**Introduction to JSON Server with Fetch Method**

**Overview:** JSON Server is a lightweight server that allows you to easily set up a REST API with zero setup. It's perfect for mocking APIs during development or for prototyping.

**Key Features:**

* 🚀 Easy setup: JSON Server requires minimal configuration, making it effortless to get started.
* 📡 RESTful API: It automatically generates RESTful endpoints for your data, enabling smooth communication between client and server.
* 💻 Lightweight: JSON Server is lightweight, meaning it doesn't consume many system resources.
* 🔄 Real-time updates: It supports real-time updates, which is beneficial for collaborative applications or real-time data visualization.
* 🔐 Customizable: You can customize routes, responses, and behavior according to your needs.
* 🛠 Mocking data: JSON Server allows you to mock data easily, making it ideal for frontend development when the backend is still in progress.
* 🌐 -platform: It works seamlessly across different platforms, ensuring compatibility with various development environments.

**Installation:** To install JSON Server, you need Node.js and npm. Then, simply run the following command:

npm install -g json-server

**Usage:**

1. **Create a JSON file:** Prepare a JSON file containing your data.
2. **Start the server:** Run JSON Server with your JSON file as a database:

cssCopy code

json-server --watch your\_file.json

1. **Access your data:** Your RESTful API is now ready to use. You can access your data via HTTP requests.

**Example:** Suppose you have a JSON file named **db.json** with the following content:

jsonCopy code

{

"posts": [

{ "id": 1, "title": "Hello World" },

{ "id": 2, "title": "JSON Server is awesome!" }

]

}

Running **json-server --watch db.json** will start a server with endpoints for accessing posts.

**Conclusion:** JSON Server simplifies the process of setting up a mock REST API, providing developers with a convenient tool for prototyping and frontend development. With its ease of use and flexibility, it's a valuable asset in the developer toolkit.

The Fetch API in JavaScript provides several HTTP methods to interact with resources on the web. These methods correspond to different actions that can be performed on resources. The main HTTP methods supported by Fetch are:

1. **GET**: Used to retrieve data from a specified resource.
2. **POST**: Used to submit data to be processed to a specified resource.
3. **PUT**: Used to update data on a specified resource.
4. **DELETE**: Used to delete data from a specified resource.
5. **POST Request Example:**

javascriptCopy code

// Data to be added

const newData = { title: "New Post", body: "This is a new post." };

// POST request

fetch('http://localhost:3000/posts', {

method: 'POST',

headers: {

'Content-Type': 'application/json',

},

body: JSON.stringify(newData),

})

.then(response => response.json())

.then(data => console.log('POST request successful:', data))

.catch(error => console.error('Error:', error));

1. **DELETE Request Example:**

javascriptCopy code

// ID of the post to be deleted

const postIdToDelete = 1;

// DELETE request

fetch(`http://localhost:3000/posts/${postIdToDelete}`, {

method: 'DELETE',

})

.then(response => {

if (response.ok) {

console.log('DELETE request successful');

} else {

console.error('Failed to delete:', response.statusText);

}

})

.catch(error => console.error('Error:', error));

1. **PUT Request Example (Update):**

javascriptCopy code

// ID of the post to be updated

const postIdToUpdate = 2;

// Updated data

const updatedData = { title: "Updated Post", body: "This post has been updated." };

// PUT request

fetch(`http://localhost:3000/posts/${postIdToUpdate}`, {

method: 'PUT',

headers: {

'Content-Type': 'application/json',

},

body: JSON.stringify(updatedData),

})

.then(response => response.json())

.then(data => console.log('PUT request successful:', data))

.catch(error => console.error('Error:', error));

These examples demonstrate how to use the Fetch API to interact with JSON Server endpoints for performing POST, DELETE, and PUT requests. Make sure to replace **http://localhost:3000/posts** with the appropriate endpoint URL based on your JSON Server configuration.

Description Page

1. **<a href="descriptionpage.html?title=${encodeURIComponent(title)}&image=${encodeURIComponent(image)}&founder=${encodeURIComponent(founder)}&category=${encodeURIComponent(category)}&price=${encodeURIComponent(price)}">**
   * 📎 This is an HTML anchor (**<a>**) element with an **href** attribute pointing to a description page (**descriptionpage.html**) with query parameters.
   * 🔗 The query parameters are encoded using **encodeURIComponent()** to ensure special characters are properly handled in the URL.
2. **window.location.search**
   * 🌐 **window.location.search** retrieves the query string portion of the current URL.
   * 🧭 It includes everything from the question mark (**?**) onward.
3. **const a = new URLSearchParams(window.location.search)**
   * 🔄 This code creates a new **URLSearchParams** object (**a**) using the query string from the current URL.
   * 📊 It provides convenient methods to access and manipulate the query parameters.
4. **a.get()**
   * 🔍 **a.get()** is a method used to retrieve the value of a specific query parameter from the **URLSearchParams** object.
   * 🎯 It takes the parameter name as an argument and returns its corresponding value, if present.