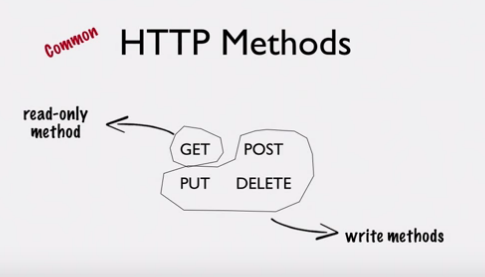
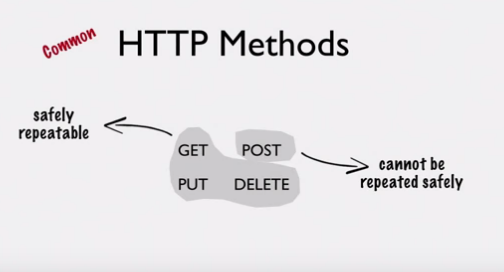
<https://javabrains.io/courses/javaee_jaxrs/lessons/Idempotence-In-HTTP-Methods>





Notes

Idempotence. Yes, that's a word. And it's an important property of HTTP methods according to the specifications.

PUT vs POST

When I was learning about RESTful web services, one thing that confused me was the difference between PUT and POST. Like we've already seen, you use PUT when you want to update an existing resource, and POST when you want to create a new resource. But if you search online, you will very likely find a lot of resources that contradict each other. Some are plain wrong, while others tell you the right thing to do, but do not explain why. I'll try to explain this difference and hopefully, it'll be clear to you by the time you are done with this tutorial.

Method classification

There are two ways in which we can classify these 4 popular HTTP methods: GET, PUT, POST and DELETE. The GET method is a *read-only* method; it lets you read information. But the methods PUT, POST and DELETE are *write* methods; they change something on the server. They either create, update or delete, but they all cause something to change on the server.

So, it is safe to assume that you can make a GET request as many times as you want without having any *impact* on the server. You should never have a GET method do things like updates or deletes.

For example: GET on /messages/20/delete. **Never do this!**

Nothing changes when you do a GET, so it's safe to make multiple requests and not worry about the side effect. But how about PUT, POST and DELETE? Since they are methods that *write* to the server, you obviously cannot make those calls multiple times! Or can you?

Just because an operation is not read-only, it doesn't automatically mean that it cannot be duplicated.

Take an example of a Java assignment statement. Assume count is an integer variable

count = 100;

This is definitely not a read-only operation. This statement writes a value 100 to the variable count. However, if you were to repeat this operation three times, lines 2 and 3 do not really do anything. Well, maybe they do write the value to the variable, but for all practical purposes, they do not have any effect.

count = 100;

count = 100;

count = 100;

This nature of some operations that let them be *repeatable* is important in HTTP methods. Like we saw, GET is clearly a repeatable operation, because it is read-only.

Let's take DELETE. Say you make a DELETE request to /messages/10. This deletes message ID 10. Say you make the same call again. Well, message 10 is already deleted. So nothing happens. While it isn't really required or desirable to make multiple DELETE calls to the same resource, you can see that it is at least not a problem. There are no unwanted side effects if you were to make a duplicate call by mistake.

Ok, how about PUT. Say you make a PUT request to /messages/20 with some message text in the request body. This is going to replace whatever message ID 20 was with this new message text that's being sent in the request body. Say you make the exact same call again. Message ID 20 is again replaced with the exact same message text again. Make the same request the third time, and the result is the same. Guess what? Even a PUT is *safe* when it comes to making multiple calls. If you were to accidentally repeat a PUT request, well, don't worry about it. The final saved message remains the same after every request.

The problem, however, is with the POST request. If you were to make a POST request to /messages, you create a new message. Say you forgot you made a POST request, and you issued the request again, and now you've actually created a duplicate message. Repeat that call, and you get another message! So every time a POST request is made, something new happens. This is clearly not a *safe* method to make multiple calls with. Every duplicate call changes things by creating a new resource. It's definitely not a good idea to make multiple POST calls, unless you actually need multiple resources.

So, we have another way to classify HTTP methods into two types. One set of methods, including GET, PUT and DELETE, are *safe* for make repeated calls without worrying about the impact. They may not all be read-only. But they do not cause side-effects if called multiple times. And the other category, consisting of POST which you have to be very careful with, and make only as many calls as you need. The methods in the first set are called **idempotent** methods. GET, PUT and DELETE are idempotent. POST is **non-idempotent**.

Here's the Wikipedia definition of *idempotence*

Idempotence is the property of certain operations in mathematics and computer science, that can be applied multiple times without changing the result beyond the initial application.

The HTTP specification requires GET, PUT and DELETE methods to always be idempotent. If a client makes a request with one of these methods, they do not have to worry about making duplicate requests. But if they are making a POST request, they *cannot* safely make duplicate requests without any side effects.

Which is why resource creation should be a POST method. Because resource creation requests are not idempotent. Which is because multiple requests to create resources results in multiple resources. But updating a resource, like we saw, can be called multiple times safely. Which is why update requests ideally use the HTTP PUT method, which is supposed to be idempotent as per the specification.

The *only* way you can safely use PUT for creating a new resource is in scenarios where the client specifies the new ID of the resource being created. In which case, the client sends the request for creating a resource to the actual instance resource URL that includes the ID. If you were to implement this, then resource creation request is idempotent. Think about it. If you repeat the request, since it has the ID in it, it doesn't create a new resource. Perhaps, the resource with the ID is re-created or updated. This is the only scenario where you can use PUT for creating resources. But in most cases, when you have the server creating IDs and you issue a create resource request to the collection URI, you'd want to use POST.

Like I've mentioned before, these methods have standard meanings. The fact that this is a standard means that if you ignore it when implementing your APIs, you'll confuse your clients or cause their code to function improperly. Also, a common thing that many APIs do is cache some of their GET responses. When a client makes a GET request, it also updates the cache, and another GET request to that same resource URI within a certain period of time will be served directly from the cache, thereby increasing performance. This works only because GET doesn't change anything on the server, so it is *cacheable*. You can definitely build an API that creates new resources when your clients call GET, but if you do that, you'll not have many clients using your API for long!

On the other hand, if you choose proper methods, your clients can build safeguards to make sure duplicate requests do not happen. Take the example of a browser refresh button. Every browser has a refresh or reload button that does a very simple function: resend the last HTTP request that was made by the browser. If the last request happens to be an idempotent request like a GET, the browser just goes ahead and resends the request when you hit refresh. But if it was a POST, like after you've submitted a form, if you hit refresh, the browser warns you with a message that says something like "You've already submitted this data before. Are you sure you wish to resubmit?". This is simply the browser protecting you from making a duplicate non-idempotent request. So, it pays to generally use the right HTTP method for the right operation.

In this tutorial, you learned about what idempotent and non-idempotent requests are. Make sure you keep these concepts in mind when you choose HTTP methods for your APIs.