NOVEMBER 3, 2023 / #API

How to Call an API in JavaScript – with Examples

Joan Ayebola

Calling an API (Application Programming Interface) in JavaScript is a fundamental action that web developers need to know how to perform. It allows you to fetch data from external sources and integrate it into your web applications.

In this tutorial, I'll walk you through the process of making API calls in JavaScript, step by step. By the end of this article, you'll have a

Table of Contents:

What is an API?

Before we dive into the technical details of calling an API in JavaScript, let's start with the basics. An API, or Application Programming Interface, is like a bridge that allows two different software systems to communicate with each other. It defines a set of rules and protocols for requesting and exchanging data.

APIs can be used to retrieve information from external sources, send data to external services, or perform various other actions. They are widely used in web development to access data from various online services such as social media platforms, weather data, financial information, and more.

needs. There are countless APIs available, providing data on a wide range of topics.

Some of the popular types of APIs include:

- RESTful APIs: These are widely used for simple data retrieval and manipulation. They use standard HTTP methods like GET, POST, PUT, and DELETE.
- Third-Party APIs: Many online services offer APIs that allow you to access their data, such as the Twitter API for tweets or the Google Maps API for location data.
- Weather APIs: If you need weather data, APIs like OpenWeatherMap or the WeatherAPI are good choices.
- Financial APIs: To fetch financial data like stock prices, you can use APIs like Alpha Vantage or Yahoo Finance.

For this guide, we'll use a fictional RESTful API as an example to keep things simple. You can replace it with the API of your choice.

How to Use the Fetch API for GET Requests

To make API requests in JavaScript, you can use the `fetch` API, which is built into modern browsers. It is a promise-based API that makes it easy to send HTTP requests and handle responses asynchronously.

Here's how to make a GET request using `fetch`:

```
// Make a GET request
fetch(apiUrl)
  .then(response => {
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    return response.json();
  })
  .then(data => {
    console.log(data);
  })
  .catch(error => {
    console.error('Error:', error);
  });
```

In the code above:

- We defined the API URL that we want to call.
- We used the `fetch` function to make a GET request to the API URL. The `fetch` function returns a Promise.
- The `.then()` method handles the asynchronous response from the server.
- The `response.ok` property is checked to ensure the response is valid.
- We parse the JSON data using the `response.json()` method.
- Finally, we log the data to the console, or handle any errors that may occur.

How to Handle Responses

When you make an API call, the server responds with data. How you handle this data depends on your application's requirements. In the

displaying it on a web page or storing it in a database.

Here's a modified example that displays the API data in an HTML element:

```
const apiUrl = 'https://api.example.com/data';
const outputElement = document.getElementById('output');

fetch(apiUrl)
  .then(response => {
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    return response.json();
  })
  .then(data => {
    // Display data in an HTML element
    outputElement.textContent = JSON.stringify(data, null, 2);
  })
  .catch(error => {
    console.error('Error:', error);
  });
```

In this example, we use the `outputElement` variable to select an HTML element where we want to display the data. The `textContent` property is used to update the content of that element with the JSON data.

Error Handling in API Calls

Error handling is an essential part of making API calls in JavaScript. API requests can fail for various reasons, such as network issues, server problems, or incorrect URLs.

In addition to the `catch` block, you can also check the HTTP status code using `response.status` to determine the nature of the error. Here's how you can do it:

```
const apiUrl = 'https://api.example.com/data';

fetch(apiUrl)
  .then(response => {
    if (!response.ok) {
      if (response.status === 404) {
        throw new Error('Data not found');
      } else if (response.status === 500) {
        throw new Error('Server error');
      } else {
        throw new Error('Network response was not ok');
      }
    }
    return response.json();
  })
  .then(data => {
    outputElement.textContent = JSON.stringify(data, null, 2);
  })
  .catch(error => {
    console.error('Error:', error);
  });
```

In this example, we check for specific HTTP status codes (such as 404 and 500) and provide more descriptive error messages. You can customize the error handling to suit your application's needs.

How to Make POST Requests

So far, we've focused on making GET requests, which are used to fetch data from an API. But you may also need to send data to an API, which you can do using POST requests.

```
const apiUrl = 'https://api.example.com/data';
const data = {
  name: 'John Doe',
  email: 'johndoe@example.com',
};

const requestOptions = {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify(data),
};

fetch(apiUrl, requestOptions)
  .then(response => {
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    return response.json();
  })
  .then(data => {
    outputElement.textContent = JSON.stringify(data, null, 2);
  })
  .catch(error => {
    console.error

('Error:', error);
  });
```

In this example:

- We defined the API URL and the data we want to send as an object.
- We created a `requestOptions` object that specifies the method (POST), the content type (application/json), and the data to be sent in JSON format.

The rest of the code remains similar to our previous examples, with error handling and data processing.

How to Work with API Keys

Many APIs require authentication through API keys to ensure that only authorized users can access their data. When working with APIs that require API keys, you need to include the key in your requests.

Here's an example of how to include an API key in a request:

```
const apiKey = 'your_api_key_here';
const apiUrl = 'https://api.example.com/data';

const requestOptions = {
  method: 'GET',
  headers: {
    'Authorization': `Bearer ${apiKey}`,
  },
};

fetch(apiUrl, requestOptions)
  .then(response => {
    if !response.ok) {
      throw new Error('Network response was not ok');
    }
    return response.json();
  })
  .then(data => {
    outputElement.textContent = JSON.stringify(data, null, 2);
  })
  .catch(error => {
    console.error('Error:', error);
  });
```

Make sure to replace `your_api_key_here` with your actual API key.

Asynchronous JavaScript

API calls are typically asynchronous, which means they do not block the execution of your code while waiting for a response. This is important because it allows your web application to remain responsive even when dealing with potentially slow network requests.

To handle asynchronous operations, we use promises and the `.then()` method to specify what should happen when the operation is completed. This allows the main thread of your JavaScript application to continue running other tasks while waiting for the API response.

Here's a recap of how asynchronous JavaScript works:

When you call `fetch`, it initiates an asynchronous operation and returns a promise immediately.

You use the `.then()` method to attach functions that should execute when the promise resolves successfully (with a response) or fails (with an error).

Any code outside of the `.then()` blocks can continue running while the API call is in progress.

This asynchronous behavior helps ensure that your application remains responsive and doesn't freeze while waiting for data.

Now that we've covered the basics of making API calls in JavaScript, let's explore a couple of real-world examples to see how this knowledge can be applied in practice.

Example 1: Fetching Weather Data

In this example, we'll use the OpenWeatherMap API to fetch weather data for a specific location. You can sign up for a free API key on their website.

Here's how you can make a GET request to fetch the weather data and display it on a webpage:

```
const apiKey = 'your_openweathermap_api_key';
const apiUrl = `https://api.openweathermap.org/data/2.5/weather?q=London&appi

const outputElement = document.getElementById('weather-output');

fetch(apiUrl)
  .then(response => {
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    return response.json();
  })
  .then(data => {
    const temperature = data.main.temp;
    const description = data.weather[0].description;
    const location = data.name;
    outputElement.innerHTML = `<p>Temperature in ${location}: ${temperature}°
                               <p>Weather: ${description}</p>`;
  })
  .catch(error => {
    console.error('Error:', error);
  });
```

temperature and weather description on a webpage.

Example 2: Posting a Form to a Server

Suppose you have a simple contact form on your website, and you want to send the form data to a server for processing. Here's how you can make a POST request to send the form data to a server:

HTML:

```
<form id="contact-form">
  <input type="text" name="name" placeholder="Name">
  <input type="email" name="email" placeholder="Email">
  <textarea name="message" placeholder="Message"></textarea>
  <button type="submit">Submit</button>
</form>
<div id="response-message"></div>
```

JavaScript:

```
const apiUrl = 'https://api.example.com/submit';

const contactForm = document.getElementById('contact-form');
const responseMessage = document.getElementById('response-message');

contactForm.addEventListener('submit', function (event) {
  event.preventDefault();

  const formData = new FormData(contactForm);

  const requestOptions = {
    method: 'POST',
    body: formData,
  };

  fetch(apiUrl, requestOptions)
```

```
      return response.text();
    })
    .then(data => {
      responseMessage.textContent = data;
    })
    .catch(error => {
      console.error('Error:', error);
    });
});
```

In this example, we listen for the form's submit event, prevent the default form submission, and use `FormData` to serialize the form data. We then make a POST request to the server, send the form data, and display the server's response.

Conclusion

Calling an API in JavaScript is a valuable skill for web developers, allowing you to access a wealth of data and services to enhance your web applications.

In this comprehensive guide, we covered the essential concepts and techniques, including making GET and POST requests, handling responses and errors, and working with API keys. You also saw two practical examples that demonstrate how to fetch weather data and send form data to a server.

As you continue to work with APIs in your projects, you'll encounter various APIs with their unique requirements and documentation. Remember that APIs can have rate limits, usage policies, and restrictions, so always review the API's documentation to ensure you're using it correctly and responsibly.

Joan Ayebola

HI, I am Joan, a frontend developer and technical writer who's deeply passionate about open-source technologies. With several years of experience in the industry, I have been involved in various projects, contributing code, and writing technical documentation to empower developers worldwide. When not coding or writing, I enjoy crocheting, reading and listening to podcasts.