Assignment 3: Client-Server Posting System

Objective

You will build a basic client-server posting system with Node.js, jQuery, and jQuery UI. The system will:

1. Accept and store **posts** (each with a unique ID).
2. Accept and store **responses** (each with a unique ID, associated with a post ID).
3. Serve the data to a frontend that displays and updates posts/responses asynchronously.
4. Be tested for functionality **and** performance (load testing) using **npm loadtest**.

Technology Stack

- **Backend**: Node.js
- **Frontend**: HTML, **jQuery**, and **jQuery UI**
- **Communication**: AJAX calls (using jQuery) to your Node.js server
- **Containerization**: Docker (`Dockerfile` and/or `docker-compose.yml`)

Project Structure

Submit a **single, compressed archive** (`.zip`) containing:

1. **docker-compose.yml** (and **Dockerfile** if needed)
2. **server.js**
3. **posting.html**
4. **report.pdf**

Part A: Node.js Backend (40 Points)

1. **In-Memory Arrays**
    - Maintain two arrays in memory:
        - `posts`: each post has `{ id, topic, data, timestamp }`

        - `responses`: each response has `{ id, postId, data, timestamp }`
2. **Endpoints**
    - **POST /postmessage**
        - Accepts `topic` and `data`.
        - Creates and stores a new **post** in the `posts` array.
        - Returns `{ success: true, id: newPostId }`.
    - **POST /postresponse**
        - Accepts `postId` and `data`.
        - Creates and stores a new **response** in the `responses` array.
        - Returns `{ success: true, id: newResponseId }`.
    - Optionally, **GET /alldata** (or similar) to retrieve both `posts` and `responses`.
3. **Allowed Modules**
    - `express`, `body-parser`
    - Standard Node.js built-ins (e.g., `fs`, `Date`)
    - **npm loadtest** (see Part C) allowed for **testing only**; no other external libraries for functionality.

Part B: HTML Frontend (50 Points)

1. **posting.html**
    - Must use **jQuery** (for AJAX and DOM manipulation) and **jQuery UI** (at least one UI component).
    - Display all **posts** (and their **responses**) fetched from your server (e.g., via `GET /alldata`).
    - Provide a form (or jQuery UI dialog) to create new posts:
        - Send `POST /postmessage` with `topic` and `data`.
        - Dynamically update the list of posts upon success.
    - For each post, allow creating a **response**:
        - Send `POST /postresponse` with `postId` and `data`.
        - Dynamically update the responses for that post upon success.
    - **No full-page reload**—updates must be asynchronous via jQuery AJAX.
2. **jQuery UI**
    - Incorporate at least one jQuery UI widget (e.g., dialog, accordion, datepicker, etc.) to enhance the interface.

Part C: Test Report (10 Points)

1. **Overall Testing Approach**
    - How you tested your Node.js backend (endpoints, data structure, error handling).
    - How you tested your frontend (creation, display, updating of posts/responses).
2. **Functional Test Cases**
    - Normal inputs, edge cases (invalid data, missing fields, invalid `postId`, etc.).
    - Expected vs. actual results, with screenshots or logs.
3. **Load Testing with npm loadtest**
    - Install **loadtest** (e.g., `npm install -g loadtest` or `npx loadtest`).
    - Run load tests (at least one) against your server.
    - In **report.pdf**, include:
        - The command(s) used (e.g., `loadtest --concurrency=10 --requests=1000 http://localhost:3000/postmessage`).
        - Results (requests per second, response times, any errors).
        - Brief analysis of server performance under load.
4. **Challenges and Solutions**
    - Document any issues you faced and how you solved them.

Submission Requirements

1. **docker-compose.yml** (and **Dockerfile** if needed)
    - Application must run with `docker compose up`.
    - Accessible at http://localhost:3000.
2. **server.js**
    - Node.js server code (with in-memory arrays, endpoints).
3. **posting.html**
    - Must use jQuery + jQuery UI for UI elements and AJAX requests.
4. **report.pdf**
    - Must include load test details (commands, results, analysis) plus functional testing coverage.

Grading Matrix (Total 100 Points)

Requirement	Points
Part A: Node.js Backend (40 Points)	
1. **POST /postmessage** endpoint	10 points

2. **POST /postresponse** endpoint	10 points
3. **Data Structures in Memory** (unique IDs, timestamps, in-memory arrays for posts/responses)	10 points
4. **Error Handling & Validation** (invalid postId, missing fields, etc.)	5 points
5. **Docker Configuration for Backend** (correct port, runs in container)	5 points
Part B: HTML + jQuery/jQuery UI Frontend (50 Points)	
1. **Displaying Existing Posts/Responses** (fetched via AJAX, rendered in UI)	10 points
2. **Creating New Posts** (async form submission, auto-update UI)	10 points
3. **Creating New Responses** (async form submission per post, auto-update UI)	10 points
4. **Use of jQuery UI Component(s)** (e.g. dialog, accordion, etc.)	10 points
5. **Overall User Experience & Code Clarity** (readability, DOM updates, styling, etc.)	10 points
Part C: Test Report (10 Points)	
1. **Test Approach & Functional Test Cases** (screenshots/logs, thoroughness)	3 points
2. **Load Testing with npm loadtest** (commands, results, analysis)	3 points
3. **Clarity & Presentation** (challenges, solutions, well-structured report)	4 points
Total	**100**

Important Notes

- Use only the allowed modules on the server side (`express`, `body-parser`, Node built-ins).
- **npm loadtest** is allowed **only** for testing purposes.
- Thoroughly test your solution under various conditions, including concurrency and load.
- Failure to provide working Docker files/configuration will result in **0 points** for the entire assignment.