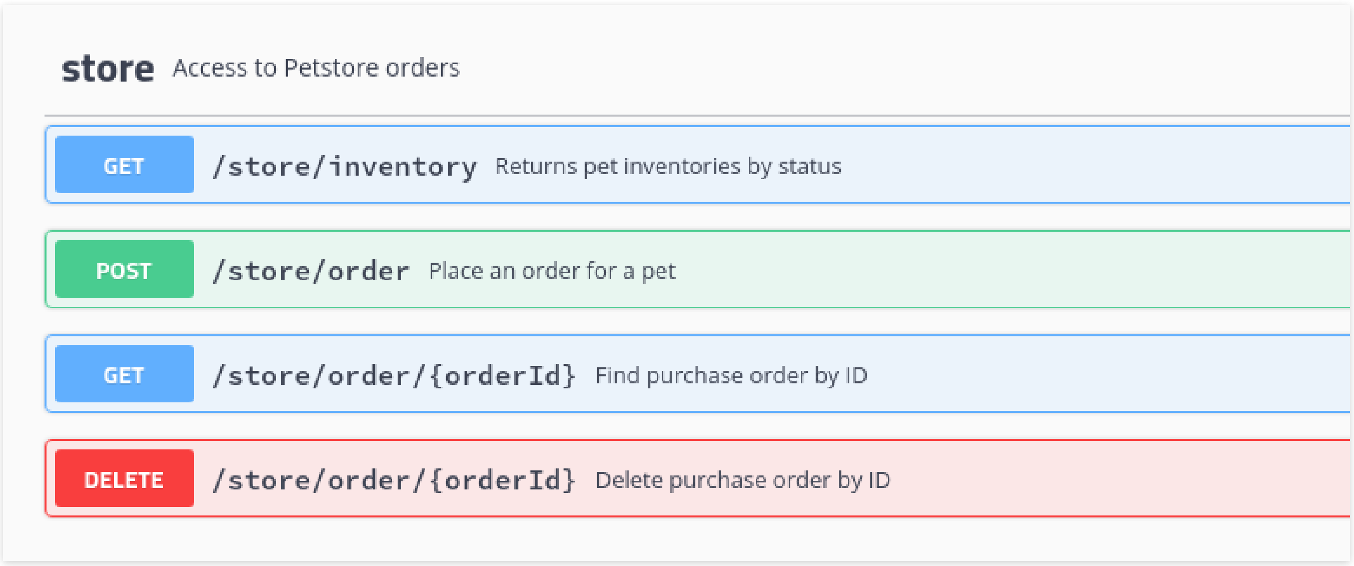
7 HTTP methods every web developer should know and how to test them

06/5/2017 Engineering Cody Reichert

Ever wondered what the difference is between `GET` and `POST` requests, or when to use `PUT`? You're not alone. Having a basic understanding of the different HTTP methods, or *verbs*, an API supports is an helpful knowledge when **exploring and testing APIs**.



store Access to Petstore orders

GET /store/inventory Returns pet inventories by status

POST /store/order Place an order for a pet

GET /store/order/{orderId} Find purchase order by ID

DELETE /store/order/{orderId} Delete purchase order by ID

In this post, I'll discuss how each HTTP method is used and how to **incorporate them in your API testing**.

HTTP Methods

- `GET`
- `POST`
- `PUT`
- `HEAD`
- `DELETE`
- `PATCH`
- `OPTIONS`

GET

`GET` requests are the most common and widely used methods in APIs and websites. Simply put, the GET method is used to **retreive data from a server at the specified resource**. For example, say you have an API with a `/users` endpoint. Making a GET request to that endpoint should return a list of all available users.

Since a GET request is only requesting data and not modifying any resources, it's considered a safe and idempotent method.

Testing an API with GET requests

When you're creating tests for an API, the `GET` method will likely be the most frequent type of request made by consumers of the service, so it's important to **check every known endpoint with a GET request**.

At a basic level, these things should be validated:

- Check that a valid GET request returns a `200` status code.
- Ensure that a GET request to a specific resource returns the correct data. For example, `GET /users` returns a list of users.

GET is often the **default method in HTTP clients**, so creating tests for these resources should be simple with any tool you choose.

POST

In web services, `POST` requests are used to **send data to the API server** to create or update a resource. The data sent to the server is stored in the request body of the HTTP request.

The simplest example is a contact form on a website. When you fill out the inputs in a form and hit *Send*, that data is put in the **response body** of the request and sent to the server. This may be JSON, XML, or query parameters (there's plenty of other formats, but these are the most common).

It's worth noting that a `POST` request is **non-idempotent**. It mutates data on the backend server (by creating or updating a resource), as opposed to a `GET` request which does not change any data. Here is a great explanation of idempotentcy.

Testing an API with POST requests

The second most common HTTP method you'll encounter in your API tests is `POST`. As mentioned above, `POST` requests are used to **send data to the API server** and create or update a resource. Since POST requests modify data, it's important to **have API tests for all of your POST methods**.

Here are some tips for testing POST requests:

- Create a resource with a `POST` request and ensure a `200` status code is returned.
- Next, make a `GET` request for that resource, and ensure the data was saved correctly.
- Add tests that ensure `POST` requests **fail** with incorrect or ill-formatted data.

For some more ideas on common API testing scenarios, check out this post.

PUT

Simlar to POST, `PUT` requests are used to send data to the API to **update or create a resource**. The difference is that **PUT requests are idempotent**. That is, calling the same PUT request multiple times **will always produce the same result**. In contrast, calling a POST request repeatedly make have side effects of creating the same resource multiple times.

Generally, when a `PUT` request *creates* a resource the server will respond with a `201` (`Created`), and if the request *modifies* existing resource the server will return a `200` (`OK`) or `204` (`No Content`).

Testing an API with PUT requests

Testing an APIs `PUT` methods is very similar to testing POST requests. But now that we know the difference between the two (idempotency), we can **create API tests to confirm this behavior**.

Check for these things when testing PUT requests:

- Repeatedly calling a `PUT` request always returns the same result (idempotent).
- The proper status code is returned when creating and updating a resource (eg, `201` or `200` / `204`).
- After updating a resource with a `PUT` request, a `GET` request for that resource should return the correct data.
- `PUT` requests should fail if invalid data is supplied in the request -- **nothing should be updated**.

PATCH

A `PATCH` request is one of the lesser-known HTTP methods, but I'm including it this high in the list since it is similar to POST and PUT. The difference with `PATCH` is that you **only apply partial modifications to the resource**.

The difference between PATCH and PUT, is that a **PATCH request is non-idempotent** (like a POST request).

To expand on partial modification, say you're API has a `/users/{{userid}}` endpoint, and a user has a *username*. With a PATCH request, **you may only need to send the updated username**in the request body - as opposed to POST and PUT which require the full user entity.

Testing an API with PATCH requests

Since the `PATCH` method is so simlar to POST and PUT, many of **the same testing techniques apply.** It's still important to validate the behavior of any API endpoints that accept this method.

What to look for when testing PATCH requests:

- A successful `PATCH` request should return a `2xx` status code.

- `PATCH` requests should fail if invalid data is supplied in the request -- **nothing should be updated**.

The semantics of PATCH requests will largely depend on the specific API you're testing.

DELETE

The `DELETE` method is exactly as it sounds: **delete the resource at the specified URL**. This method is one of the more common in RESTful APIs so it's good to know how it works.

If a new user is created with a POST request to `/users`, and it can be retrieved with a `GET` request to `/users/{{userid}}`, then making a `DELETE` request to `/users/{{userid}}` will completely remove that user.

Testing an API with DELETE requests

`DELETE` requests should be heavily tested since they generally remove data from a database. Be careful when testing DELETE methods, make sure you're using the correct credentials and not testing with real user data.

A **typical test case for a DELETE request** would look like this:

1. Create a new user with a `POST` request to `/users`
2. With the user id returned from the `POST`, make a `DELETE` request to `/users/{{userid}}`
3. A subsequent `GET` request to `/users/{{userid}}` should return a 404 not found status code.

In addition, sending a DELETE request to an unknown resource should return a non-200 status code.

HEAD

The `HEAD` method is almost identical to `GET`, **except without the response body**. In other words, if `GET /users` returns a list of users, then `HEAD /users` will make the same request but won't get back the list of users.

HEAD requests are **useful for checking what a GET request will return** before actually making a GET request -- like before downloading a large file or response body. Learn more about HEAD requests on MDN.

It's worth pointing out that not every endpoint that supports GET will support HEAD - it completely depends on the API you're testing.

Testing an API with HEAD requests

Making API requests with `HEAD` methods is actually an effective way of simply **verifying that a resource is available**. It is good practice to have a test for HEAD requests everywhere you have a test for GET requests (as long as the API supports it).

Check these things when testing an API with HEAD requests:

- Verify and check HTTP headers returned from a HEAD request
- Make assertions against the status code of HEAD requests
- Test requests with various query parametesr to ensure the API responds

Another useful case for `HEAD` requests is API smoke testing - **make a HEAD request against every API endpoint** to ensure they're available.

OPTIONS

Last but not least we have `OPTIONS` requests. OPTIONS requests are one of my favorites, though not as widely used as the other HTTP methods. In a nutshell, an OPTIONS request should **return data describing what *other* methods and operations the server supports** at the given URL.

OPTIONS requests are more loosely defined and used than the others, making them a good candidate to **test for fatal API errors**. If an API isn't expecting an OPTIONS request, it's good to put a test case in place that verifies failing behavior.

Testing an API with OPTIONS requests

Testing an `OPTIONS` request is dependent on the web service; whether or not it supports that and what is supposed to return will **define how you should test it**.

How to validate an endpoint using OPTIONS:

- Primarily, check the response headers and status code of the request
- Test endpoints that don't support OPTIONS, and ensure they fail appropriately

More resources

What I've discussed above is just a starting point for digging in to HTTP methods and testing various resources of an API. It also assumes a mostly ideal case - in the real world, APIs are not as structured as the examples above. This makes testing various methods against an API an **effective way to find unexpected bugs**.

Resources

- HTTP request methods (Wikipedia)
- HTTP status codes (Wikipedia)
- 4 common API errors and how to test them

:: Cody Reichert