Assignment: Build and Deploy a Full-Stack Dashboard for Monitoring Real-Time Data with Mock Data Generation

Objective:

Create a full-stack dashboard using
Next.js 14 (with the app-router) for the frontend and **Express.js** for the backend. The dashboard will display live updates from a simulated real-time data source using **Socket.IO** for real-time communication. The backend will generate and insert mock data into a database every 5 seconds, storing the `datetime` in **UTC** along with a `numeric value` (e.g., temperature). The frontend will consume and display this data in real-time. Additionally, the application should be deployed to a cloud provider for viewing.

Example Data Structure:

Each mock data entry generated by the backend and stored in the database should follow this structure:

```
{
  "created_at": "2024-10-21T09:00:00Z",    // UTC timestamp
  "value": 72                              // Numeric value as temperature
}
```

- `created_at`: The time the data was generated, stored in **UTC** format (`Z` denotes UTC).
  - Example: `2024-10-21T09:00:00Z` (this is 09:00 AM UTC).
- `value`: A randomly generated or incremented number to represent the temperature data.

Part 1: Frontend (Next.js 14)

1. **Dashboard UI**:
   - Build a dashboard that displays a list of real-time temperature data.
   - Implement a **Chart** (**bar and line chart**) to visualize real-time data using **shadcn/ui** charts.
   - Allow the user to **choose a timezone** (Indonesia/Jakarta, Singapore, and Australia/Sydney).

- The **bar and line chart** should display the data **converted to the selected timezone.**
- The **bar and line chart** should update dynamically based on the selected timezone, without requiring a full page reload.
- Create a **responsive design** using **shadcn/UI.**

Specifics to the chart display:

- **X-axis**: Timestamps (adjusted to the selected timezone) should be displayed in **5-second intervals** to reflect the frequency of the mock data generation.
- **Y-axis**: The numeric values (e.g., temperature).
- Ensure that the **axes labels** clearly reflect what they represent (e.g., "Time (5-second intervals)" on the X-axis, "Temperature (°C)" on the Y-axis).

2. **Socket.IO Integration**:
   - Integrate **Socket.IO** on the frontend to receive real-time updates from the backend.
   - The chart should **automatically update** whenever new data is received through the WebSocket connection.
   - The connection should be established when the user visits the dashboard, and data should continuously update as long as the connection remains open.
3. **Data Fetching (Initial Load)**:
   - Implement **server-side rendering (SSR)** for the dashboard to load the initial data from the backend.
   - Use **Socket.IO** to handle the live updates after the initial data has been loaded.

Part 2: Backend (Express.js)

1. **Express API**:
   - Build an **Express.js** API that provides access to real-time data.
   - **Set up a Socket.IO server** that is attached to the **same HTTP server** used by Express. This will allow real-time communication with clients, so that both HTTP requests and WebSocket connections are handled by the same server.
   - Create an API endpoint (`/api/data`) that retrieves the initial data from the database and returns it as an array of objects, following the **Example Data Structure**:

```
{
  "created_at": "2024-10-21T09:00:00Z",    // UTC timestamp
  "value": 72                              // Numeric value as temperature
}
```

   - The same server should handle both Express routes and WebSocket events without requiring separate instances.

- **Socket.IO Real-Time Updates**:
  - Use **Socket.IO** to broadcast new mock data to all connected clients immediately after it is generated and inserted into the database.
  - Socket.IO should emit the data right after it's generated, ensuring the frontend receives updates in real-time.

    Example flow:

    - Generate mock data every 5 seconds.
    - Insert the mock data into the database.
    - Immediately emit the newly inserted data to all connected clients using Socket.IO.
- **Mock Data Generation**:
  - Create a function that generates mock data every 5 seconds.
  - Each data entry should include:
    - `created_at` **field**: Use **UTC time** when generating the timestamp (e.g., `new Date().toISOString()` in JavaScript). This ensures the data reflects a consistent global time.
    - `value` **field**: A randomly generated or incremented numeric value (e.g., a number between `1` and `100`).
- **Periodic Data Insertion**:
  - Use any `CRON library` to insert the generated data into the database every 5 seconds.
  - Ensure that each record has a proper `created_at` field with **UTC** timestamp and `value`.
- **WebSocket Communication**:
  - Implement real-time updates using **Socket.IO**. The **same HTTP server** used by Express will emit an event (e.g., `'new-data'`) to the clients whenever new mock data is generated, ensuring real-time updates are pushed to the frontend.
  - The frontend will automatically receive and update the chart based on the new data.

Part 3: Deployment

1. **Frontend Deployment**:
   - Deploy the **Next.js** frontend to a cloud provider (e.g., **Vercel**, **Netlify, any**).
   - Ensure the frontend is publicly accessible via a URL for review.
2. **Backend Deployment**:
   - Deploy the **Express.js** API along with **Socket.IO** to any cloud provider.
   - Ensure the backend is accessible, with real-time data being broadcast to the connected clients and stored correctly in the database.

3. **Database Setup**:
    - Deploy the database (MongoDB, PostgreSQL, or similar) to any cloud provider such as **MongoDB Atlas, AWS RDS or any**.
    - Ensure the backend is connected to the cloud database and that data is inserted into the database every 5 seconds.
4. **Connect Frontend and Backend**:
    - Ensure the frontend fetches data from the deployed backend, updating the dashboard in real-time based on the mock data generation.

Part 4: Testing (Frontend & Backend) Bonus

Frontend Tests:

1. **Test 1: Timezone Selection Behavior**
    - Write a test to ensure that when a user selects a specific timezone (e.g., Indonesia/Jakarta), the data displayed in the chart is correctly filtered and displayed in that timezone.
    - **What it tests**: This ensures the timezone logic is working properly, and the date-time conversion functions are applied correctly.
2. **Test 2: Real-Time Data Update**
    - Write a test to check that when new data arrives via **Socket.IO**, the chart updates accordingly without the user having to refresh the page.
    - **What it tests**: This tests the real-time update mechanism to ensure that the frontend reacts to incoming data appropriately through WebSocket communication.

Backend Tests:

1. **Test 1: Mock Data Generation**
    - Write a test to verify that the mock data generator creates valid data every 5 seconds, with `created_at` having the correct **UTC** timestamp format.
    - **What it tests**: This ensures the mock data generation function works as expected, generating accurate data with valid timestamps and values.
2. **Test 2: API Data Retrieval**
    - Write a test to ensure that the `/api/data` endpoint correctly retrieves and returns data from the database, and that it includes the `created_at` field in **UTC** format.
    - **What it tests**: This verifies the backend API is functioning correctly, and the data being retrieved from the database includes the necessary fields in the correct format.

Testing Instructions:

1. Include the test cases in your submission with instructions in the README on how to run them.
2. Tests should be automated and runnable via a command like `npm run test` or similar setup.

3. Clearly indicate what testing frameworks or tools were used (e.g., Jest, Mocha, React Testing Library).

Expectations & Evaluation Criteria

1. **Code Structure & Best Practices**:
   - Well-structured, clean code with proper comments following best practices for both **Next.js**, **Socket.IO**, and **Express.js**.
   - Meaningful commit messages and clear separation of concerns between frontend, backend, and database code.
2. **Real-Time Data Handling**:
   - Proficiency in handling real-time data generation, insertion into the database, and real-time updates on the frontend using **Socket.IO**.
3. **Timezone Management**:
   - Proper management and display of timezones when generating, storing, and presenting data. The candidate should demonstrate how they handle time zone conversions and adjustments when switching between different user-selected timezones.
4. **Frontend & UI**:
   - Functional, responsive UI using **shadcn/ui**. The chart must update dynamically based on user interactions and reflect the real-time data accurately.
5. **Backend Design**:
   - Scalable API with efficient handling of database interactions, data storage, and real-time data handling using **Socket.IO** for WebSocket communication.
6. **Testing**:
   - Well-written, maintainable tests for both frontend and backend components, ensuring that the functionality works as expected and any issues are caught early.
7. **Deployment**:
   - Demonstrates the ability to deploy both frontend and backend (with **Socket.IO**) to cloud platforms (e.g., Vercel, Heroku) and ensures seamless integration between the two.

Submission Instructions

1. **GitHub Repository**:
   - Provide a GitHub repository link containing:
     - Source code for both frontend and backend.
     - A `README.md` file with clear setup instructions, including how to run the project locally, deploy to the cloud, and run the tests.

2. **Deployed Application**:
    - Include the public URLs for both the frontend and backend (from the cloud provider) in the README.
3. **Additional Information**:
    - Briefly explain your approach, the challenges faced, and any assumptions made during development.