How to Consume REST APIs in React – a Beginner's Guide

React is a popular frontend library that developers use to create applications. And you will need to integrate APIs into your React application at some point if you want to build production-ready apps.

Every developer who wants to build modern, robust web applications with React must understand how to consume APIs to fetch data into their React applications.

In this beginners guide, you will learn how to consume RESTful API in React, including fetching, deleting, and adding data. We'll also go over the two main ways to consume RESTful APIs and how to use them with React hooks.

What is a REST API?

If you've ever spent any time programming or researching programming, you've likely come across the term "API."

API stands for Application Programming Interface. It is a medium that allows different applications to communicate programmatically with one another and return a response in real time.

Roy Fielding defined REST in 2000 as an architectural style and methodology commonly used in the development of internet services, such as distributed hypermedia systems.

It is an acronym that stands for "REpresentational State Transfer."

When a request is made via a REST API, it sends a representation of the resource's current state to the requester or endpoint. This state representation can take the form of JSON (JavaScript Object Notation), XML, or HTML.

JSON is the most widely used file format because it is language-independent and can be read by both humans and machines.

For example:


```
[
    {
        "userId": 1,
        "id": 1,
        "title": "sunt excepturi",
        "body": "quia et suscipit\nsuscipit recusandae consequuntur "
    },
    {
        "userId": 1,
        "id": 2,
        "title": "qui est esse",
        "body": "est rerum tempore vitae\nsequi sint nihil"
    }
]
```


How to Consume REST API's in React

You can consume REST APIs in a React application in a variety of ways, but in this guide, we will look at two of the most popular approaches: Axios (a promise-based HTTP client) and Fetch API (a browser in-built web API).

Note: To fully comprehend this guide, you should be familiar with JavaScript, React, and React hooks, as they are central to it.
Before we get into how to consume APIs, it's important to understand that consuming APIs in React is very different from how it's done in JavaScript. This is because these requests are now done in a React Component.

In our case, we'll be using functional components, which means that we need to use two major React Hooks:

- **useEffect Hook:** In React, we perform API requests within the `useEffect()` hook. It either renders immediately when the app mounts or after a specific state is reached. This is the general syntax we'll use:

```
useEffect(() => {
    // data fetching here
}, []);
```

- **useState Hook:** When we request data, we must prepare a state in which the data will be stored when it is returned. We can save it in a state management tool such as Redux or in a context object. To keep things simple, we'll store the returned data in the React local state.

```
const [posts, setPosts] = useState([]);
```

Let's now get into the meat of this guide, where we'll learn how to get, add, and delete data using the JSONPlaceholder posts API. This knowledge is applicable to any type of API, as this guide is intended for beginners.

How to Consume APIs Using The Fetch API

The Fetch API is a JavaScript built-in method for retrieving resources from a server or an API endpoint. It's built-in, so you don't need to install any dependencies or packages.

The `fetch()` method requires a mandatory argument, which is the path or URL to the resource you want to fetch. Then it

returns a Promise so you can handle success or failure using the `then()` and `catch()` methods.
A basic fetch request is very simple to write and looks like the below code. We are simply fetching data from a URL that returns data as JSON and then logging it to the console:

```
fetch('https://jsonplaceholder.typicode.com/posts?_limit=10')
    .then(response => response.json())
    .then(data => console.log(data));
```

The default response is usually a regular HTTP response rather than the actual JSON, but we can get our output as a JSON object by using the response's json() method.

How to Perform a GET Request in React With Fetch API

You can use the HTTP GET method to request data from an endpoint.

As previously stated, the Fetch API accepts one mandatory argument, which is true. It also accepts an option argument, which is optional, especially when using the GET method, which is the default. But for other methods such as POST and DELETE, you'll need to attach the method to the options array:

```
fetch(url, {
      method: "GET"  // default, so we can ignore
})
```

So far, we've learned how things work, so let's put everything we've learned together and perform a get request to fetch data from our API.

Again, we'll be using the free online API JSONPlaceholder to fetch a list of posts into our application:

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

const App = () => {
    const [posts, setPosts] = useState([]);
```

```
    useEffect(() => {
        fetch('https://jsonplaceholder.typicode.com/posts?_limit=10')
            .then((response) => response.json())
            .then((data) => {
                console.log(data);
                setPosts(data);
            })
            .catch((err) => {
                console.log(err.message);
            });
    }, []);

return (
    // ... consume here
);
};
```

We created a state in the preceding code to store the data we will retrieve from the API so that we can consume it later in our application. We also set the default value to an empty array.

```
const [posts, setPosts] = useState([]);
```

The major operation then occurred in the useEffect state, so that the data/posts are fetched as soon as the application loads. The fetch request yields a promise, which we can either accept or reject:

```
useEffect(() => {
    fetch('https://jsonplaceholder.typicode.com/posts?_limit=10').then(
        (response) => console.log(response)
    );
}, []);
```

This response contains a large amount of data, such as the status code, text, and other information that we'll need to have to handle errors later.

So far, we've handled a resolve using `.then()`, but it returned a response object, which isn't what we want. So we need to resolve the Response object to JSON format using the `json()` method. This also returns a promise for us to get the actual data using the second `.then()`.

```
useEffect(() => {
    fetch('https://jsonplaceholder.typicode.com/posts?_limit=10')
        .then((response) => response.json())
        .then((data) => {
            console.log(data);
            setPosts(data);
        });
}, []);
```

If we look at the console, we'll see that we've retrieved 10 posts from our API, which we've also set to the state we specified earlier.

This is not complete because we have only handled the promise's resolve and not the promise's rejection, which we'll handle using the `.catch()` method:

```
useEffect(() => {
    fetch('https://jsonplaceholder.typicode.com/posts?_limit=10')
        .then((response) => response.json())
        .then((data) => {
            console.log(data);
            setPosts(data);
        })
        .catch((err) => {
            console.log(err.message);
        });
}, []);
```

So far we have seen how to perform a `GET` request. This can be consumed easily into our application by looping through our array:

```
const App = () => {
// ...
```

```
    return (
    <div className="posts-container">
        {posts.map((post) => {
            return (
                <div className="post-card" key={post.id}>
                    <h2 className="post-title">{post.title}</h2>
                    <p className="post-body">{post.body}</p>
                    <div className="button">
                    <div className="delete-btn">Delete</div>
                    </div>
                </div>
            );
        })}
    </div>
    );
};

export default App;
```

How to Perform a POST Request in React With Fetch API

You can use the HTTP `POST` method to send data to an endpoint. It works similarly to the `GET` request, the main difference being that you need to add the method and two additional parameters to the optional object:

```
const addPosts = async (title, body) => {
await fetch('https://jsonplaceholder.typicode.com/posts', {
method: 'POST',
body: JSON.stringify({
    title: title,
    body: body,
    userId: Math.random().toString(36).slice(2),
}),
headers: {
    'Content-type': 'application/json; charset=UTF-8',
},
})
.then((response) => response.json())
.then((data) => {
    setPosts((posts) => [data, ...posts]);
```

```
    setTitle('');
    setBody('');
})
.catch((err) => {
    console.log(err.message);
});
};
```

The major parameters that might appear strange are the body and header.

The body holds the data we want to pass into the API, which we must first stringify because we are sending data to a web server. The header tells us the type of data, which is always the same when consuming REST API's. We also set the state to hold the new data and distribute the remaining data into the array.

Looking at the `addPost()` method we created, it expects these data from a form or whatever. In our case, I created a form, obtained the form data via states, and then added it to the method when the form was submitted:

```
import React, { useState, useEffect } from 'react';
const App = () => {
const [title, setTitle] = useState('');
const [body, setBody] = useState('');
// ...
const addPosts = async (title, body) => {
    await fetch('https://jsonplaceholder.typicode.com/posts', {
        method: 'POST',
        body: JSON.stringify({
            title: title,
            body: body,
            userId: Math.random().toString(36).slice(2),
        }),
        headers: {
            'Content-type': 'application/json; charset=UTF-8',
        },
    })
```

```
        .then((response) => response.json())
        .then((data) => {
            setPosts((posts) => [data, ...posts]);
            setTitle('');
            setBody('');
        })
        .catch((err) => {
            console.log(err.message);
        });
};

const handleSubmit = (e) => {
    e.preventDefault();
    addPosts(title, body);
};

return (
    <div className="app">
        <div className="add-post-container">
            <form onSubmit={handleSubmit}>
                <input type="text" className="form-control" value={title}
                    onChange={(e) => setTitle(e.target.value)}
                />
                <textarea name="" className="form-control" id="" cols="10" rows="8"
                    value={body} onChange={(e) => setBody(e.target.value)}
                ></textarea>
                <button type="submit">Add Post</button>
            </form>
        </div>
        {/* ... */}
    </div>
);
};

export default App;
```

How to Perform a DELETE Request in React With Fetch API

You can use the HTTP `DELETE` method to remove data from an endpoint. It works similarly to the `GET` request, the main difference being the addition of the method:

```
const deletePost = async (id) => {
await fetch(`https://jsonplaceholder.typicode.com/posts/${id}`, {
    method: 'DELETE',
}).then((response) => {
    if (response.status === 200) {
        setPosts(
            posts.filter((post) => {
                return post.id !== id;
            })
        );
    } else {
        return;
    }
});
};
```

This gets triggered when the button is clicked, and we get the `id` of the specific post in which the button was clicked. Then we remove that data from the entire retuned data. This will be removed from the API but not immediately from the UI, which is why we have added a filter to remove the data as well. For each item in the loop, your delete button will look like this:

```
const App = () => {
// ...

    return (
    <div className="posts-container">
        {posts.map((post) => {
            return (
                <div className="post-card" key={post.id}>
                    {/* ... */}
                    <div className="button">
                        <div className="delete-btn" onClick={() => deletePost(post.id)}>
                            Delete
```

```
                    </div>
                </div>
            </div>
        );
    })}
</div>
);
};

export default App;
```

How to Use Async/Await in Fetch API

So far, we've seen how to make fetch requests normally using the promise syntax, which can be confusing at times. Then comes the chaining. We can avoid the `.then()` chaining by using Async/await and write more readable code.

To use async/await, first call `async` in the function. Then when making a request and expecting a response, add the `await` syntax in front of the function to wait until the promise settles with the result.

When we use async/await, all of our Fetch requests will look like this:

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

const App = () => {
    const [title, setTitle] = useState('');
    const [body, setBody] = useState('');
    const [posts, setPosts] = useState([]);

    // GET with fetch API
    useEffect(() => {
        const fetchPost = async () => {
            const response = await fetch(
                'https://jsonplaceholder.typicode.com/posts?_limit=10'
            );
            const data = await response.json();
            console.log(data);
```

```
        setPosts(data);
    };
    fetchPost();
}, []);

// Delete with fetchAPI
const deletePost = async (id) => {
    let response = await fetch(
        `https://jsonplaceholder.typicode.com/posts/${id}`,
        {
            method: 'DELETE',
        }
    );
    if (response.status === 200) {
        setPosts(
            posts.filter((post) => {
                return post.id !== id;
            })
        );
    } else {
        return;
    }
};

// Post with fetchAPI
const addPosts = async (title, body) => {
    let response = await fetch('https://jsonplaceholder.typicode.com/posts', {
        method: 'POST',
        body: JSON.stringify({
            title: title,
            body: body,
            userId: Math.random().toString(36).slice(2),
        }),
        headers: {
            'Content-type': 'application/json; charset=UTF-8',
        },
    });
    let data = await response.json();
```

```
        setPosts((posts) => [data, ...posts]);
        setTitle('');
        setBody('');
    };

    const handleSubmit = (e) => {
        e.preventDefault();
        addPosts(title, body);
    };

    return (
        // ...
    );
};

export default App;
```

How to Handle Errors with Fetch API

In this section, we'll look at how to handle errors both traditionally and with async/await.

We can use the response data to handle errors in the Fetch API, or we can use the try/catch statement when using async/await.

Let's look at how we can do this typically in Fetch API:

```
const fetchPost = () => {
fetch('https://jsonplaceholder.typicode.com/posts?_limit=10')
    .then((response) => {
        if (!response.ok) {
            throw Error(response.statusText);
        }
        return response.json();
    })
    .then((data) => {
        console.log(data);
        setPosts(data);
```

```
    })
    .catch((err) => {
        console.log(err.message);
    });
};
```

You can read more about Fetch API errors here.

And for async/await we can use the `try` and `catch` like this:

```
const fetchPost = async () => {
    try {
        const response = await fetch(
            'https://jsonplaceholder.typicode.com/posts?_limit=10'
        );
        const data = await response.json();
        setPosts(data);
    } catch (error) {
        console.log(error);
    }
};
```

How to Consume APIs Using Axios

Axios is an HTTP client library based on promises that makes it simple to send asynchronous HTTP requests to REST endpoints. This endpoint in our case is the JSONPlaceholder Posts API, to which we will make `GET`, `POST`, and `DELETE` requests.

How to Install and Configure an Axios Instance

Axios, unlike the Fetch API, is not built-in, so we will need to incorporate it into our project in order to use it.

You can add Axios to your project by running the following command:

```
npm install axios
```

Once you've successfully installed Axios, we can proceed to create an instance, which is optional but recommended as it saves us from unnecessary repetition.

To create an instance, we use the `.create()` method, which we can use to specify information such as the URL and possibly headers:

```
import axios from "axios";

const client = axios.create({
    baseURL: "https://jsonplaceholder.typicode.com/posts"
});
```

How to Perform a GET Request in React With Axios

We will use the instance we declared earlier for to perform the GET request. All we will do is set the parameters, if any, and get the response as JSON by default.

Unlike the Fetch API method, no option is required to declare the method. We simply attach the method to the instance and query it.

```
useEffect(() => {
    client.get('?_limit=10').then((response) => {
        setPosts(response.data);
    });
}, []);
```

How to Perform a POST Request in React With Axios

As previously stated, you can use the `POST` method to send data to an endpoint. It functions similarly to the `GET` request, with the main difference being the requirement to include the method and an option to hold the data we are sending in:

```
const addPosts = (title, body) => {
    client
        .post('', {
            title: title,
            body: body,
        })
        .then((response) => {
            setPosts((posts) => [response.data, ...posts]);
```

```
    });
};
```

How to Perform a DELETE Request in React With Axios

We can perform delete requests using the delete method, which gets the `id` and deletes it from the API. We'll also use the filter method to remove it from the UI, as we did with the Fetch API method:

```
const deletePost = (id) => {
    client.delete(`${id}`);
    setPosts(
        posts.filter((post) => {
            return post.id !== id;
        })
    );
};
```

How to Use Async/Await in Axios

So far, we've seen how to make Axios requests using the promise syntax. But now let's see how we can use async/await to write less code and avoid the `.then()` chaining.

When we use async/await, all of our Axios requests will look like this:

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

const App = () => {
    const [title, setTitle] = useState('');
    const [body, setBody] = useState('');
    const [posts, setPosts] = useState([]);

    // GET with Axios
    useEffect(() => {
        const fetchPost = async () => {
            let response = await client.get('?_limit=10');
            setPosts(response.data);
        };
        fetchPost();
```

```
  }, []);

  // Delete with Axios
  const deletePost = async (id) => {
    await client.delete(`${id}`);
    setPosts(
      posts.filter((post) => {
        return post.id !== id;
      })
    );
  };

  // Post with Axios
  const addPosts = async (title, body) => {
    let response = await client.post('', {
      title: title,
      body: body,
    });
    setPosts((posts) => [response.data, ...posts]);
  };

  const handleSubmit = (e) => {
    e.preventDefault();
    addPosts(title, body);
  };

  return (
    // ...
  );
};

export default App;
```

How to Handle Errors with Axios

For promise-based Axios requests, we can use the `.then()` and `.catch()` methods, but for async/await, we can use the `try...catch` block. This is very similar to how we implemented the Fetch API, and the `try...catch` block will look like this:

```
const fetchPost = async () => {
    try {
        let response = await client.get('?_limit=10');
        setPosts(response.data);
    } catch (error) {
        console.log(error);
    }
};
```

You can read more about handling errors with Axios here.

Fetch API vs Axios

You may have noticed some differences, but let's put them in a handy table so we can compare Fetch and Axios properly.

These distinctions will help you decide which method to use for a specific project. Among these distinctions are:

AXIOS	FETCH
Axios is a standalone third-party package that is simple to install.	Fetch is built into most modern browsers. **No installation** is required as such.
Axios uses the **data** property.	Fetch uses the **body** property.
Axios data contains the **object**.	Fetch's body has to be **stringified**.
When the status is 200 and the statusText is 'OK,' the Axios request is accepted.	Fetch request is ok when **response object contains the ok property**.
Axios performs **automatic transforms of JSON data**.	Fetch is a **two-step process** when handling JSON data- first, to make the actual request; second, to call the .json() method on the response.
Axios allows **cancelling request and request timeout**.	Fetch does not.
Axios has **built-in support for download progress**.	Fetch does not support upload progress.
Axios has **wide browser support**.	Fetch is only compatible with Chrome 42+, Firefox 39+, Edge 14+, and Safari 10.1+. (This is known as Backward Compatibility).

Conclusion

In this guide, we learned how to consume REST APIs in React using either the Fetch API or Axios.

This will help you get started with API consumption in React, and from there you will be able to consume data in more complex ways and manipulate your APIs however you choose.