

Yes, since the API you're using (<https://freetestapi.com/api/v1/users>) is a fake or test API, it might not support POST, PUT, DELETE requests as expected, which can lead to CORS errors or other issues when trying to perform create, update, or delete operations.

Here are a few reasons why this could happen:

1. CORS Errors:

- The API might not be configured to allow CORS (Cross-Origin Resource Sharing) for POST, PUT, DELETE requests from your React app. This is common with test APIs that are primarily read-only.

2. Read-Only API:

- Some fake or public APIs are designed for demonstration purposes and only allow GET requests. They might return static data and don't support adding, editing, or deleting data.

3. Limited API Capabilities:

- The API might not fully implement CRUD operations, so while you can retrieve data, making changes to it (such as adding or updating a user) isn't supported.

What You Can Do:

1. Use a Different API:

- If you need to perform actual CRUD operations (create, read, update, delete), consider using another test API like JSONPlaceholder or [MockAPI](#). These services support all HTTP methods and are designed for testing full CRUD operations.

2. Create Your Own Mock API:

- You can set up your own mock server using tools like [json-server](#) or [MirageJS](#). These tools allow you to simulate a full API that can handle all types of HTTP requests.

3. Disable CORS in Development:

- For development purposes, you can use browser extensions like Moesif CORS or configure a proxy in your React app to bypass CORS issues. However, this isn't recommended for production.

Example of Using [json-server](#):

You can create a local mock API using [json-server](#) to handle POST, PUT, DELETE requests:

Install [json-server](#):

```
bash
```

Copy code

```
npm install -g json-server
```

1.

Create a `db.json` file:

json

Copy code

```
{
  "users": [
    { "id": 1, "name": "Nick", "age": 25, "location": "New York" },
    { "id": 2, "name": "Sam", "age": 30, "location": "San Francisco" }
  ]
}
```

2.

Start the server:

bash

Copy code

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

3.

4. **Update your Axios calls:**

- Use <http://localhost:5000/users> as the API endpoint to perform all CRUD operations.

Alternative Solution:

- **MockAPI:** You can use [MockAPI](#) to create your own API with support for CRUD operations, and it provides a simple interface to manage your data and endpoints.

This way, you can practice making POST, PUT, DELETE requests without running into CORS issues or other limitations.

40

You said:

and if i want books also in db.json the? and the calling urls will be like?

ChatGPT said:

ChatGPT

If you want to add `books` data to your `db.json` file, you can simply extend the JSON structure to include a `books` array. Here's how you can modify your `db.json` file:

db.json File

json

Copy code

```
{
  "users": [
    { "id": 1, "name": "Nick", "age": 25, "location": "New York" },
    { "id": 2, "name": "Sam", "age": 30, "location": "San Francisco" }
  ],
  "books": [
    { "id": 1, "title": "The Great Gatsby", "author": "F. Scott Fitzgerald", "year": 1925 },
    { "id": 2, "title": "1984", "author": "George Orwell", "year": 1949 },
    { "id": 3, "title": "To Kill a Mockingbird", "author": "Harper Lee", "year": 1960 }
  ]
}
```

Running the Server

Once you have updated the `db.json` file, you can start the `json-server` as before:

bash

Copy code

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

API Endpoints

Now that you have both `users` and `books` in your `db.json`, you can interact with them using the following URLs:

- **Users Endpoints:**
 - **GET all users:** `http://localhost:5000/users`
 - **GET a single user by ID:** `http://localhost:5000/users/1`
 - **POST a new user:** `http://localhost:5000/users`
 - **PUT (update) a user by ID:** `http://localhost:5000/users/1`
 - **DELETE a user by ID:** `http://localhost:5000/users/1`
- **Books Endpoints:**
 - **GET all books:** `http://localhost:5000/books`
 - **GET a single book by ID:** `http://localhost:5000/books/1`
 - **POST a new book:** `http://localhost:5000/books`

- **PUT (update) a book by ID:** `http://localhost:5000/books/1`
- **DELETE a book by ID:** `http://localhost:5000/books/1`

Example Axios Calls

Here are some examples of how you can use Axios to interact with the `users` and `books` endpoints:

Fetching Users

javascript

Copy code

```
axios.get('http://localhost:5000/users')
  .then(response => {
    console.log(response.data);
  })
  .catch(error => {
    console.error('There was an error fetching the users!', error);
  });
```

Fetching Books

javascript

Copy code

```
axios.get('http://localhost:5000/books')
  .then(response => {
    console.log(response.data);
  })
  .catch(error => {
    console.error('There was an error fetching the books!', error);
  });
```

Adding a New User

javascript

Copy code

```
axios.post('http://localhost:5000/users', {
  name: "John",
  age: 28,
  location: "Miami"
})
  .then(response => {
    console.log('User added:', response.data);
  })
  .catch(error => {
```

```
    console.error('There was an error adding the user!', error);
  });
```

Adding a New Book

javascript

Copy code

```
axios.post('http://localhost:5000/books', {
  title: "New Book Title",
  author: "New Author",
  year: 2024
})
.then(response => {
  console.log('Book added:', response.data);
})
.catch(error => {
  console.error('There was an error adding the book!', error);
});
```

React Router Example

If you want to create routes for both users and books in your React app, you can use React Router like this:

javascript

Copy code

```
import { BrowserRouter as Router, Route, Switch } from
'react-router-dom';
import UserList from './UserList';
import BookList from './BookList';

function App() {
  return (
    <Router>
      <Switch>
        <Route path="/users" component={UserList} />
        <Route path="/books" component={BookList} />
      </Switch>
    </Router>
  );
}
```

```
export default App;
```

Conclusion

With this setup, you can manage both `users` and `books` data in your local mock API. You can perform CRUD operations on both collections using the appropriate endpoints, and you can easily extend this setup to include more data types if needed.