React – JSON-server and Firebase Real Time Database

• Question 1: What do you mean by RESTful web services?

Ans - RESTful web services are a way to provide communication between systems using the **REST (Representational State Transfer)** architecture. REST is a lightweight, scalable, and stateless architecture commonly used for building web APIs (Application Programming Interfaces).

• Question 2: What is Json-Server? How we use in React ?

Ans -  **What is json-server?**

json-server is a lightweight, easy-to-use tool that allows you to quickly set up a **RESTful API** by using a simple **JSON file** as your database. It is commonly used for **mock APIs** during frontend development and testing.

It helps simulate a backend by providing endpoints for **CRUD operations** without writing actual backend code.

**Key Features of json-server:**

* **Quick Setup:** Create a REST API in minutes.
* **Full CRUD Support:** Supports GET, POST, PUT, PATCH, DELETE.
* **Custom Routes:** Customize endpoints as needed.
* **Middleware Support:** Add custom behavior with middleware.
* **Pagination, Sorting & Filtering:** Easily handle advanced queries.

**How to Use json-server in a React Project**

**Step 1: Install json-server**

Make sure you have **Node.js** installed.

Open your **React project** folder and run:

npm install -g json-server

Or if you want it only for your project:

npm install json-server --save-dev

**Step 2: Set Up the JSON Database**

Create a db.json file in the root directory of your React project.

Example db.json:

{

"users": [

{

"id": 1,

"name": "Dinesh Rajput",

"email": "dinesh@example.com"

},

{

"id": 2,

"name": "Aarav Singh",

"email": "aarav@example.com"

}

]

}

**Step 3: Start json-server**

Run the following command:

npx json-server --watch db.json --port 5000

✅ **API will run on:** http://localhost:5000

**Step 4: Use json-server API in React**

You can interact with this mock API using fetch or **Axios**.

Example **GET request** in App.js:

import React, { useEffect, useState } from 'react';

import axios from 'axios';

const App = () => {

const [users, setUsers] = useState([]);

useEffect(() => {

// Fetching data from json-server

axios.get('http://localhost:5000/users')

.then((response) => {

setUsers(response.data);

})

.catch((error) => {

console.error('Error fetching data:', error);

});

}, []);

return (

<div>

<h1>User List</h1>

<ul>

{users.map(user => (

<li key={user.id}>{user.name} - {user.email}</li>

))}

</ul>

</div>

);

};

export default App;

**Step 5: CRUD Operations Using Axios**

1. **Create (POST Request)**:

axios.post('http://localhost:5000/users', {

name: 'New User',

email: 'newuser@example.com'

});

1. **Update (PUT/PATCH Request)**:

axios.put('http://localhost:5000/users/1', {

name: 'Updated Name'

});

1. **Delete (DELETE Request)**:

axios.delete('http://localhost:5000/users/1');

**Step 6: Automate with package.json**

Add a script to your package.json:

"scripts": {

"start": "react-scripts start",

"server": "json-server --watch db.json --port 5000"

}

Now you can run:

npm run server

Question 3: How do you fetch data from a Json-server API in React? Explain the role of fetch() or axios() in making API requests.

Ans -  **Fetching Data from a json-server API in React**

To fetch data from a **json-server** API in a React application, you can use either the **fetch()** method (built-in JavaScript) or the **axios** library (an external package for HTTP requests).

**1. Using fetch() Method**

The **fetch()** function is a **native JavaScript method** for making HTTP requests. It returns a **Promise** and supports asynchronous communication.

**🛠️ Example: Fetching Data with fetch()**

Here’s how to fetch data from a json-server API:

import React, { useEffect, useState } from 'react';

const App = () => {

const [users, setUsers] = useState([]);

useEffect(() => {

// Fetch data from json-server API

fetch('http://localhost:5000/users')

.then((response) => response.json()) // Convert response to JSON

.then((data) => {

setUsers(data); // Set data to state

})

.catch((error) => {

console.error('Error fetching data:', error);

});

}, []);

return (

<div>

<h1>User List</h1>

<ul>

{users.map((user) => (

<li key={user.id}>{user.name} - {user.email}</li>

))}

</ul>

</div>

);

};

export default App;

**How fetch() Works Step by Step:**

1. **Request:** Sends a GET request to http://localhost:5000/users.
2. **Response:** Receives the response and converts it to JSON.
3. **Update State:** Stores the data in the users state.
4. **Render:** Displays the fetched data in the component.

**2. Using axios Library**

**axios** is a popular HTTP client for JavaScript that simplifies making requests. It handles JSON conversion automatically and supports advanced features like interceptors and request cancellation.

**Installation:**

If axios is not already installed, run:

npm install axios

**Example: Fetching Data with axios()**

Here’s the same example using axios:

import React, { useEffect, useState } from 'react';

import axios from 'axios';

const App = () => {

const [users, setUsers] = useState([]);

useEffect(() => {

// Fetch data using axios

axios.get('http://localhost:5000/users')

.then((response) => {

setUsers(response.data); // Axios automatically parses JSON

})

.catch((error) => {

console.error('Error fetching data:', error);

});

}, []);

return (

<div>

<h1>User List</h1>

<ul>

{users.map((user) => (

<li key={user.id}>{user.name} - {user.email}</li>

))}

</ul>

</div>

);

};

export default App;

**How axios() Works Step by Step:**

1. **Request:** Sends a GET request using axios.get().
2. **Automatic JSON Parsing:** No need to manually convert the response.
3. **Update State:** Stores the received data in the users state.
4. **Render:** Displays the user list.

• Question 4: What is Firebase? What features does Firebase offer?

Ans - **What is Firebase?**

Firebase is a **Backend-as-a-Service (BaaS)** platform developed by **Google** that provides a comprehensive suite of cloud-based tools to build and manage web and mobile applications. It helps developers handle common backend tasks like **authentication**, **database management**, **cloud storage**, and **hosting** without managing servers.

**Core Features of Firebase:**

Firebase offers a wide range of services across three main categories: **Build**, **Release & Monitor**, and **Engage**.

**1. Build Better Apps**

These features help in creating and scaling your applications.

1. **Realtime Database:**
   * NoSQL cloud-hosted database that syncs data in **real time** across clients.
   * Useful for chat apps, collaborative tools, and live updates.
2. **Cloud Firestore:**
   * Modern **NoSQL database** for storing, syncing, and querying large datasets.
   * Supports **offline** access and **complex queries**.
3. **Firebase Authentication:**
   * Secure, easy-to-use system for **user authentication**.
   * Supports multiple sign-in methods:
     + Email/Password
     + Google, Facebook, Twitter, Apple
     + Phone number (OTP-based)
4. **Cloud Functions:**
   * Allows you to write **serverless backend code** that automatically runs in response to events (e.g., database updates, HTTP requests).
5. **Firebase Hosting:**
   * Fast and secure hosting for **web apps**, **static** and **dynamic** content.
   * Supports custom domains and automatic SSL.
6. **Cloud Storage:**
   * Stores and serves **large media files** (images, videos, PDFs).
   * Ideal for user-generated content and backups.

**2. Release & Monitor Performance**

Monitor app health and diagnose issues in real-time.

1. **Firebase Crashlytics:**
   * Real-time crash reporting for **Android** and **iOS** apps.
   * Helps identify and fix bugs quickly.
2. **Performance Monitoring:**
   * Tracks your app’s performance (e.g., screen load times, network requests).
   * Useful for diagnosing **slow page loads** and **API bottlenecks**.
3. **Test Lab:**
   * Cloud-based infrastructure for testing your app across multiple **devices** and **configurations**.

**3. Engage Your Users**

Enhance user engagement and retention with these tools.

1. **Firebase Cloud Messaging (FCM):**
   * Send **push notifications** across web, Android, and iOS.
   * Supports both targeted and global messaging.
2. **Firebase In-App Messaging:**
   * Display contextual in-app messages to **engage users** (e.g., feature announcements).
3. **Remote Config:**
   * Modify your app’s **behavior** and **appearance** dynamically without redeploying.
4. **Firebase Analytics:**
   * Free and unlimited **user analytics** to track user behavior and app performance.
   * Provides insights for marketing campaigns.
5. **Dynamic Links:**
   * Create smart links that redirect users to **specific screens** in your app, even after installation.

• Question 5: Discuss the importance of handling errors and loading states when working with APIs in React

Ans - When working with APIs in **React**, handling **errors** and **loading states** is essential for providing a **smooth user experience** and ensuring that your application is **robust and reliable**.

1. **Improves User Experience (UX):**
   * Users can see a **loading indicator** while data is being fetched.
   * Displays **error messages** when something goes wrong, instead of a blank screen.
2. **Prevents UI Breakage:**
   * If the API request fails or returns invalid data, handling errors prevents the app from **crashing**.
3. **Better Debugging:**
   * Clear error messages help identify **issues** quickly during development and in production.
4. **Optimizes Performance:**
   * Avoids **repeated API calls** and manages **retry logic** for failed requests.
5. **Enhances Accessibility:**
   * Provides feedback like "Loading..." or "Error occurred" for users with **slow connections**.

**Common Scenarios to Handle in React:**

1. **Loading State:** While waiting for API response.
2. **Error State:** If the API call fails due to network errors or invalid responses.
3. **Success State:** When the data is successfully fetched.
4. **Empty State:** When the API returns **no data**.

**Best Practices for Error and Loading State Management in React:**

1. **Always Handle Edge Cases:**
   * Check for **empty** or **undefined** data before rendering.
2. **Use Try-Catch with Async/Await:**
   * Ensures **proper error handling** in asynchronous API calls.
3. **Show Clear Messages:**
   * Display **meaningful** error and loading messages to the user.
4. **Graceful Degradation:**
   * If the API fails, the app should still function **partially** (e.g., offline mode).
5. **Centralize Error Handling:**
   * For large projects, use **Error Boundaries** or **Global Error Handlers**.