## When to Make Ajax Requests in React

When working with a web app, we should have data to be rendered, and the data should be consumed either from local mock data or from a remote endpoint.
React does not have an in-built package for calling the API because React is a library, as opposed to Angular, which is a complete model view controller framework.
We need to call APIs from a remote endpoint to access external resources, which we can do using Ajax to configure the request and respond to the resources.
But the limitation is that we cannot access the remote endpoint, so we need third-party libraries such as Axios or an in-built one like Fetch to consume the data coming from the server.

## Making Ajax Calls in React

As we know, React is a JavaScript-based library that does not have the ability to make HTTP requests; thus, we need to use third-party libraries to achieve this.
There are plenty of libraries available to make HTTP calls into React apps. A few of them are listed below.

- Axios
- Fetch
- Superagent
- React-axios
- Use-http
- React-request

Many more third-party libraries are supported for making HTTP requests; We can choose any..

## Using Axios to Make HTTP Calls in React

Axios is a promise-based HTTP client that supports the browser and the node.js as well.
Below are the few features of the Axios library.

- It is used to make XMLHttpRequests from the browser .
- It makes HTTP requests from the node.js platform .
- It intercepts the request and the response .
- It automatically transforms JSON-specific data .
- You can cancel requests any time .
- It supports the complete promise-based API .
- It transforms the request and response data .

As we can see, many features are supported; that's why the Axios is one of the most widely used libraries on the market.
Let's install the Axios library into the existing application using the below command.

```
npm install axios
```

After executing the above command, we will be able to use the Axios library into the existing React application.
But if we don't want to use npm package, then we can also import the script into the existing application like this.

```
<script src="https://unpkg.com/axios/dist/axios.min.js"></script>
```

The basic syntax of Axios will look like this.

```
axios
.get("API_URL")
.then(response => {
   // manipulate the response here
})
.catch(function(error) {
   // manipulate the error response here
});
```

As you can see in the this syntax, we should pass the API URL accordingly so that it can access the remote endpoint to fetch the data from the server.
After getting the appropriate response from the server, we should manipulate the response from the component so that we can make use of the response data.
Let's jump directly into an example that shows how to use Axios in our React app by using a dummy endpoint URL to get dummy data to render.

```
async getTodos() {
   // Simple one
    let todos = await axios.get(
     "https://jsonplaceholder.typicode.com/todos?_page=1&_limit=10"
   );
   console.log(todos.data);
 }
```

In this example, we have used the Axios package and the GET HTTP call, with the URL provided as an argument.
But what if we get the response from the server? We don't have a section in the above example that shows how to use the response from the server, so the solution is given below along with the response section.

```
axios
.get("API_URL")
.then(response => {
   this.setState({ todos: response.data });
})
.catch(function(error) {
   console.log(error);
});
```

The .then the section manages the response if we get the expected response from the server. If not, then we have an additional section called .catch that is used to manage the error while accessing the remote endpoint.
Now let’s access the different properties of the response object, like this.

```
async getTodos() {
// With all properties
    axios
        .get("https://jsonplaceholder.typicode.com/todos?_page=1&_limit=10")
        .then(response => {
            console.log(response.data);
            console.log(response.status);
            console.log(response.statusText);
            console.log(response.headers);
            console.log(response.config);
        })
        .catch(function(error) {
            console.log(error);
        });
}
```

We have accessed different properties of the response object, including data, status, headers, config, and status text. We can make use of such properties to do things like show the response status, like 200, 404, 502, and so on.
Let’s post the data using the Axios POST request with this example.

```
async getTodos() {
    // With all properties
     let body = {
      userId: 1111,
      title: "This is POST request with body",
      completed: true
    };
    axios
      .post("https://jsonplaceholder.typicode.com/todos", body)
      .then(function(response) {
        console.log(response.data);
      })
      .catch(function(error) {
        console.log(error);
      });
}
```

Here in this example, we have used the POST request along with the Axios client that passes the additional request data to post the record to the server.
The complete example will look like this.

```
import React, { Component } from "react";
import axios from "axios";
```

```
class UsingAxios extends Component {
  constructor() {
    super();
    this.state = {
      name: "React"
    };
    this.getTodos = this.getTodos.bind(this);
  }

    componentDidMount() {
        this.getTodos();
    }

    async getTodos() {
        // With all properties
        let body = {
        userId: 1111,
        title: "This is POST request with body",
        completed: true
        };
        axios
        .post("https://jsonplaceholder.typicode.com/todos", body)
        .then(function(response) {
            console.log(response.data);
        })
        .catch(function(error) {
            console.log(error);
        });
    }

    render() {
        const { todos } = this.state;
        return (
        <div>
            <h3>Using Axios in React for API call</h3>
            <hr />
        </div>
        );
    }
}

export default UsingAxios;
```

This example shows that we will add one todo item using the Axios POST request, bypassing the additional object, which is the request object. If the request is successful, then the data can be used for further manipulation.
That is why Axios is a good promise-based HTTP client to make HTTP-based calls to access data from the server or updat existing data.
Now we will use a different approach for making an HTTP request alled fetch().

## Using Fetch to Make HTTP Calls in React

Fetch is browser-based API which is widely used to make HTTP calls from apps, but it's not limited to React—it can also be used from other frameworks such as Angular or Vuejs.
Fetch is a modern method used by developers around the globe because it supports all the modern browsers, but we have to keep in mind that a few browsers still don't support Fetch, which is a drawback.
Let's look at the basic syntax of the Fetch API, which looks like this.

```
let response = fetch(
  API_URL,
  [additioanl options]
);
```

Here in the syntax, we pass the API URL, which is mandatory to fetch the data from the remote server. If we have other request types, like POST or PUT, then we can also pass the additional arguments along with the URL.
We can simply use the URL with the Fetch API like this.

```
let response = fetch("https://jsonplaceholder.typicode.com/users");
```

The Fetch API always returns the promise so that it can be resolved into the response object as soon as we get the appropriate response from the remote server.
Let's look at the additional options we can use along with the Fetch API with a simple example.

```
async getUsers() {
  // With additional headers
  const response = await fetch("https://jsonplaceholder.typicode.com/users", {
    method: "GET", // *Type of request GET, POST, PUT, DELETE
    mode: "cors", // Type of mode of the request
    cache: "no-cache", // options like default, no-cache, reload, force-cache
    credentials: "same-origin", // options like include, *same-origin, omit
    headers: {
      "Content-Type": "application/json" // request content type
    },
    redirect: "follow", // manual, *follow, error
    referrerPolicy: "no-referrer", // no-referrer, *client
    // body: JSON.stringify(data) // Attach body with the request
  });
  this.setState({ users: await response.json() });
 }
```

Here in this example, we have a simple API URL, but along with the URL, we have used various arguments.

- Method: Which type of HTTP calls
- Mode: The type of request mode, i.e. cors, no-cors, and so on
- Headers: The additional request header to specify the authorization type, content type of the request, and other options
- Body: If we have additional request data, we can use the body option, which passes the data to the server as requested data

Other options are also supported, but you don't need to use every option along with every API call. You can just can make use of the required one in a specific HTTP call.
We can also get the response by using the .then clause along with the .catch block to manage the response and the error while making the HTTP call, like this.

```
async getUsers() {
  // With .then and .catch section
  let response = await fetch("https://jsonplaceholder.typicode.com/users")
    .then(response => {
      return response.json();
    })
    .catch(error => {
      console.log(error);
    });
 }
```

As you can see, we have two additional sections called .then and .catch. Both of these sections are used to get the response from the server, and if something goes wrong while making an HTTP call, then the appropriate error will be highlighted.
Now let's try the POST request to post the data to the server using the POST as the request type with Fetch.

```
async getUsers() {
   let body = {
    userId: 1111,
    title: "This is POST request with body",
    completed: true
  };
  fetch("https://jsonplaceholder.typicode.com/todos", {
    method: "POST",
    body: JSON.stringify(body)
  })
    .then(response => {
      let json = response.json();
      console.log(json);
      if (!response.ok) {
        throw new Error("Network response was not ok");
      }
      return response.blob();
    })
    .catch(error => {
      console.error(
        "There has been a problem with your fetch operation:",
        error
      );
    });
}
```
In this method, we have used the additional options with the URL to specify the type of request and a body attached to the request.
```
fetch("https://jsonplaceholder.typicode.com/todos", {
      method: "POST",
      body: JSON.stringify(body)
    })
```
The property method specifies that the request is the POST and we have to pass the requested data as the body of the API call. Here is the complete code snippet of the file.
```
import React, { Component } from "react";

class UsingFetch extends Component {
  constructor() {
    super();
    this.getUsers = this.getUsers.bind(this);
  }

  componentDidMount() {
    this.getUsers();
  }

  async getUsers() {
    // With error handling
    let body = {
      userId: 1111,
      title: "This is POST request with body",
      completed: true
    };
    fetch("https://jsonplaceholder.typicode.com/todos", {
      method: "POST",
      body: JSON.stringify(body)
    })
      .then(response => {
        let json = response.json();
        console.log(json);
        this.setState({ users: json })
        if (!response.ok) {
```

```
        throw new Error("Network response was not ok");
      }
      return response.blob();
    })
    .catch(error => {
      console.error(
        "There has been a problem with your fetch operation:",
        error
      );
    });
  }

  render() {
    const { users } = this.state;
    return (
      <div>
        <h3>Using Fetch in React for API call</h3>
        <hr />
        {users &&
          users.map((user, index) => {
            return <p key={user.id}>{user.name}</p>;
          })}
        <div />
      </div>
    );
  }
}

export default UsingFetch;
```
We have called the method getUsers(), which contains our API call, using the Fetch API.As soon as we get the response, we can show the list of users coming from the response.
This is how to use the Fetch API to make HTTP calls and manipulate the response based on the functional requirement.