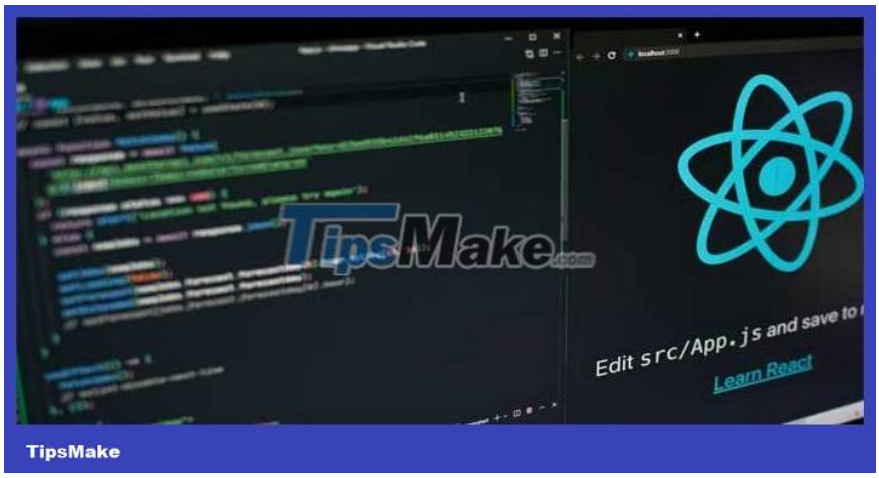
How to build and use Mock API in React app using Mirage.js

Don't have an API available? Alright! Let's program and use mock API with Mirage.js.

Don't have an API available? Alright! Let's **program and use mock API with Mirage.js** .



When developing full-stack applications, much of the frontend work depends on real-time data from the backend.

This means that you must pause front-end development until the API is available for use. However, waiting for an API to be ready to set up a user interface can significantly reduce productivity and lengthen project time.

A great solution to this challenge involves using mock APIs. These APIs allow you to develop and test the frontend with data that mimics real data structures without relying on the actual API.

Instructions for using Mirage.js Mock API

Mirage is a JavaScript library that allows you to create mock APIs, complete with a test server running on the client side of your web application. This means you can test the frontend code without worrying about the availability or behavior of the actual backend API.

To use Mirage.js, you first need to create mock API endpoints and define the response they will return. Mirage.js then intercepts all HTTP requests your frontend code makes and instead returns mock responses.

Once the API is ready, you can easily switch to it by just changing the configuration of Mirage.js.

Create Mock API server with Mirage.js

To demonstrate how to set up a mock API, you'll build a simple to-do React app using the Mirage.js backend. But first, create a React app using the create-react-app command. Additionally, you can use Vite to set up a React project. Next, install the Mirage.js dependency.

```
npm install --save-dev miragejs
```

Now create a Mirage.js server instance to intercept requests and simulate API responses, using the **createServer** method . This method takes a configuration object as a parameter.

This object includes **the environment** and **namespace** for the API. This environment defines the development stage of the API, such as development while the namespace is a prefix added to all API endpoints.

Create a new **src/server.js** file and include the following code:

```
import { createServer, Model } from 'miragejs'; const DEFAULT_CONFIG = { environ
```

If necessary, you can customize the namespace to match the URL structure of the actual API, including specifying the version. This way, once your API is ready, you can easily integrate it into your front-end application with minimal code changes.

Additionally, in the server instance configuration, you can also define a data model to simulate data storage and retrieval in a mock environment.

Finally, start the Mirage.js server by importing the server object in the **index.jsx** or **main.js** file as follows:

```
import React from 'react' import ReactDOM from 'react-dom/client' import App fro
```

Add sample data to Mock API

Mirage.js has an in-memory database that you can use to pre-populate mock APIs with initial raw data and manage test data from your client application. This means you can store and fetch test data from the simulation database and use it in the client application.

To add sample data to the Mock API, add the following code in the **server.js file right below the models** object .

```
seeds(server) { server.create('Todo', { title: 'item no 1', body: 'Do something
```

The seeds function provides the Mirage.js server with three to-do items, each with its own title and description. Optionally, instead of hard-coding the test data, you can integrate a library like Faker.js to generate the necessary test data.

Identify the Mock API roadmap

Now define some API routes for the mock API. In this case, define the route for handling mock API GET, POST, and DELETE queries.

Just below the sample data, add the code:

```
routes() { this.namespace = 'api/todos'; this.get('/', (schema, request) => { re
```

Build React clients

After setting up the mock API, build a React client to interact and use the API endpoints. You're free to use any of your favorite UI component libraries, but this tutorial will use Chakra UI to style the app.

First, install the dependencies:

```
npm install @chakra-ui/react @emotion/react @emotion/styled framer-motion
```

Next, create a new **src/components/TodoList.jsx** file , and include the following code:

```
import React, { useState, useEffect } from 'react'; import { Button, Box, Contai
```

Now define a functional component to display the to-do list UI, including input fields for adding new tasks and a list of existing tasks.

```
export default function TodoList() { return ( Todo List
 {loading ? ( Loading... ) : ( todos.map((todo) => ( {todo.body}
handleDelete(todo.id)}>Delete
 )) )} ); }
```

Add Todo

Now define handler functions for adding and removing operations. But first, add these states. Additionally, you can use **useReducer** to define state management logic for your to-do list app.

```
const [todos, setTodos] = useState([]); const [newTodo, setNewTodo] = useState({
```

Now define the logic to retrieve and display sample data in the in-memory database when the application first loads in the browser by including the **fetch** method in the **useEffect** hook .

```
useEffect(() => { fetch('/api/todos') .then((response) => response.json()) .then
```

The renderKey state is also included in useEffect to ensure that this code triggers the re-rendering of newly added data in the memory database while the server is running.

Simply put, whenever a user adds data that needs to be refreshed to the Mirage.js database - this component will re-render to show the updated data.

Add data to the API

Now define the logic for adding data to the API via POST queries. Just below the useEffect hook, include the following code:

```
const handleInputChange = (e) => { const { name, value } = e.target; setNewTodo(
```

When the user enters data in the todo input field and clicks the **Add Todo button, the code updates the newTodo** state with the user's input. It then sends the sample POST query to the API using the new data object in the query body to save it to the in-memory database.

If the POST query is successful, this code adds the new item to the todos array, and finally, triggers the re-rendering of this element to show the new to-do item.

Mock API DELETE query

Now, define the logic to delete data via sample DELTE API queries. This process involves sending a DELETE query to remove the to-do item from the in-memory database. **If successful, update both the todo** and **loading** states to show the deletion process.

```
const handleDelete = (id) => { let deleteInProgress = true; fetch(`/api/todos/${
```

Remember that this process can only delete newly added data, not sample data.

Finally, import the **TodoList** component in the **App.jsx** file to display it in the DOM.

```
import TodoList from './components/TodoList'; //code .
```

It's done! When you start the programming server, you can fetch sample data, add & delete new data from the sample API in your React app.

You finished reading the article "**How to build and use Mock API in React app using Mirage.js**" edited by the **TipsMake** team. We hope this article has provided you with many useful tech tips and tricks. You can search for similar articles on tips and guides. Thank you for reading and for following us regularly.

© 2019 TipsMake.com