implement a user management service that handles these operations:

1. **CREATE_USER** - Create a new user
    - Input: { operation: "CREATE_USER", name: string, email: string, age: number }
    - Output: { success: true, userId: string, message: "User created" }
2. **GET_USER** - Retrieve user by ID
    - Input: { operation: "GET_USER", userId: string }
    - Output: { success: true, user: {userId, name, email, age} }
3. **UPDATE_USER** - Update user information
    - Input: { operation: "UPDATE_USER", userId: string, updates: {name?, email?, age?} }
    - Output: { success: true, user: {updated user data} }
4. **DELETE_USER** - Remove a user
    - Input: { operation: "DELETE_USER", userId: string }
    - Output: { success: true, message: "User deleted" }
5. **LIST_USERS** - Get all users
    - Input: { operation: "LIST_USERS" }
    - Output: { success: true, users: [...], count: number }

Requirements:

- Store users in-memory (use a JavaScript object/Map)
- Generate unique user IDs (use crypto or UUID)
- Validate input data (email format, age > 0, etc.)
- Return proper error messages for invalid operations

Part 2: Client Demo (3 points)

Create a new file homework-demo.js that demonstrates ALL 5 operations in sequence:

SUBMISSION REQUIREMENTS (SCREENSHOTS ONLY)
Submit 5-6 screenshots showing:
Screenshot 1:

Docker Containers Running - Command: docker ps - Shows Kafka and Zookeeper containers active

Screenshot 2:
Consumer Service Started - Terminal showing npm run start: consumer - Shows "Consumer is ready and listening" message

Screenshot 3:
Demo Execution - Part 1 - Terminal showing node homework-demo.js - Shows CREATE operations (3 users) and first LIST operation

Screenshot 4:
Demo Execution - Part 2 - Same terminal (scrolled down) - Shows GET, UPDATE, DELETE operations

Screenshot 5:
Demo Execution - Part 3 - Same terminal (scrolled down) - Shows final LIST operation and error handling demonstrations

Part 2: 3-agent mini-system with kafka + langchain

Build a tiny system with 3 agents that talk through Kafka:

1. **Planner** : makes a plan for the question
2. **Writer** : writes a short answer with LangChain
3. **Reviewer** : checks the answer and approves it

All messages go through Kafka topics. No agent calls another agent directly.

You can use already existing agents from your previous agents

what you will build

you : (send a question) : topic: inbox

Planner : reads inbox, sends a TODO : topic: tasks

Writer : reads tasks, writes a draft : topic: drafts

Reviewer : reads drafts, approves : topic: final

You'll run each agent in its own terminal.

create the Kafka topics

Create four topics: inbox, tasks, drafts, final.
(Use your Kafka CLI or UI; example CLI commands:)

```
kafka-topics --create --topic inbox  --partitions 1 --replication-factor 1 --bootstrap-server localhost:9092
kafka-topics --create --topic tasks  --partitions 1 --replication-factor 1 --bootstrap-server localhost:9092
kafka-topics --create --topic drafts --partitions 1 --replication-factor 1 --bootstrap-server localhost:9092
kafka-topics --create --topic final  --partitions 1 --replication-factor 1 --bootstrap-server localhost:9092
```

run each agent in their own terminal

```
# terminal 1
python planner.py

# terminal 2
python writer.py
```

```
# terminal 3
python reviewer.py

# terminal 4 (send one question)
python send_question.py
```

Now read the final result (pick one):

You should see a JSON with "status": "approved" and an "answer".

what to submit

- Screenshot or log lines showing each agent received and sent messages
- Screenshot of the message in topic **final** with "status": "approved"
- A short report explaining how Kafka helped your agents coordinate