Assignment 5: Client-Server Posting System with CouchDB & React

Objective

You will build a basic client-server posting system using **Node.js**, **React**, and **CouchDB** (via the nano library). The system will:

1. **Accept and store posts**: Each post must have a unique ID, a topic, data, and a timestamp.
2. **Accept and store responses**:
    - Users should be able to create responses to posts **and** responses to existing responses.
    - Each response must have a unique ID, a reference to its parent post or response (e.g., `parentId`), contain data, and include a timestamp.
3. **Serve data to a frontend**: The data should be fetched and updated asynchronously so that all users see new posts and responses.
4. **Undergo load testing**: The system will be tested for functionality and performance (load testing) using `npm loadtest`.

Technology Stack

- **Backend**: Node.js with CouchDB integration using the nano module
- **Frontend**: React (created via `create-react-app` or a similar setup)
- **Communication**: Fetch or Axios calls from React to your Node.js server
- **Containerization**: Docker (`Dockerfile` and/or `docker-compose.yml`)

Database Schema / Document Design in CouchDB

In CouchDB, data is stored as JSON documents. You may store **posts** and **responses** in separate databases or use a single database (e.g., `postsdb`) to differentiate documents by type (e.g., `_type: 'post'` vs. `_type: 'response'`). A possible approach:

- **posts** (doc type):
    - `_id` (unique identifier generated by CouchDB or your application)
    - `_type: 'post'`
    - `topic`
    - `data`
    - `timestamp`
- **responses** (doc type):
    - `_id`
    - `_type: 'response'`
    - `parentId` (could reference a **post** `_id` or another **response** `_id`)
    - `data`
    - `timestamp`

  Note: CouchDB will auto-generate `_id` if not provided. You can also create your own naming scheme for IDs (e.g., `post:UUID`, `response:UUID`).

Project Structure

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

- `docker-compose.yml` (and `Dockerfile` if needed; ensure both the Node.js server and CouchDB container are configured)
- `server.js`
- `frontend/` (React app source code, which includes `package.json` for the React project)
- `package.json` (for the Node.js server)
- `report.pdf`

  Note: You may have two `package.json` files—one for your Node.js backend and one for your React frontend. Organize them clearly.

Part A: Node.js Backend with CouchDB (Nano) Integration (40 Points)

1. Database Connection

1. Install `nano` (`npm install nano`) and configure a connection to your CouchDB instance.

2. Use environment variables for CouchDB credentials (e.g., username, password, host, port).

Verify the connection works by creating or using an existing database, for example:

js
Copy

```
const nano = require('nano')(process.env.COUCHDB_URL);
const db = nano.db.use('postsdb');
```

3.

2. Endpoints

`POST /postmessage`

- **Input**: Accepts `topic` and `data` from the client.
- **Process**: Inserts a new document in the CouchDB database with:
  - `_type: 'post'`
  - `_id` (either auto-generated by CouchDB or your own)
  - the provided `topic`, `data`
  - a `timestamp` (e.g., `Date.now()` or `new Date()`)
- **Output**: Returns `{ success: true, id: newPostId }`.

`POST /postresponse`

- **Input**: Accepts `parentId` and `data`.
  - `parentId` can reference either a **post** or another **response**.
- **Process**: Inserts a new document with:
  - `_type: 'response'`
  - `parentId`
  - `data`
  - a `timestamp`
- **Output**: Returns `{ success: true, id: newResponseId }`.

(Optional) `GET /alldata`

- **Purpose**: Retrieve all posts and their corresponding responses (including nested responses).
- **Implementation**:
    - You can fetch all docs and separate them by `_type`.
    - Or create a CouchDB view to query posts and responses more efficiently.

3. Error Handling & Validation

- **Validate Input**: Check that required fields (e.g., `topic`, `data`, `parentId`) are provided where needed.
- **Handle Errors**: Catch CouchDB connection/query errors and return appropriate HTTP responses.

4. Performance Requirement

- While there is **no strict 300ms latency requirement**, your system should still aim to be responsive.
- Use efficient queries, indexing, or views if needed for faster lookups.

5. Allowed Modules

- **express**
- **body-parser** (or built-in Express JSON middleware)
- **nano** (for CouchDB)
- Standard Node.js built-ins (e.g., `fs`, `Date`)

Part B: React Frontend (50 Points)

Create a React application (e.g., via `create-react-app`), ensuring it communicates with the Node.js server via REST calls (Axios or Fetch).

1. Display Data

- **Fetch and Display** all posts and corresponding responses (potentially nested).
- You can structure the data in a threaded view if you want to show nesting levels.

2. Create New Posts

- **Form**: Provide a form in React to submit a new post (`topic` and `data`).
- **Request**: On form submission, send a `POST /postmessage`.
- **UI Update**: After success, refresh or update the list of posts without a full-page reload.

3. Create New Responses

- **For posts**: Provide a button or link to add a response, which references the post's `_id`.
- **For responses**: Provide a button or link to add a response, which references the **response** `_id`.
- **Request**: `POST /postresponse` with the `parentId` (the post's or response's ID) and `data`.
- **UI Update**: Automatically display the new response in the correct nested position.

4. User Interface Enhancements

- Use a React component library (e.g., React Bootstrap, Material UI) or your own custom styling.
- Consider a **modal** or **dialog** for creating new posts or responses for a polished UX.

5. Asynchronous Updates

- Use **polling** (setInterval) or other techniques to keep data fresh, or implement a manual "Refresh" button.
- Changes should appear to all users eventually, but no strict 300ms rule is required.

Part C: Test Report (10 Points)

1. Overall Testing Approach

- **Backend Testing**:
  - Verify Node.js endpoints, CouchDB integration, and document structure.
  - Confirm new documents are created, read, and updated correctly.
- **Frontend Testing**:
  - Check creation, display, and updating of posts and nested responses.
  - Confirm the threaded or hierarchical structure is rendered correctly.

2. Functional Test Cases

- **Coverage**: Include normal and edge cases (e.g., missing fields, invalid `parentId`).
- **Documentation**: Show expected vs. actual results with screenshots or console logs.

3. Load Testing with `npm loadtest`

- **Setup**: `npm install -g loadtest` or `npx loadtest`
- **Execution**:

    - Example: `loadtest --concurrency=10 --requests=1000 http://localhost:3000/postmessage`
- **Report**:
    - Provide requests per second, latency, error counts, etc.
    - Briefly discuss your server performance under load.

4. Challenges and Solutions

- Document any major issues (e.g., CouchDB credentials, Docker networking, or nesting logic) and how you addressed them.

Submission Requirements

1. **docker-compose.yml** (and `Dockerfile` if needed)
    - Must configure the Node.js server and CouchDB container so the app runs with `docker-compose up`.
2. **server.js**
    - Node.js server implementing the CouchDB integration, API endpoints, and document handling.
3. **frontend/** (React App)
    - Source code for the React UI.
    - Contains its own `package.json` with dependencies.
4. **package.json** (Node.js backend)
    - Must include relevant dependencies (`express`, `nano`, etc.) and scripts to run the server.
5. **report.pdf**
    - Detailed test report (functional and load testing) with focus on the creation of nested responses and overall system responsiveness.

Grading Matrix (Total 100 Points)

Part A: Node.js Backend with CouchDB Integration	Points
1. POST /postmessage endpoint (creating new posts)	10
2. POST /postresponse endpoint (creating new responses)	10
3. CouchDB Document Design & Nano Integration (unique IDs, timestamps)	10

4. Error Handling & Validation (missing fields, etc.)	5
5. Docker Configuration (for Backend & CouchDB)	5
Subtotal	**40**

Part B: React Frontend	**Points**
1. Displaying Existing Posts/Responses (including nesting)	10
2. Creating New Posts (async form submission, UI auto-update)	10
3. Creating New Responses (nested replies, UI auto-update)	10
4. Use of React UI Components (modal, styling, etc.)	10
5. Overall UX & Code Clarity	10
Subtotal	**50**

Part C: Test Report	**Points**
1. Test Approach & Functional Test Cases (screenshots/logs)	3
2. Load Testing with `npm loadtest` (commands, results)	3
3. Clarity & Presentation (challenges, solutions)	4
Subtotal	**10**

| **Total** | **100** |

Important Notes

1. **Required Modules**: Use only the allowed server-side modules (`express`, `body-parser`, `nano`, Node.js built-ins).
2. **Environment Configuration**: Use environment variables for CouchDB URL and credentials.
3. **Nested Responses**: Carefully handle the `parentId` to allow for multiple levels of responses.
4. **Docker Setup**: Your application should be accessible at `http://localhost:3000` when running `docker-compose up`.
5. **Testing**: Thoroughly test under various conditions (including concurrent load tests).