Continue

When building web applications, it is often necessary to send data from the client to the server. One common way to accomplish this is by making a POST request. In this article, we will explore how to make a x-www-form-urlencoded POST request using the Fetch API in JavaScript. What is x-www-form-urlencoded? x-www-form-urlencoded is a format used to send data in HTTP requests. It is commonly used in HTML forms and is the default format for sending data in POST requests. In this format, the data is encoded as key-value pairs, separated by ampersands (&), and the keys and values are URL-encoded. Using the Fetch API The Fetch API provides a modern and flexible way to make HTTP requests in JavaScript. It is supported in all major browsers and allows us to easily send data to a server. To make a x-www-form-urlencoded POST request, we can use the Fetch API's fetch() function. fetch(url, { method: 'POST', headers: { 'Content-Type': 'application/x-www-form-urlencoded' }, body: 'key1=value1&key2=value2' }) .then(response => response.json()) .then(data => { // Handle the response data }) .catch(error => { // Handle any errors }); In the above example, we pass the URL of the server as the first argument to the fetch() function. The second argument is an options object where we specify the method as 'POST' and set the 'Content-Type' header to 'application/x-www-form-urlencoded'. The body of the request is a string containing the encoded key-value pairs. URL-Encoding the Data Before sending the data in the body of the request, we need to URL-encode the key-value pairs. This ensures that any special characters are properly encoded and can be safely transmitted over the network. In JavaScript, we can use the encodeURIComponent() function to perform URL encoding. const data = { key1: 'value1', key2: 'value2' }; const encodedData = Object.keys(data) .map(key => `${encodeURIComponent(key)}=${encodeURIComponent(data[key])}`) .join('&'); fetch(url, { method: 'POST', headers: { 'Content-Type': 'application/x-www-form-urlencoded' }, body: encodedData }) .then(response => response.json()) .then(data => { // Handle the response data }) .catch(error => { // Handle any errors }); In this example, we define an object data containing the key-value pairs to be sent in the request. We then use Object.keys() to get an array of the keys, and map() to iterate over the keys and values, URL-encoding them using encodeURIComponent(). Finally, we join the encoded key-value pairs with ampersands to form the body of the request. Conclusion In this article, we have learned how to make a x-www-form-urlencoded POST request using the Fetch API in JavaScript. We explored the concept of x-www-form-urlencoded, saw an example of making a POST request using the Fetch API, and learned how to URL-encode the data before sending it. By understanding and applying these concepts, you can effectively send data from the client to the server in your web applications. For further information, you may refer to the following resources: In this quick example, we will see how to send a post body in restTemplate as x-www-form-urlencoded with an example. Consider we have two parameters token and client_id that we want to send as post data as x-www-form-urlencoded. The token and client_id have some value as below. Sending post body using restTemplate as x-www-form-urlencoded using postman Let's see an example to send post body using RestTemplate and x-www-form-urlencoded import com.fasterxml.jackson.databind.JsonNode; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.beans.factory.annotation.Value; import org.springframework.http.HttpEntity; import org.springframework.http.HttpHeaders; import org.springframework.http.MediaType; import org.springframework.http.ResponseEntity; import org.springframework.stereotype.Component; import org.springframework.util.LinkedMultiValueMap; import org.springframework.util.MultiValueMap; import org.springframework.web.client.RestTemplate; @Component public class RestTemplateClient { @Value("${rest.client.url}") private String url; @Autowired private RestTemplate restTemplate; public ResponseEntity callRestClient() { String token = "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.Jf36POk6yJV_adQssw5c"; String clientId = "87ooooohhja22121hhahjrtyyysdddyyuu"; MultiValueMap map = new LinkedMultiValueMap(); map.add("token", token); map.add("client_id", clientId); HttpHeaders headers = new HttpHeaders(); headers.setContentType(MediaType.APPLICATION_FORM_URLENCODED); HttpEntity requestBodyFormUrlEncoded = new HttpEntity(map, headers); ResponseEntity responseEntity = null; try { responseEntity = restTemplate.postForEntity(url, requestBodyFormUrlEncoded, JsonNode.class); } catch (Exception e) { e.printStackTrace(); } return responseEntity; } } In the above example, we are reading the rest URI using @Value annotation. We have created a RestTemplate reference using @Autowired annotation. We are using MultiValueMap and LinkedMultiValueMap for storing requested x-www-form-urlencoded. We need to make sure we set x-www-form-urlencoded in headers. headers.setContentType(MediaType.APPLICATION_FORM_URLENCODED); We are using restTemplate.postForEntity() method restTemplate.postForEntity(url, requestBodyFormUrlEncoded, JsonNode.class); That's all about How to send post body using restTemplate as x-www-form-urlencoded. Other examples. When working with APIs or web services in C#, sending POST requests is a common task. In this guide, we will explore how to use the WebClient class to make POST requests while encoding data in the x-www-form-urlencoded format. Setting up WebClient Before we can start making POST requests, we need to create an instance of WebClient. Here's how you can do it: using System.Net; WebClient client = new WebClient(); Sending a POST Request To send a POST request with WebClient, we need to specify the URL we want to send the request to and the data we want to include in the request body. Let's see an example: string url = " ; string postData = "key1=value1&key2=value2"; client.Headers[HttpRequestHeader.ContentType] = "application/x-www-form-urlencoded"; string response = client.UploadString(url, "POST", postData); Console.WriteLine(response); In the example above, we set the ContentType header to application/x-www-form-urlencoded and then use the UploadString method to send the POST request with the encoded data. Handling Responses When sending POST requests, it's important to handle the response from the server. Here's how you can read and process the response: string response = client.UploadString(url, "POST", postData); Console.WriteLine(response); Best Practices Always encode your data in x-www-form-urlencoded format to ensure compatibility with server-side processing. Handle exceptions that may occur during the request, such as network errors or server timeouts. Consider using HttpClient for more advanced scenarios, as WebClient is a simpler but less flexible option. By following these guidelines and examples, you can effectively make POST requests with WebClient in C# using the x-www-form-urlencoded format. Happy coding! Baeldung Pro – NPI EA (cat = Baeldung) Baeldung Pro comes with both absolutely No-Ads as well as finally with Dark Mode, for a clean learning experience: >> Explore a clean Baeldung Once the early-adopter seats are all used, the price will go up and stay at $33/year. Partner – Microsoft – NPI EA (cat = Baeldung) Azure Container Apps is a fully managed serverless container service that enables you to build and deploy modern, cloud-native Java applications and microservices at scale. It offers a simplified developer experience while providing the flexibility and portability of containers. Of course, Azure Container Apps has really solid support for our ecosystem, from a number of build options, managed Java components, native metrics, dynamic logger, and quite a bit more. To learn more about Java features on Azure Container Apps, visit the documentation page. You can also ask questions and leave feedback on the Azure Container Apps GitHub page. Partner – Microsoft – NPI EA (cat= Spring Boot) Azure Container Apps is a fully managed serverless container service that enables you to build and deploy modern, cloud-native Java applications and microservices at scale. It offers a simplified developer experience while providing the flexibility and portability of containers. Of course, Azure Container Apps has really solid support for our ecosystem, from a number of build options, managed Java components, native metrics, dynamic logger, and quite a bit more. To learn more about Java features on Azure Container Apps, you can get started over on the documentation page. And, you can also ask questions and leave feedback on the Azure Container Apps GitHub page. Partner – Orkes – NPI EA (cat=Spring) Modern software architecture is often broken. Slow delivery leads to missed opportunities, innovation is stalled due to architectural complexities, and engineering resources are exceedingly expensive. Orkes is the leading workflow orchestration platform built to enable teams to transform the way they develop, connect, and deploy applications, microservices, AI agents, and more. With Orkes Conductor managed through Orkes Cloud, developers can focus on building mission critical applications without worrying about infrastructure maintenance to meet goals and, simply put, taking new products live faster and reducing total cost of ownership. Try a 14-Day Free Trial of Orkes Conductor today. Partner – Orkes – NPI EA (tag=Microservices) Modern software architecture is often broken. Slow delivery leads to missed opportunities, innovation is stalled due to architectural complexities, and engineering resources are exceedingly expensive. Orkes is the leading workflow orchestration platform built to enable teams to transform the way they develop, connect, and deploy applications, microservices, AI agents, and more. With Orkes Conductor managed through Orkes Cloud, developers can focus on building mission critical applications without worrying about infrastructure maintenance to meet goals and, simply put, taking new products live faster and reducing total cost of ownership. Try a 14-Day Free Trial of Orkes Conductor today. eBook – Mockito – NPI EA (tag = Mockito) Mocking is an essential part of unit testing, and the Mockito library makes it easy to write clean and intuitive unit tests for your Java code. Get started with mocking and improve your application tests using our Mockito guide: Download the eBook eBook – Java Concurrency – NPI EA (cat=Java Concurrency) Handling concurrency in an application can be a tricky process with many potential pitfalls. A solid grasp of the fundamentals will go a long way to help minimize these issues. Get started with understanding multi-threaded applications with our Java Concurrency guide: >> Download the eBook eBook – Reactive – NPI EA (cat=Reactive) Spring 5 added support for reactive programming with the Spring WebFlux module, which has been improved upon ever since. Get started with the Reactor project basics and reactive programming in Spring Boot: >> Join Pro and download the eBook eBook – Java Streams – NPI EA (cat=Java Streams) Since its introduction in Java 8, the Stream API has become a staple of Java development. The basic operations like iterating, filtering, mapping sequences of elements are deceptively simple to use. But these can also be overused and fall into some common pitfalls. To get a better understanding on how Streams work and how to combine them with other language features, check out our guide to Java Streams: >> Join Pro and download the eBook eBook – HTTP Client – NPI EA (cat=Http Client-Side) Get the most out of the Apache HTTP Client Download the E-book eBook – Persistence – NPI EA (cat=Persistence)Course – LS – NPI EA (cat=Jackson) Get started with Spring and Spring Boot, through the Learn Spring course: >> LEARN SPRING Course – LSS – NPI EA (cat=Spring Security) Yes, Spring Security can be complex, from the more advanced functionality within the Core to the deep OAuth support in the framework. I built the security material as two full courses - Core and OAuth, to get practical with these more complex scenarios. We explore when and how to use each feature and code through it on the backing project. You can explore the course here: >> Learn Spring Security Course – All Access – NPI EA (cat= Spring) All Access is finally out, with all of my Spring courses. Learn JUnit is out as well, and Learn Maven is coming fast. And, of course, quite a bit more affordable. Finally. >> GET THE COURSE Course – LSD – NPI EA (tag=Spring Data JPA) Spring Data JPA is a great way to handle the complexity of JPA with the powerful simplicity of Spring Boot. Get started with Spring Data JPA through the guided reference course: >> CHECK OUT THE COURSE Partner – LambdaTest – NPI EA (cat=Testing) End-to-end testing is a very useful method to make sure that your application works as intended. This highlights issues in the overall functionality of the software, that the unit and integration test stages may miss. Playwright is an easy-to-use, but powerful tool that automates end-to-end testing, and supports all modern browsers and platforms. When coupled with LambdaTest (an AI-powered cloud-based test execution platform) it can be further scaled to run the Playwright scripts in parallel across 3000+ browser and device combinations: >> Automated End-to-End Testing With Playwright Course – Spring Sale 2025 – NPI EA (cat= Baeldung) Yes, we're now running our Spring Sale. All Courses are 25% off until 26th May, 2025: >> EXPLORE ACCESS NOW Course – Spring Sale 2025 – NPI (cat=Baeldung) Yes, we're now running our Spring Sale. All Courses are 25% off until 26th May, 2025: >> EXPLORE ACCESS NOW Postman provides multiple ways to interact with an API or server request with different types of body parameters. These represent different ways of sending data through an HTTP request to the API. In this tutorial, we'll explore the differences between using form-data, x-www-form-urlencoded, and raw for our request body. 2. form-data Form-data represents the data sent from website forms to APIs as part of multipart/form-data. The form-data option in Postman simulates filling out a form on a website and submitting it. We can edit the form data and let it set the different key/value pairs by transforming the key-value editor in the data. This can also be used for attaching files to the keys as well. However, we should note that using HTML5 means that the files aren't in any history or collections. Therefore, we must select the file again at the time of sending the request body. Also, uploading multiple files with their content type isn't supported by Postman. Note that Postman will persist the file paths with subsequent use, i.e., when we repeatedly make an API call that sends the same file onto the server. This helps in running collections with multiple requests to upload files. Let's see what using form-data looks like in Postman: To send data as form-data, Postman generates a request like this: curl -X POST -F "key=value" -F "image=value" 3. x-www-form-urlencoded The URL-encoded data sends encoded data to the server, and uses the same encoding as that of the URL parameters. To use it, we need to select the x-www-form-urlencoded tab in the body of the request. We need to enter the key-value pairs for sending the request body to the server, and Postman will encode the desired data before sending it. Postman encodes both the key and the value. Note that it can't be used for encoding files, so we need to manually do it ourselves. However, it can only encode the request body data or the URL parameters. This is also known as the default content type. All the forms submitted with this content type follow the below encoding pattern: The control names and values are escaped. All space characters will be replaced by the '+' symbol, and reserved characters follow the RFC 17.38 notations. An equal symbol, '=', is used for separating the key and value, and the key/value pairs use '&' to be separated from each other. Let's take a look at the x-www-form-urlencoded tab in Postman: To send data as x-www-form-urlencoded, Postman generates a request like this: curl -X POST -d "phone=+179822xxxxx" 4. raw As the name suggests, raw data can consist of anything. Postman doesn't touch the raw string or make any type of alterations to it. The string added into the raw editor goes unmodified, except for replacing the defined environment variables. This editor lets us set different formatting styles supported by the Postman, along with the correct header that's required to be sent with the raw body. The following types are supported: Text Javascript JSON HTML XML We can also set these Content-Types manually into our request body: To send raw JSON data, Postman generates a request like this: curl -X POST -H "Content-Type: application/json" -d '{"key1": "value1", "key2": "value2"}' 5. Difference BETWEEN form-data, x-www-form-urlencoded and raw Here's a detailed breakdown of the three primary content types: form-data, x-www-form-urlencoded, and raw: 5.1. form-data This content type is used for sending data as multipart/form-data. It is particularly useful for uploading files or sending large amounts of binary or non-ASCII data. However, form-data does not handle encoding for files and other complex structures in the request body itself. 5.2. x-www-form-urlencoded This is the default content type for simple text form submissions. It encodes the data as key-value pairs and is similar to URL query parameters. This format is straightforward and efficient for basic form submissions. However, it has a size limit and is best used for simpler requests where the data is primarily text. 5.3. raw The raw option provides the most flexibility, allowing us to send any type of data without modifications. It supports various content types, such as JSON, XML, and plain text. This format is ideal for sending complex or custom-formatted data, including JavaScript functions and markup languages. Unlike form-data and x-www-form-urlencoded, raw data is sent as-is, with no additional encoding or alteration by Postman. The following table highlights their key characteristics and use cases: Aspect form-data x-www-form-urlencoded raw Encoding Multipart form-data URL encoding No encoding (data sent as-is) File Uploads Supports file uploads Doesn't support file uploads Not applicable Data Types Text and binary data (files) Key-value pairs (text only) Custom data formats (e.g., JSON, XML, plain text) Use Cases Uploading files, complex forms Simple form submissions Sending JSON objects, XML documents, scripts Size Limitation No significant size limitation Limited by URL length and server configuration Generally unrestricted by size limitations 6. Conclusion In this article, we examined a few of the request body data types supported by Postman. We also outlined the differences between form-data, x-www-form-urlencoded and raw in Postman. However, we only gained a basic understanding of the requests. If you want to understand these request body types more deeply, you can explore the Postman online documentation. How can I send POST requests with x-www-form-urlencoded parameters using the new HTTP client introduced in Java 11? Copied import java.net.URI; import java.net.http.HttpClient; import java.net.http.HttpRequest; import java.net.http.HttpResponse; import java.net.http.HttpRequest.BodyPublishers; String url = " ; String form = "param1=value1¶m2=value2"; HttpRequest request = HttpRequest.newBuilder() .uri(URI.create(url)) .header("Content-Type", "application/x-www-form-urlencoded") .POST(BodyPublishers.ofString(form)) .build(); HttpClient client = HttpClient.newHttpClient(); client.sendAsync(request, HttpResponse.BodyHandlers.ofString()) .thenApply(HttpResponse::body) .thenAccept(System.out::println); Java 11 introduced a new HTTP client that simplifies the process of sending HTTP requests. This modern client provides a cleaner API and facilitates sending various types of requests, including POST requests with x-www-form-urlencoded parameters. Copied import java.net.URI; import java.net.http.HttpClient; import java.net.http.HttpRequest; import java.net.http.HttpResponse; import java.net.http.HttpRequest.BodyPublishers; String url = " ; String form = "param1=value1¶m2=value2"; HttpRequest request = HttpRequest.newBuilder() .uri(URI.create(url)) .header("Content-Type", "application/x-www-form-urlencoded") .POST(BodyPublishers.ofString(form)) .build(); HttpClient client = HttpClient.newHttpClient(); client.sendAsync(request, HttpResponse.BodyHandlers.ofString()) .thenApply(HttpResponse::body) .thenAccept(System.out::println); Causes In Java 11, the new HttpClient class replaces the old HttpURLConnection class, providing a more efficient and user-friendly API. The 'x-www-form-urlencoded' format is commonly used in HTML forms and is the standard way of encoding form data for submission via POST. Solutions To send a POST request with x-www-form-urlencoded parameters, utilize the HttpRequest class to construct your request and the BodyPublishers class to send form data. Set the 'Content-Type' header to 'application/x-www-form-urlencoded' to ensure the server accurately interprets the data. Mistake: Forgetting to set the 'Content-Type' header, which results in the server not processing the data correctly. Solution: Always set the 'Content-Type' header to 'application/x-www-form-urlencoded'. Mistake: Incorrectly formatting the form data string causing the server to receive unexpected values. Solution: Ensure that your form data is URL-encoded, e.g., replace spaces with '+' and properly encode special characters. Java 11 HTTP Client send POST requests x-www-form-urlencoded Java HTTP requests HTTP Client tutorial © Copyright 2025 - CodingTechRoom.com I'm planning a new API and was curious about what kind of requests I should accept. The most common Content-Types I've seen for popular modern REST APIs include: application/x-www-form-urlencoded application/json Shout-out to application/graphql too, but this article will focus on the two above :) Of course, when just starting out it's easy to accept both and forget about it. For example... import express from 'express'; import bodyParser from 'body-parser'; const app = express(); // application/x-www-form-urlencoded app.use(bodyParser.urlencoded({extended: false})); // application/json app.use(bodyParser.json()); export default app; A minimal example of a Node.js webserver accepting both request body types I've done this many times in the past. However, I wanted to set out and discover the true pros and cons of each. I wanted to understand why someone would choose one over the other. Initial bias Before I started my search, I wanted to keep my biases in check. What do I currently believe and what am I probably predisposed to want to validate? application/json is beginner-friendly...mostly axios and superagent, two of the more popular npm HTTP libraries, work with JSON bodies by default. To send as application/x-www-form-urlencoded, you have to add additional configuration options. For example, here's how you might do it with axios. Leveraging a querystring encoding library is necessary: import axios from 'axios'; import qs from 'qs'; axios({ method: 'POST', url: '/resource', data: qs.stringify({ foo: 'bar', }), }); It's not really a significant amount of boilerplate, but it's something I've run into in the past. On the other hand, tools like Postman are much easier to work with when using application/x-www-form-urlencoded: Toggling and descriptions? Fantastic! Whereas application/json doesn't allow toggling nor describing individual keys: Yeah yeah, I know. Comments aren't valid JSON anyway. But come on! URL encoding is used by some of the best APIs out there I commonly see Stripe referred to as the gold standard as far as API quality. Guess what encoding they use? The Stripe API is organized around REST. Our API has predictable resource-oriented URLs, accepts form-encoded request bodies, returns JSON-encoded responses, and uses standard HTTP response codes, authentication, and verbs -- Stripe Docs And it's not just them. Twilio, the market leader for SMS, is on the same page: Creating or updating a resource involves performing an HTTP PUT or HTTP POST to a resource URI. In the PUT or POST, you represent the properties of the object you wish to update as form urlencoded key/value pairs. Don't worry, this is already the way browsers encode POSTs by default. But be sure to set the HTTP Content-Type header to "application/x-www-form-urlencoded" for your requests if you are writing your own client. -- Twilio Docs These folks have got to know what they're doing, right? Okay, so some positive and negative biases in regards to both approaches. At this point, I started Googling around. My main search terms: application/json vs application/x-www-form-urlencoded json vs form differences performance best practices What I learned Essentially, there's a whitelist of request properties and values. As long as your HTTP requests stay within this whitelist, no preflight requests are sent. The application/json Content-Type isn't on the list, so using this for requests will result in preflight requests. You can read more about this below: The whitelist is fairly restrictive. No Authorization headers, Cache-Control, Keep-Alive, etc. I'd recommend not trying to circumvent CORS in this regard unless you find there's a significant performance opportunity and minimal additional code complexity. URL encoded arrays can be a nightmare Sending any kind of complex data with application/x-www-form-urlencoded has historically been...not ideal. The following are two patterns I've seen for sending an array called a with members b and c. Neither of them are particularly great: a[]=b&a[]=c a[0]=b&a[1]=c For larger payloads, you may be saving bytes with JSON as well. tl;dr If you can do so securely, support both content types. Always sanitize user input, don't bite off more than you can chew, etc. If you must choose one request body for your API, go with whatever works for your use case. For complex data (especially array/nested object parameters) or if you already return JSON and want to be consistent: consider application/json. For mostly flat param trees, application/x-www-form-urlencoded is tried and tested. After all, it's the default for browsers! Further reading When making HTTP requests in Python, the requests library is a commonly-used tool. One common use case is to send POST requests with an x-www-form-urlencoded body. This format is used when sending data in the body of the request as key-value pairs separated by an equals sign.Using Requests Library with x-www-form-urlencoded BodyTo send a POST request with an x-www-form-urlencoded body using the requests library, you can use the data parameter for the request. The data should be provided in the form of a dictionary with keys and values for each field you want to send. import requests url = ' data = { 'name': 'John Doe', 'email': '' } response = requests.post(url, data=data) In the example above, we're sending a POST request to with two fields "name" and "email" and their corresponding values "John Doe" and "".Using urllib.parse to Encode DataIf you don't have a dictionary of key-value pairs and instead have a string of data you want to send, you can use the urllib.parse library to encode it in the x-www-form-urlencoded format. import requests from urllib.parse import urlencode url = ' data = 'name=John+Doe&email=johndoe%40example.com' headers = {'Content-Type': 'application/x-www-form-urlencoded'} response = requests.post(url, data=data, headers=headers) The example above shows how to use urlencode to encode the data before sending a POST request. We're also including the Content-Type header to specify that we're sending an x-www-form-urlencoded body.ConclusionSending POST requests with an x-www-form-urlencoded body is a common task in web development. Using the requests library and the data parameter, it's easy to send key-value pairs in the body of a request. If you have a string of data, you can use the urllib.parse library to encode it before sending the request.