API TESTING

Welcome to this course about API testing. This course is placing relatively high demands on its participants, since the topic is one of the most challenging testing categories.

API testing differs from regular testing or bug hunts in that you will not be using a web browser or mobile device for testing, but you will have to use specialized client applications like *Postman* instead. Where all requests to the customer's server will be sent by you directly, and not from a web page.

API is the acronym for *Application Programming Interface*, which describes a set of well-defined instructions created for two **applications** communicating with each other.

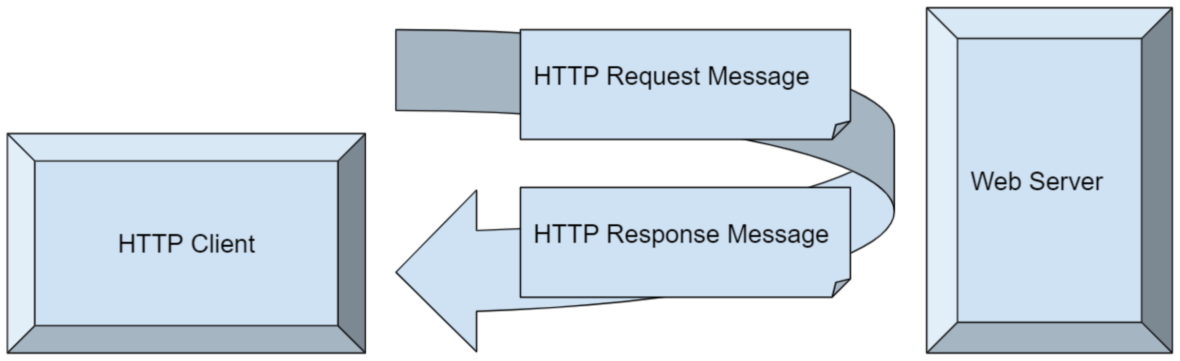
Topics

- Http Basics
- Postman - an API testing tool
    - Overview
    - Collections
    - Request methods
    - Authentication methods
- What to test for – and how
    - Common API- types
    - API description languages
- Other interesting API testing tools and technology

Introduction to HTTP Basics

Hypertext Transfer Protocol (HTTP) is the foundation of data communication for the World Wide Web, and therefore the most popular application protocol used in the Internet.

It functions as a stateless, asymmetric request-response protocol. The HTTP client submits an HTTP *request* message to the *server*. The server returns a *response* message to the client:



HTTP resources are uniquely identified by URLs (Uniform Resource Locators) in the form of *http* and *https* URIs (Uniform Resource Identifiers).

URLs have the following syntax:

protocol://hostname:port/path-and-resource-name

Protocol: The application protocol. Usually *https* or *http* in Web APIs.

Hostname: A domain name to identify a host computer (the web server).

Port: The TCP port number on which the host is listening for client-requests. If omitted it will be ***443*** for https-requests, and ***80*** for http-requests by default.

Path and resource name: consists of a sequence of path segments separated by a forward slash (/), and the requested resource's name, and may include the file-extension.

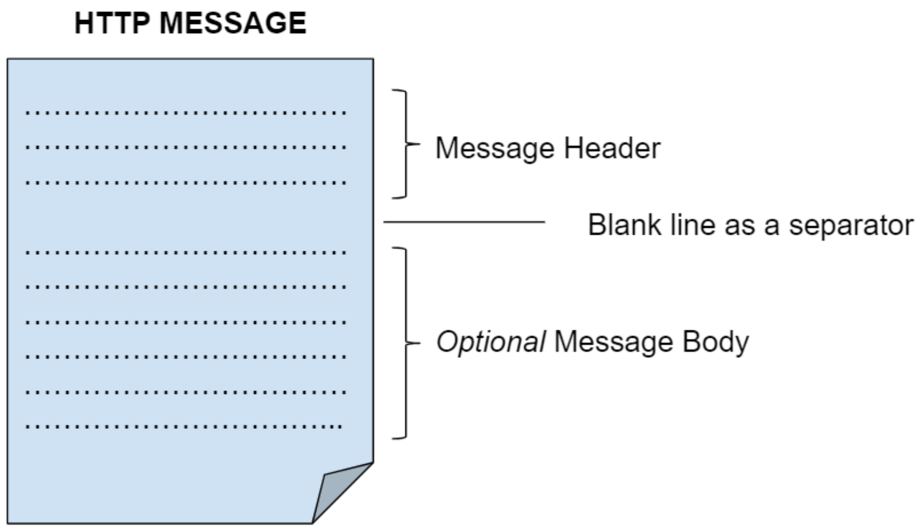
Example:

https://www.test.com/test/1

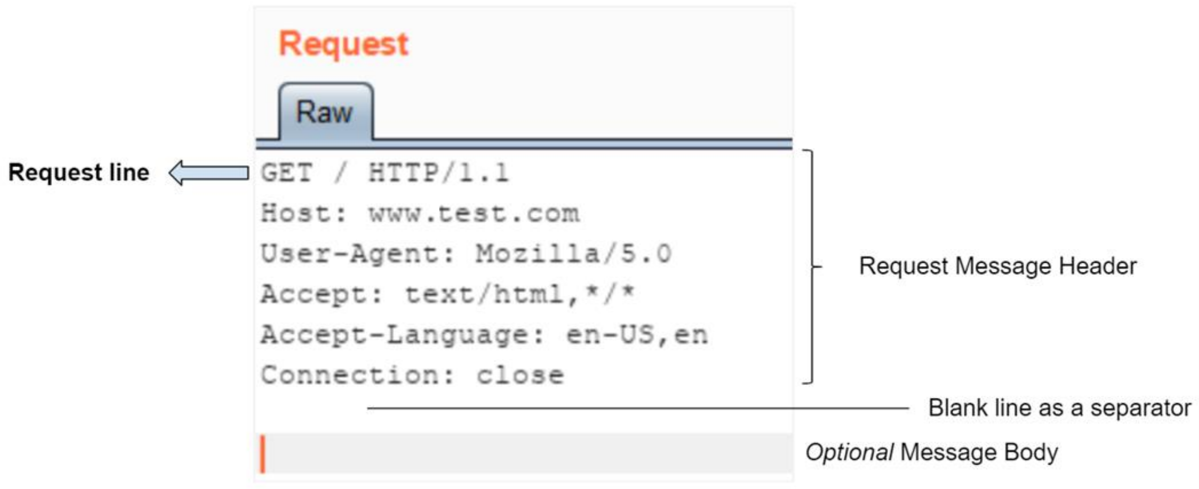
Where *https* is the protocol, www.test.com is the domain, and */test/1* is the "*path and resource name*"- component.

Since the port is not explicitly stated, the client (i.e. the web browser) sets it to *443* by default, in this case.

HTTP Request and HTTP Response - messages share the same basic structure:



Example of an HTTP Request:



```
GET / HTTP/1.1
Host: www.test.com
User-Agent: Mozilla/5.0
Accept: text/html,*/*
Accept-Language: en-US,en
Connection: close

```

The *request line* specifies the HTTP method, resource ("/") and the HTTP-version ("1.1").

In above example the method *"GET"* is used to request a specific resource. Other common HTTP methods are:

POST – to *post* content to the server.

HEAD – to request only *http headers* for a request, omitting the *response message body*. This can be used to retrieve certain configuration options.

OPTIONS – to request a description of communication options for the target resource, the server should respond with a list of supported HTTP methods.

PUT – another method (among 'POST') to *send content* to the server. POST is more commonly used on web pages, whereas PUT is more common in APIs.

DELETE – to instruct the web server to *delete* a specific resource or content.

Request Headers

HTTP header fields are part of the header section of request and response messages. They can define a wide variety of operating parameters of an HTTP transaction:

Host: www.test.com
The domain name of the server.

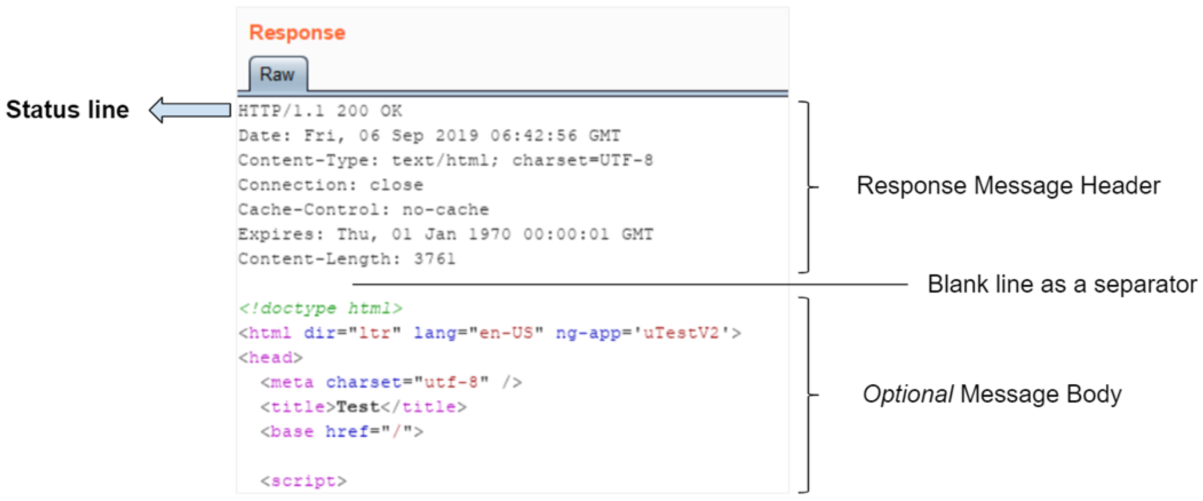
User-Agent: Mozilla/5.0
The *user agent string* of the browser.

Accept: text/html,*/*
Part of the *content negotiation*; Acceptable media types for the response.

Accept-Language: en-US,en
List of acceptable human languages for response (content negotiation).

Connection: close
Control options for the current connection.

Example of an HTTP Response:



```
HTTP/1.1 200 OK
Date: Fri, 06 Sep 2019 06:42:56 GMT
Content-Type: text/html; charset=UTF-8
Connection: close
Cache-Control: no-cache
Expires: Thu, 01 Jan 1970 00:00:01 GMT
Content-Length: 3761

<!doctype html>
<html dir="ltr" lang="en-US" ng-app='uTestV2'>
<head>
  <meta charset="utf-8" />
  <title>Test</title>
  <base href="/">

  <script>
```

The *status line* indicates whether the request was successful, also it shows the HTTP version, and it includes an *http status code*:

200 OK … The request was successful.

302 Found … Instructs the client to look at (browse to) another URL.

400 Bad Request … The server cannot or will not process the request due to a client error (e.g. syntax error, size, message format).

501 Not Implemented … The server cannot fulfil the request or does not recognize the request method.

HTTP response status codes are separated into five categories, where the first digit of the status code defines the class of the response:

- 1xx (Informational)
- 2xx (Successful)
- 3xx (Redirection)
- 4xx (Client Error)
- 5xx (Server Error)

<u>Response Headers</u>

Header fields (also in request headers) are colon-separated "key-value pairs". 'Connection' being the *key* in – for example – the "Connection: close" header. And 'close' is the *value*.

Some examples from above response:

Date: Fri, 06 Sep 2019 06:42:56 GMT
HTTP-date determining the date and time when the message was sent. ("Server time"!)

Content-Type: text/html; charset=UTF-8
The *MIME type* of the content in the response message body.

Connection: close
Control options for the current connection. "close" meaning that the connection will be closed after completion of the response. The only other possible option is "keep-alive", indicating that there may be other messages following, and therefore the client should maintain a persistent connection.

Cache-Control: no-cache
This tells all caching mechanisms from server to client whether they may cache this message. Another possible value would be for example: “Cache-Control: max-age=3600”, measured in seconds. Indicating that the object may not be cached for longer than *x* seconds.

Expires: Thu, 01 Jan 1970 00:00:01 GMT
HTTP-date after which the response is considered stale.

Content-Length: 3761
The length of the response body in bytes.

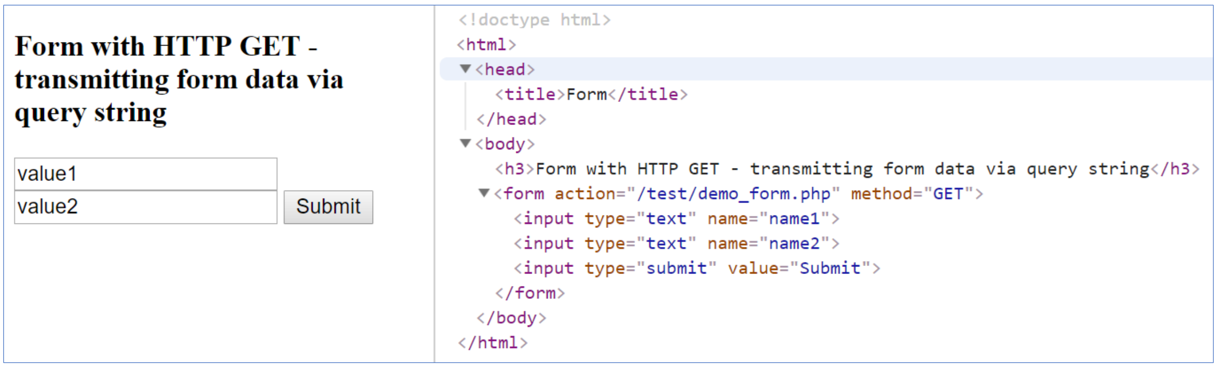
There are dozens of other possible request and response headers. Many of them are self-explanatory, others must be looked up.

Query strings

There are a lot of options to transmit data to a web server in HTTP requests. In websites the most common ways are via query strings, JSON in background-requests (without reloading a page), and *multipart* POST requests.

In APIs data will often be transmitted via XML or JSON instead of forms. Those methods will be discussed later in this course.

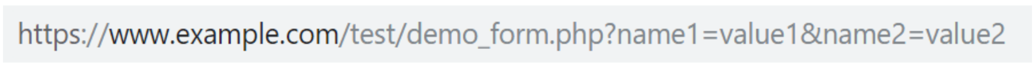
Let’s first look at the ‘traditional’ way of transmitting form-data with query strings:


Form with HTTP GET - transmitting form data via query string
value1
value2
Submit

```
<!doctype html>
<html>
▼<head>
    <title>Form</title>
  </head>
▼<body>
    <h3>Form with HTTP GET - transmitting form data via query string</h3>
  ▼<form action="/test/demo_form.php" method="GET">
      <input type="text" name="name1">
      <input type="text" name="name2">
      <input type="submit" value="Submit">
    </form>
  </body>
</html>
```

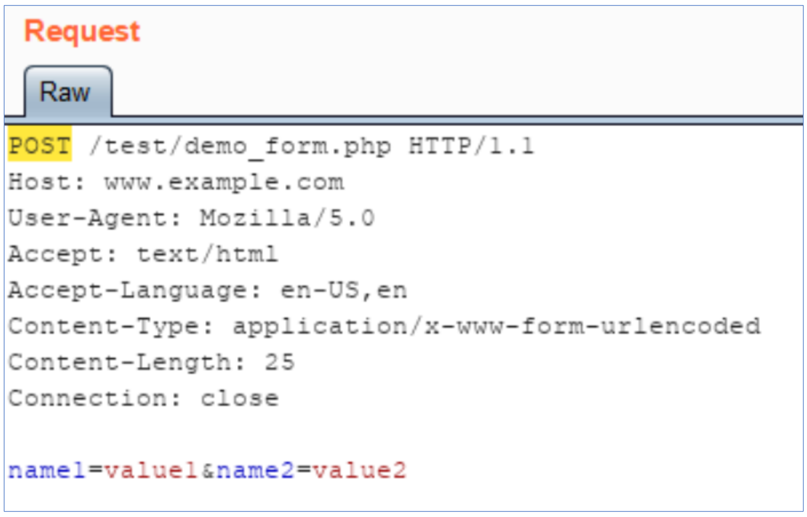
Above screenshot shows a simple html-form with two input-fields and a submit button. The “method” of the form is defined as “GET”, which is also the default value if the attribute is left omitted.

Once submitted, the input will be sent to the specified location **“/test/demo_form.php”** of the current host (where the form is located):


```
https://www.example.com/test/demo_form.php?name1=value1&name2=value2
```

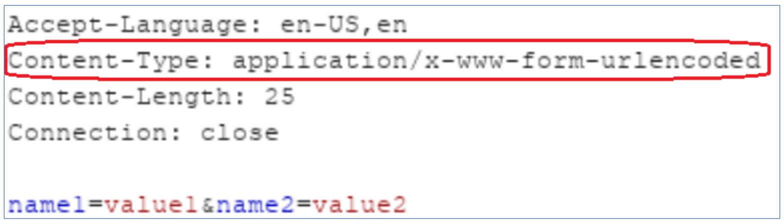
And, as you can see, the form-contents are transmitted in the URL of the GET request, in the form of a **query-string**. Consisting of a string of name/value pairs, appended to the target page after a question mark. The individual name/value pairs (*name1=value1 and name2=value2*) are separated with ampersand **&**. The target page is defined in the form's "action" parameter.

The same request when the **method** is changed to **POST** will look like this:



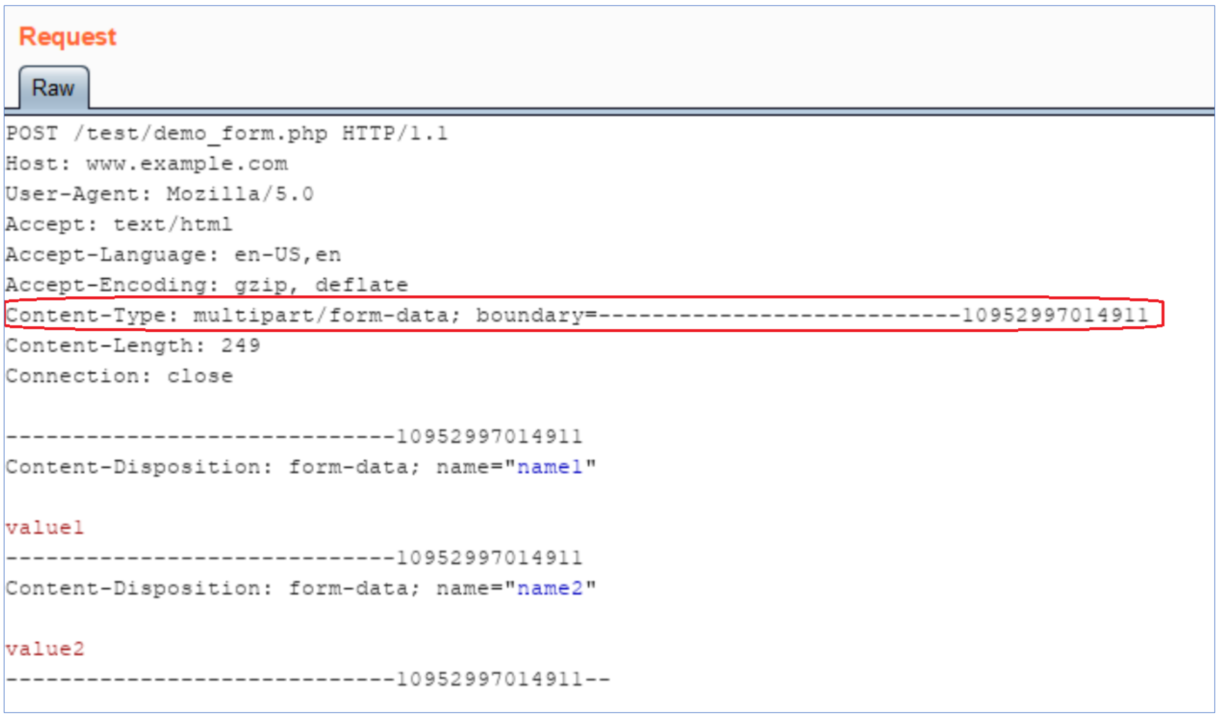
```
POST /test/demo_form.php HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0
Accept: text/html
Accept-Language: en-US,en
Content-Type: application/x-www-form-urlencoded
Content-Length: 25
Connection: close

name1=value1&name2=value2
```

The same key/value pair **"name1=value1&name2=value2"** is now transmitted in the *request message body* at the bottom of the request, below the blank line separating it from the message header.


```
Accept-Language: en-US,en
Content-Type: application/x-www-form-urlencoded
Content-Length: 25
Connection: close

name1=value1&name2=value2
```

The *content-type* header highlighted above is telling the server that the data is url-encoded. That means that all non-ASCII characters will be converted. For example, a space character becomes +, an "@" becomes "*%40*" and so on.

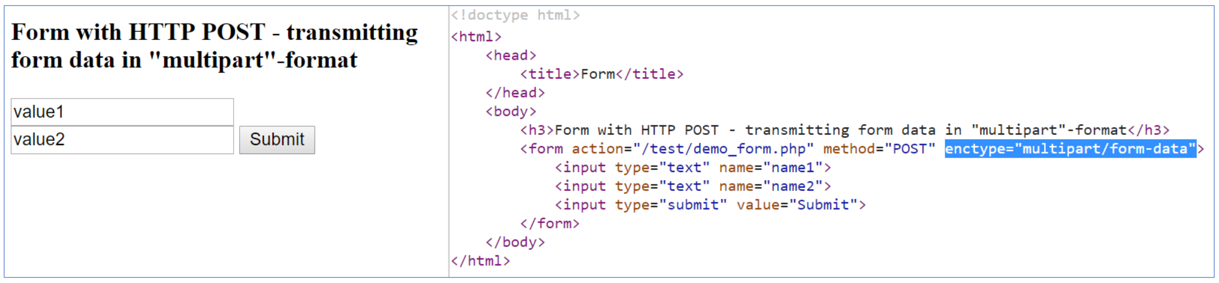
The same form transmitted as **multipart/form-data**, by defining the *"enctype" attribute*, will look like this:



```
POST /test/demo_form.php HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0
Accept: text/html
Accept-Language: en-US,en
Accept-Encoding: gzip, deflate
Content-Type: multipart/form-data; boundary=---------------------------10952997014911
Content-Length: 249
Connection: close

-----------------------------10952997014911
Content-Disposition: form-data; name="name1"

value1
-----------------------------10952997014911
Content-Disposition: form-data; name="name2"

value2
-----------------------------10952997014911--
```

The values are now separated by the "boundary", which will be automatically set by the browser. And each value has one or more additional attributes. This method is typically used for file-upload forms in web sites.

Notice the added "enctype" attribute in the form's source-code:



Form with HTTP POST - transmitting form data in "multipart"-format

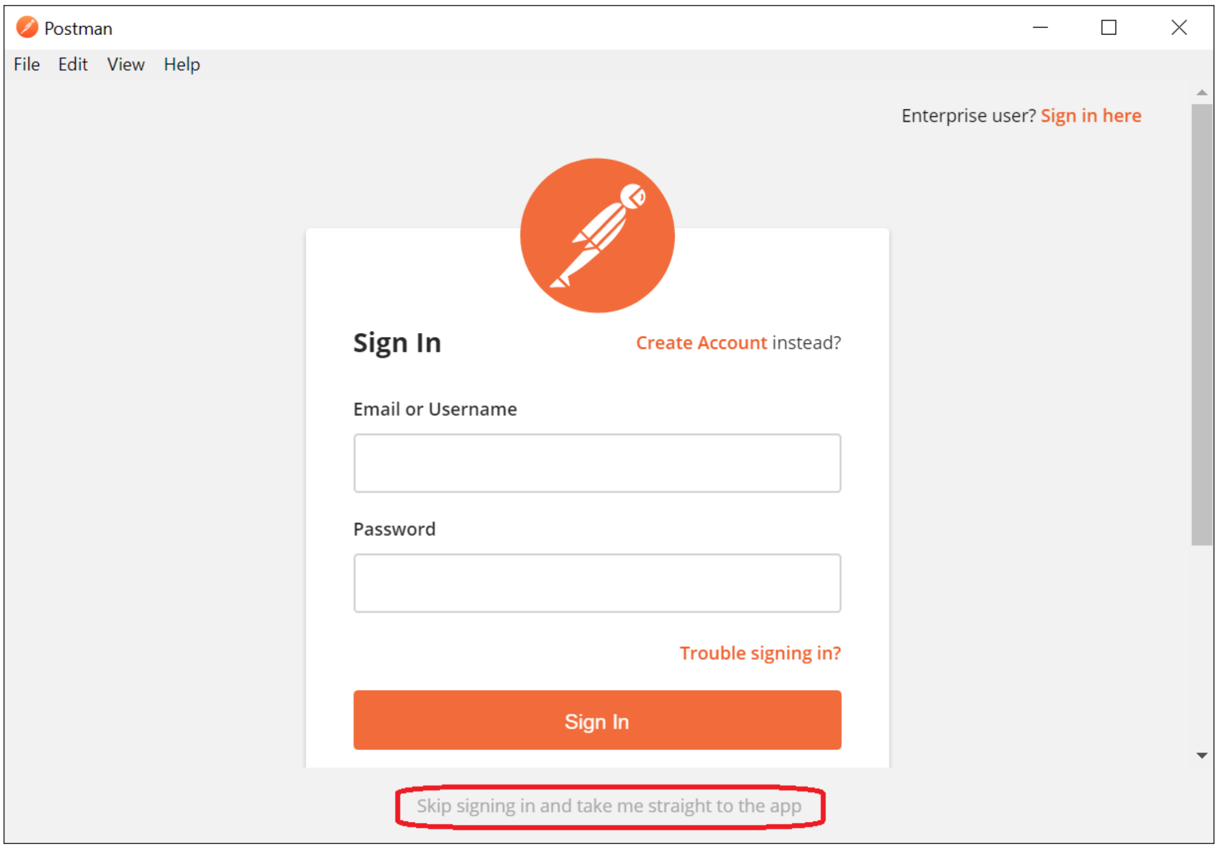
```
<!doctype html>
<html>
    <head>
        <title>Form</title>
    </head>
    <body>
        <h3>Form with HTTP POST - transmitting form data in "multipart"-format</h3>
        <form action="/test/demo_form.php" method="POST" enctype="multipart/form-data">
            <input type="text" name="name1">
            <input type="text" name="name2">
            <input type="submit" value="Submit">
        </form>
    </body>
</html>
```

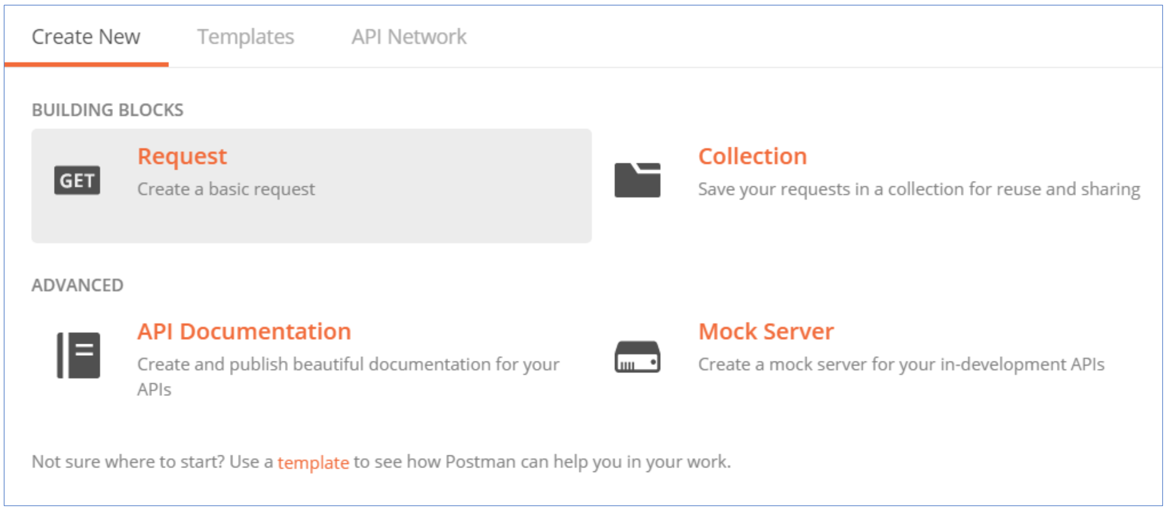
API testing with Postman

Postman (getpostman.com) is a widely used and sophisticated API Development Environment which is ideal for most API testing. The basic version is free, and available for Windows, macOS or Linux.

It features a neat and clean interface and doesn't require a lot of setup steps. Therefore, it's ideal for taking the first steps in API testing!

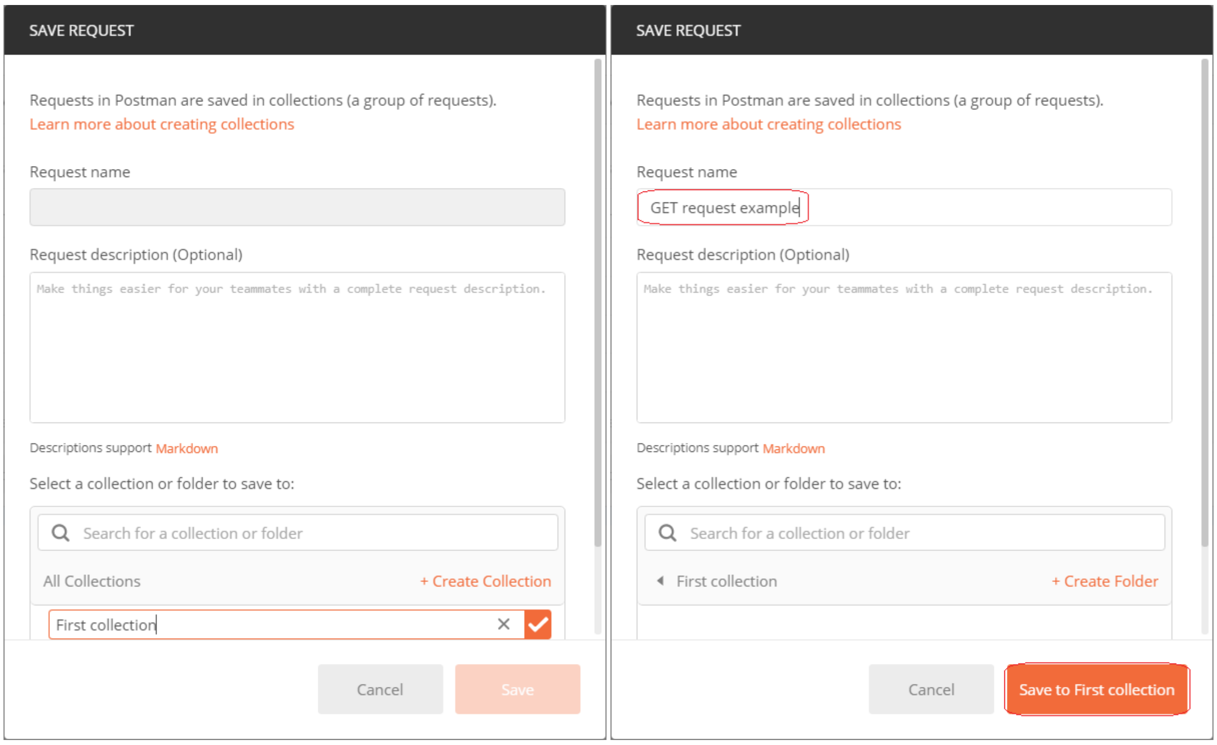
After installation when you run Postman you will see a login-screen. At the bottom of the page click the button: "Skip signing in and take me straight to the app".




Create New
Templates
API Network
BUILDING BLOCKS
GET
Request
Create a basic request
Collection
Save your requests in a collection for reuse and sharing
ADVANCED
API Documentation
Create and publish beautiful documentation for your APIs
Mock Server
Create a mock server for your in-development APIs
Not sure where to start? Use a template to see how Postman can help you in your work.

On launch you will see the “Create New” screen (pictured above). Select “Request” to create a new basic request.

For the first test create a new collection by clicking the “Create Collection” button near the bottom of the **Save Request** window and give it a name like “First collection”. Then click the checkmark-button to save it. Next, enter a Request name and click **Save to First collection**:


SAVE REQUEST

Requests in Postman are saved in collections (a group of requests).
Learn more about creating collections

Request name:

Request description (Optional): Make things easier for your teammates with a complete request description.

Descriptions support Markdown

Select a collection or folder to save to:

Search for a collection or folder

All Collections + Create Collection

First collection

SAVE REQUEST

Requests in Postman are saved in collections (a group of requests).
Learn more about creating collections

Request name: GET request example

Request description (Optional): Make things easier for your teammates with a complete request description.

Descriptions support Markdown

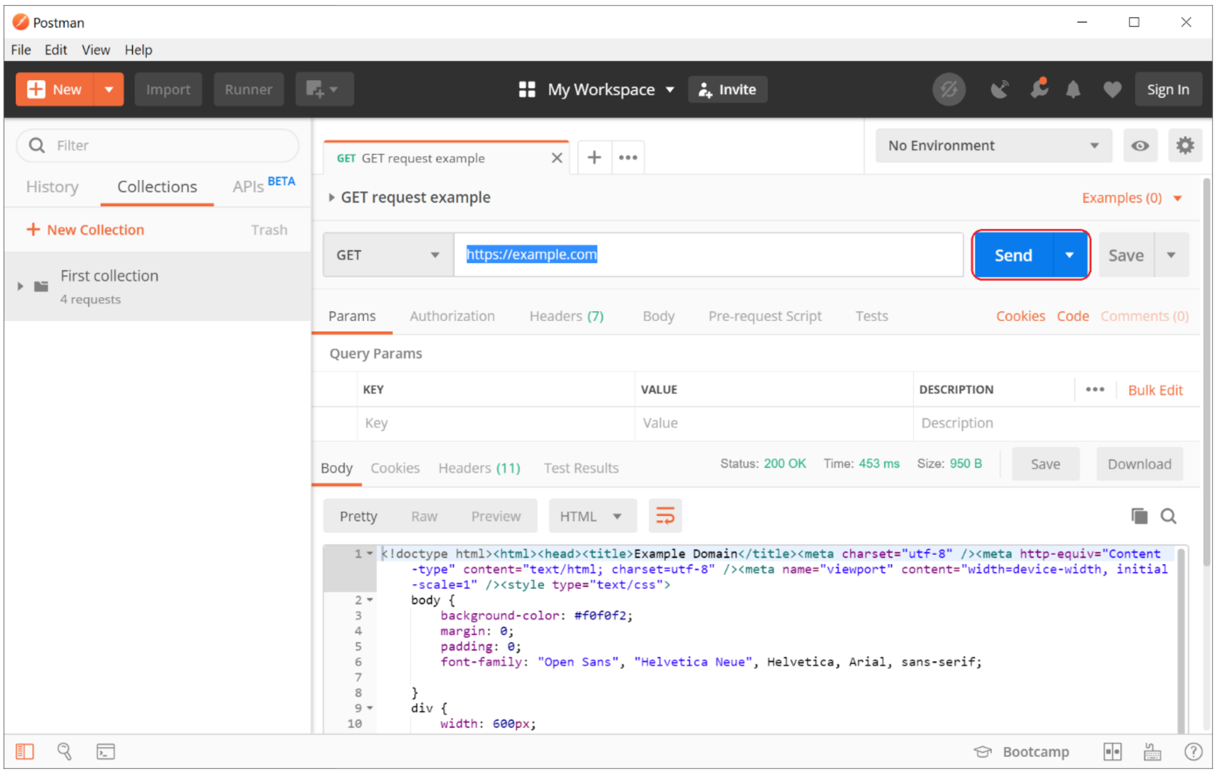
Select a collection or folder to save to:

Search for a collection or folder

First collection + Create Folder

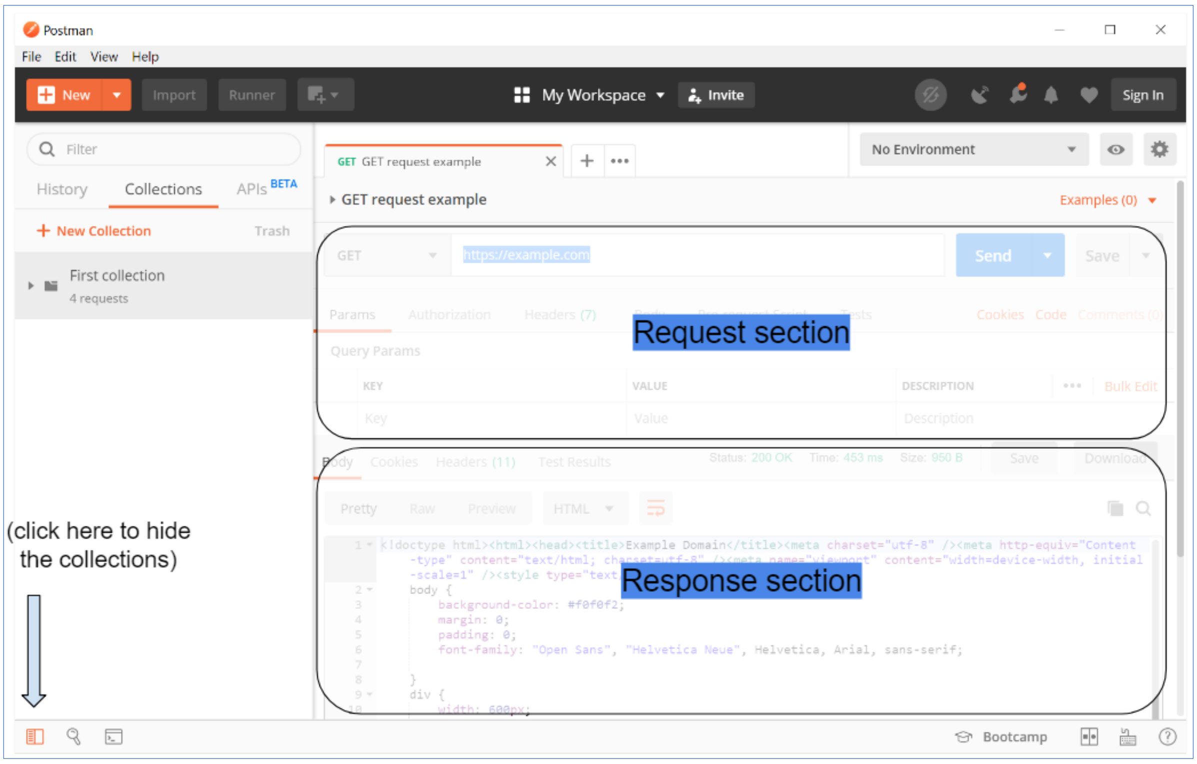
Now it’s time to create your first GET request.

Enter the full URL of the resource you want to request (try *https://example.com*) into the field labelled as “Enter request URL” and click **Send**:

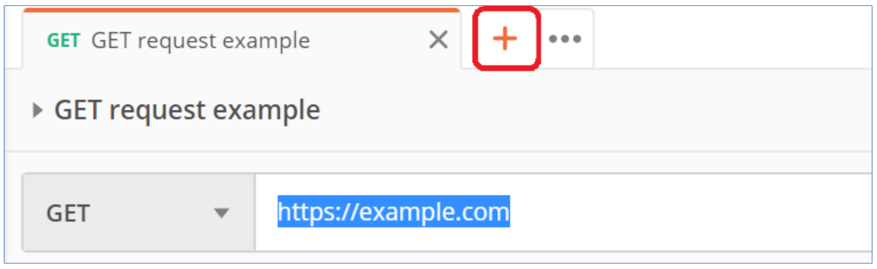


```
<!doctype html><html><head><title>Example Domain</title><meta charset="utf-8" /><meta http-equiv="Content
    -type" content="text/html; charset=utf-8" /><meta name="viewport" content="width=device-width, initial
    -scale=1" /><style type="text/css">
    body {
        background-color: #f0f0f2;
        margin: 0;
        padding: 0;
        font-family: "Open Sans", "Helvetica Neue", Helvetica, Arial, sans-serif;

    }
    div {
        width: 600px;
```

The view you'll see is separated into the request area at the top, with Params, Authorization, Headers and other tabs. And the response area at the bottom, again with Cookies, Headers and other tabs. By default, the response body will be shown at the bottom, in a slightly *beautified* form.



To create another request, click the **+** icon. You can later choose 'Save' to store new requests in the same – or any other – collection.



CRUD within HTTP (Create, read, update, delete)

CRUD in computer programming are the four basic operations of persistent storage.

In HTTP they are represented by the following HTTP-methods:

Create: *PUT*, *POST*

Read: *GET*

Update: *PUT*, *POST*, *PATCH*

Delete: *DELETE*

There is obviously some ambivalence going on between PUT, POST and PATCH. In the real world, it depends on the API (or "*web service*") which method will be used when.

But ideally, **PUT** will be used to *update* a resource in an **idempotent** way: multiple identical requests should always have the same result, just like GET. The same **GET** request should always lead to the same result – the same response. But unlike **POST**, which *creates* a resource. So, if you call it multiple times, *multiple resources* will be created (if allowed).

PATCH is meant to apply a *partial update* to the resource. But only if it already exists! So, unlike PUT it will *not* create a new entry, if the specified resource does not exist.

While many of above decisions are up to the developers, there is *one* common mistake regarding the choice of the correct HTTP-method: if during your testing you should ever see any GET request that requires authentication (we will cover this later) which *updates* a resource, that would most likely be an issue you should report. It can have a significant security impact, because GET requests, as specified in the HTTP Protocol, are supposed to be *idempotent* and *safe*. Web browsers expect it to be that way and are therefore less restrictive.

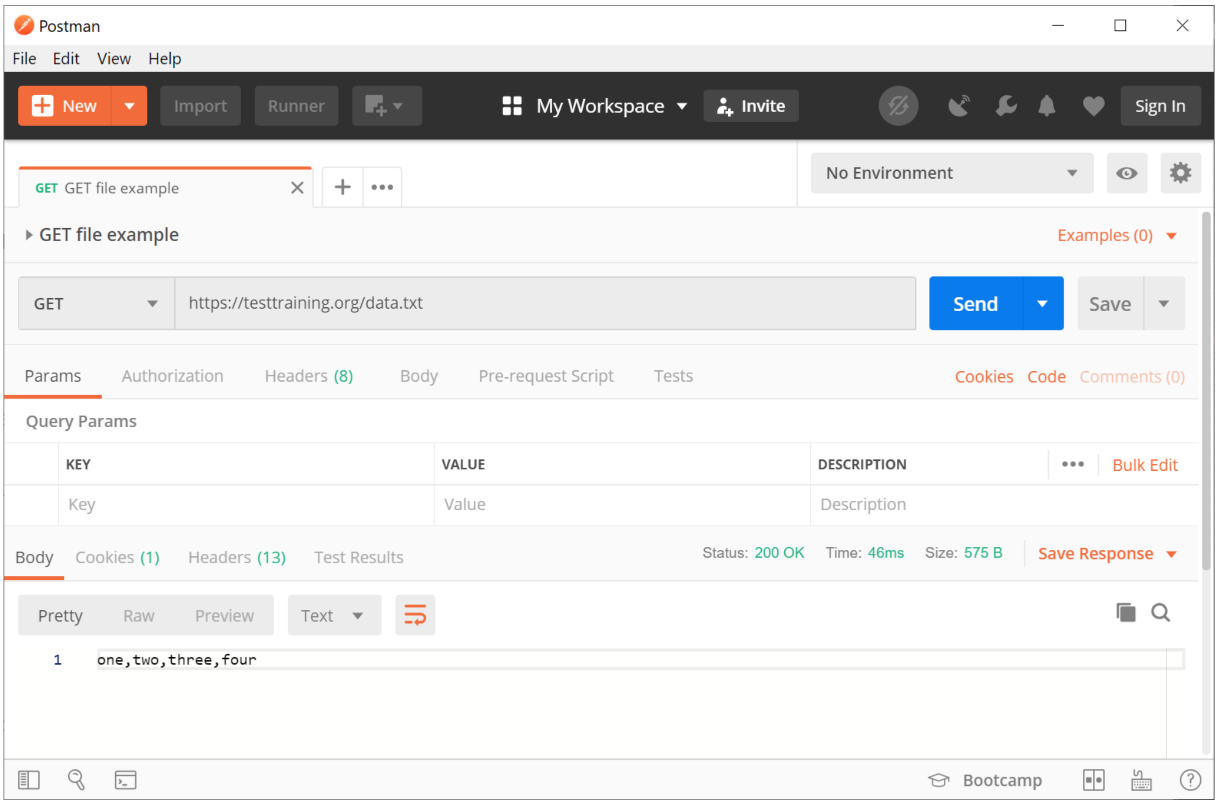
Safe HTTP method means that any request using a *safe* method cannot modify or change a resource.

HTTP Method	Idempotent	Safe
HEAD	yes	yes
OPTIONS	yes	yes
GET	yes	yes
PUT	yes	no
POST	no	no
PATCH	no*	no
DELETE	yes	no

* It is possible to issue PATCH requests in such a way as to be idempotent, by using the HTTP headers "ETag" and/or "If-Modified-Since".

<u>In the following, let's look at examples for each of them.</u>

GET



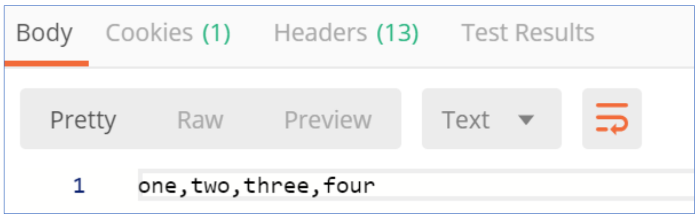
GET https://testtraining.org/data.txt

Status: 200 OK Time: 46ms Size: 575 B

```
one,two,three,four
```

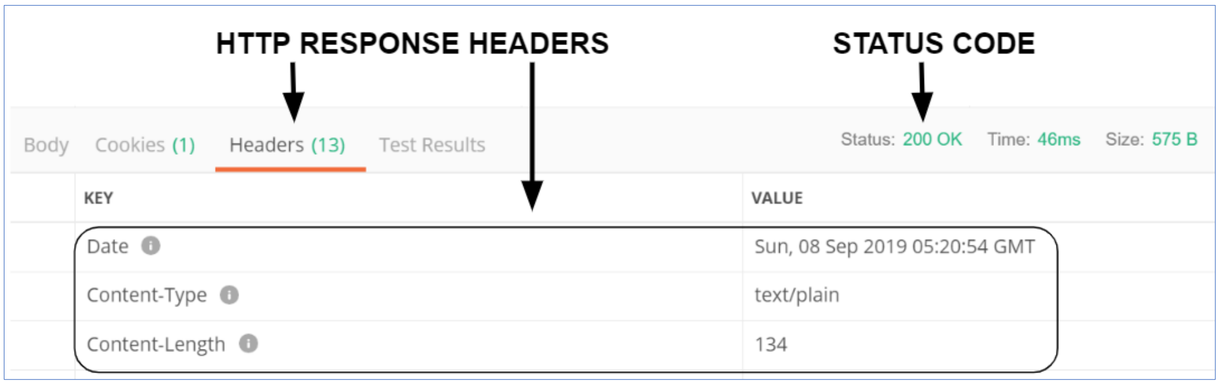
For this example create a new request (click the + button at the top), it will be a GET request by default.

To request the contents of a simple text file with Postman, just enter the full URL and click **Send** to submit the request. The contents of the file will be displayed in the **Response** section at the bottom:



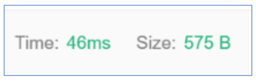
```
1   one,two,three,four
```

Check out the “Headers” tab, which displays the full list of the response headers:



KEY	VALUE
Date	Sun, 08 Sep 2019 05:20:54 GMT
Content-Type	text/plain
Content-Length	134

In the response section Postman will also display the HTTP status code (200 OK), and next to it the execution time for the request (46 milliseconds). As well as the response size of 575 Bytes:

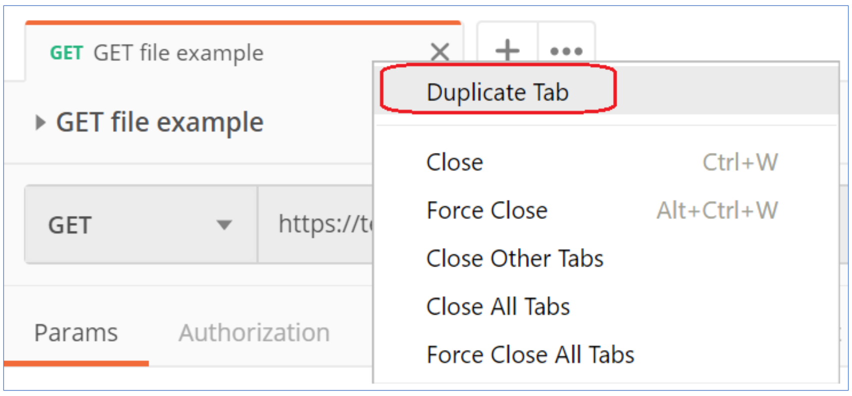


The result / response returned in this example is the plain text “one,two,three,four”, which is a list of those four words in CSV-format. **CSV** stands for “comma-separated values”, it is used to store a list of individual values, separated by a single comma in-between them.

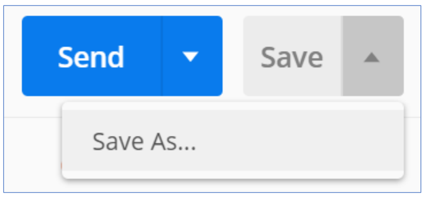
Please store this GET request in your collection as: “GET file example”, before you move on to the next *PUT* request!

PUT

Now it’s time to send some data. Another method to create a new request without needing to enter all input again is to right-click the tab of an existing request, and selecting “Duplicate tab” from the context-menu:

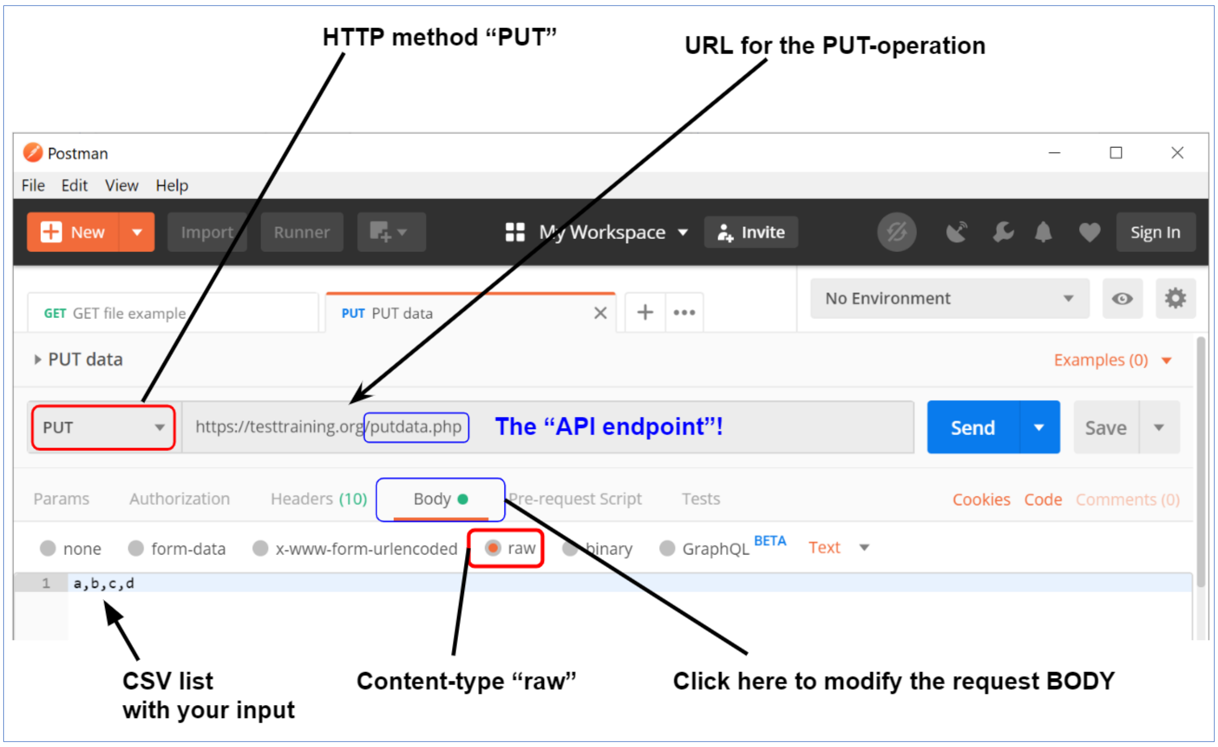


Please do this now and change the request-method to “PUT”. Afterwards choose “Save As...” from the dropdown next to the **Save** button to save it as a new request to your collection:



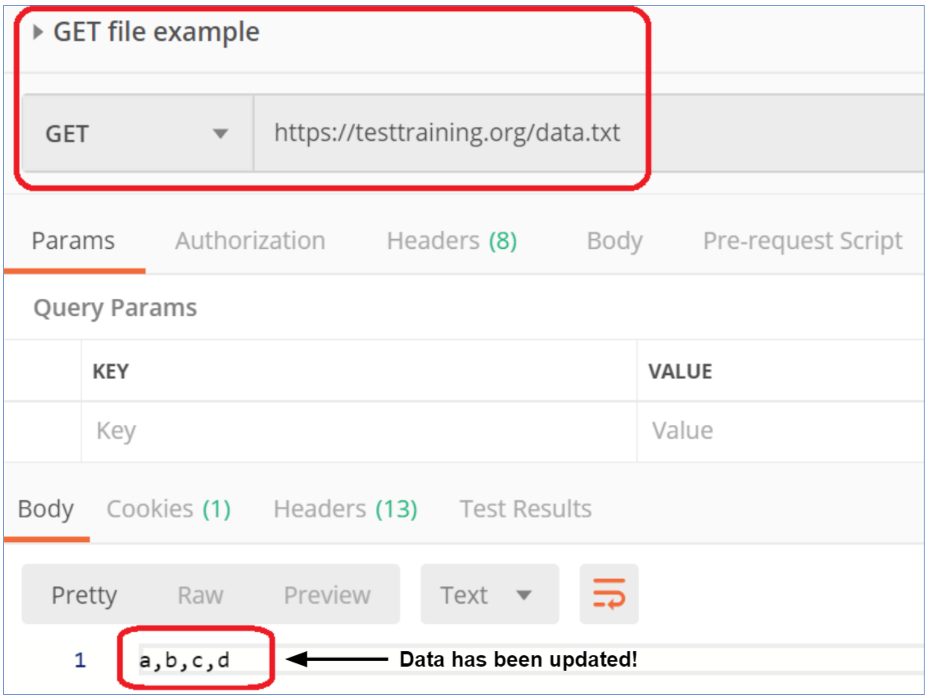
Afterwards fill in the URL where you want to post the data to (this will be called the ‘endpoint’ or ‘**API endpoint**’). And select the “Body” tab in the request section. It will let you choose the content-type. By default, it will be set to “none” since there is no message body.

Change that by picking “raw” as the body content type and type the CSV-data “a,b,c,d” in the new text field that appears once you’ve picked the content type.



```
a,b,c,d
```

Once you are all set click **Send**, and then go back to your previous request ("GET file example") and submit that one again, to verify that the data has been updated. It should now show the CSV-data that you just submitted in the response:



```
a,b,c,d
```

Now, if you don't want to modify the entire CSV list, but only one entry – e.g. you want to replace "c" with "three" – then you can use the PATCH method.

PATCH

The syntax for *patching* records can vary a lot between APIs or web services. You will learn about the details later in this course, APIs may use standards like JSON, XML, or even proprietary formats.

In this example the PATCH- endpoint allows to update specific entries of the CSV list to a new value. As the **parameters** it *expects* the index that you want to modify, as well as the new value.

0=Text

This means that you want to update the first entry (it starts counting at 0) and set it to the value: "Text".

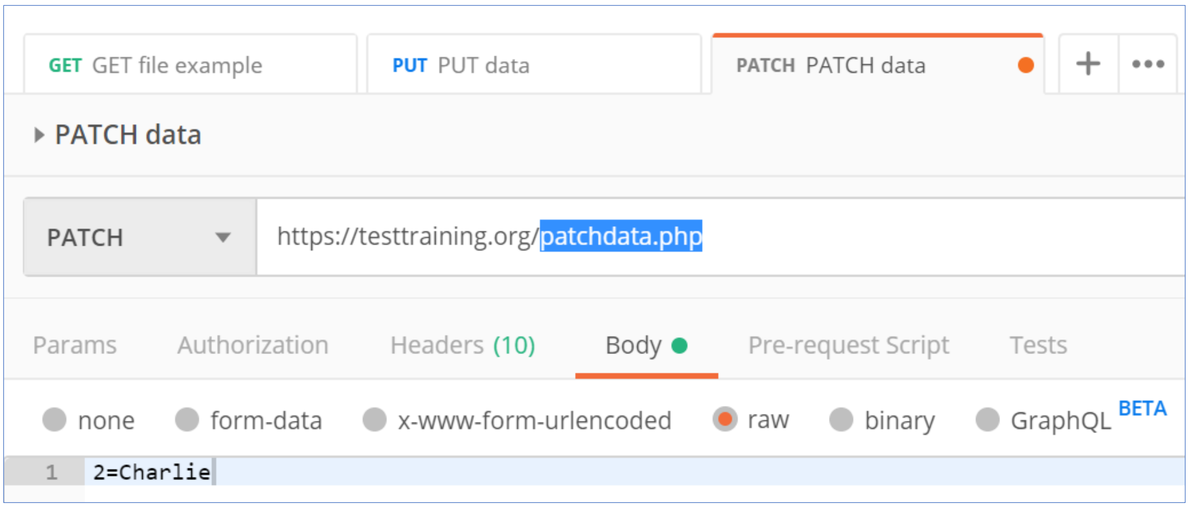
When you submit this, the list will change from the previous "a,b,c,d" to **"Text,b,c,d"**.

Or if you patch:

2=Charlie

That instruction would set the CSV data to the new value: **"a,b,Charlie,d"**.

The index is *"2"*, or the *third entry* in the CSV list, and the new value is *"Charlie"*.

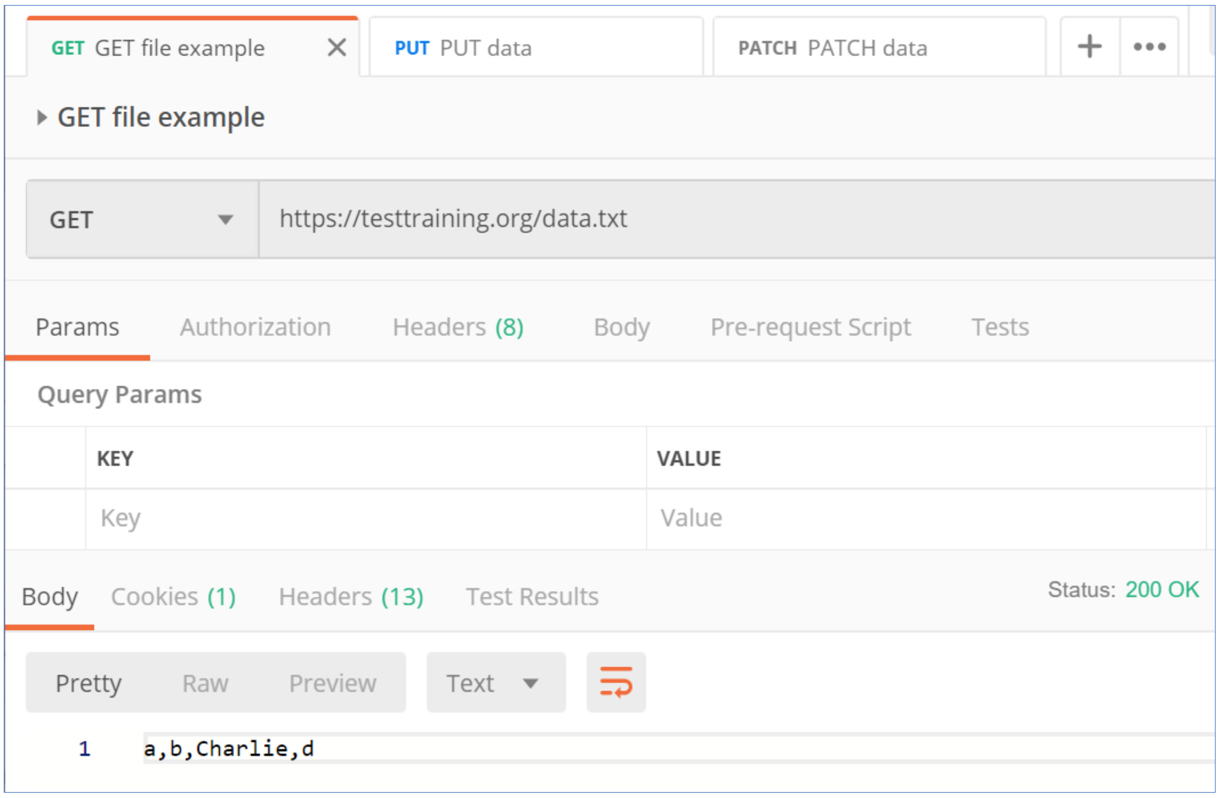


PATCH https://testtraining.org/patchdata.php

```
2=Charlie
```

This is the complete PATCH- request to change the **third** entry (remember, indexes like this usually start counting at zero) of the CSV list to the new value "Charlie". Like in the PUT request example before, it uses a plain text **Body** for the message data. When you duplicate the previous "PUT data" request, just change the HTTP method to "PATCH" and modify the body text ("2=Charlie"). Then click **Send** to submit it, to get the pictured result. Remember to use "Save as…" to store it as a new request afterwards, and don't overwrite the original.

If you request the data afterwards with the previous “GET file example” request, it should look like this:



```
a,b,Charlie,d
```

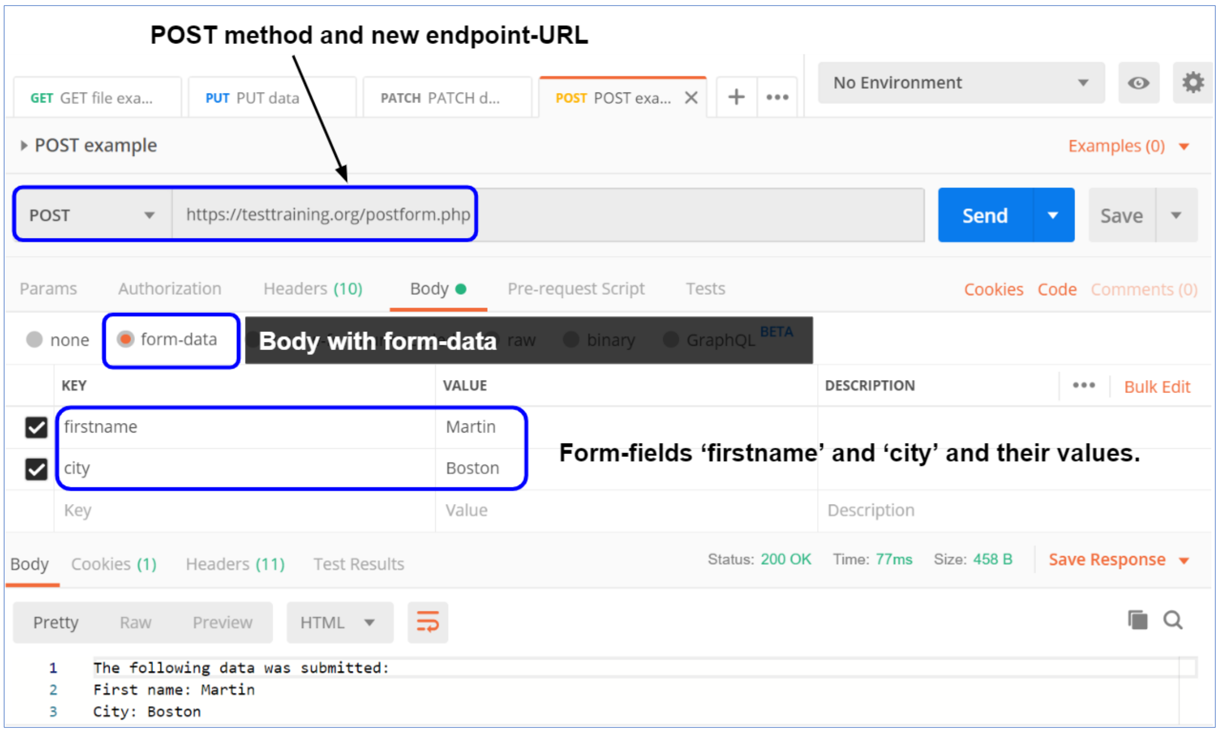
Remember to *save* the new PATCH request to your collection, before you move on to the next method!

POST

POST can be used for both, full and partial updates of a resource. In web applications POST is often used to submit form content.

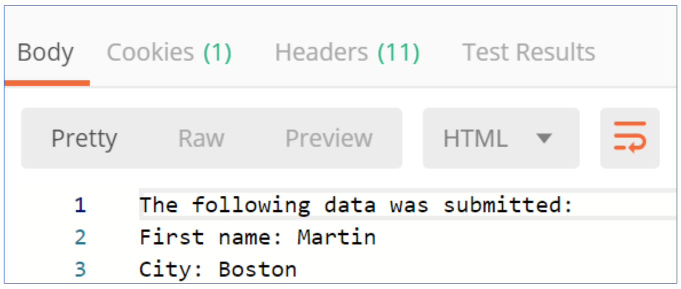
Create a new request with POST as the HTTP method, and use the “form-data” format for the request body. Enter two keys: “firstname” and “city”, and fill in any values of your choice.

The complete request should look like this:



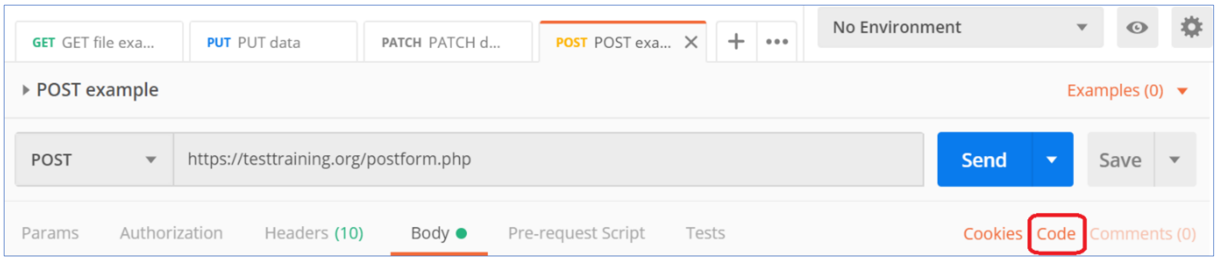
```
The following data was submitted:
First name: Martin
City: Boston
```

When you submit the request by clicking the **Send** button, it should show a confirmation in the response, and – in this example – the response also contains the submitted data:

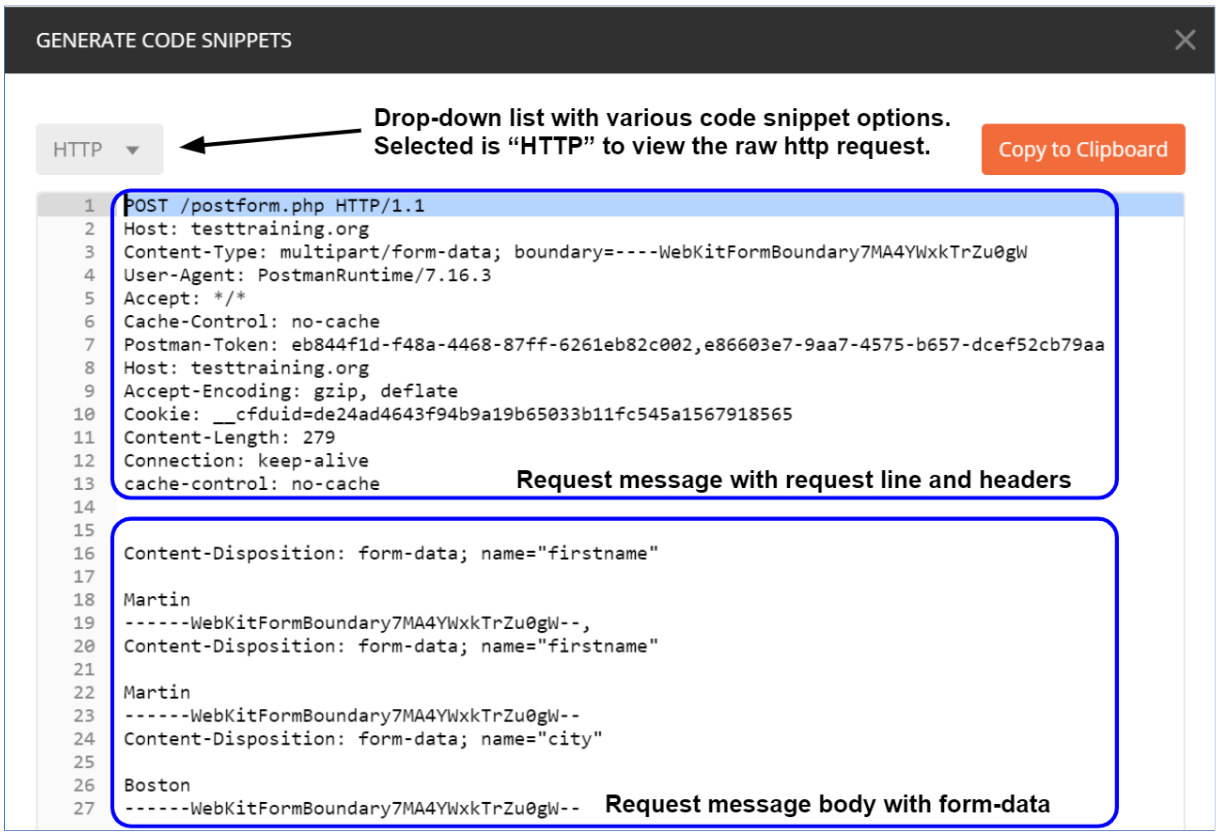


```
The following data was submitted:
First name: Martin
City: Boston
```

You may have noticed the "Code" button already:

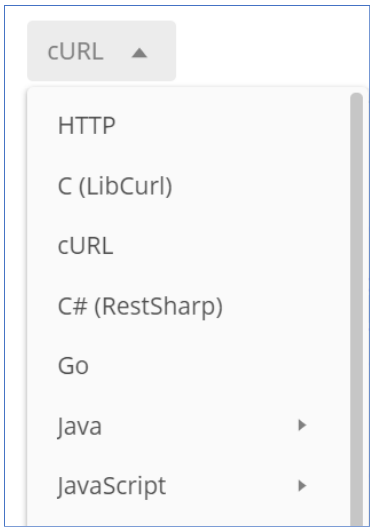


When you click on it you will see options to convert the *request data* into various formats for exporting.


GENERATE CODE SNIPPETS
HTTP
Drop-down list with various code snippet options.
Selected is "HTTP" to view the raw http request.
Copy to Clipboard
Request message with request line and headers
Request message body with form-data

```
POST /postform.php HTTP/1.1
Host: testtraining.org
Content-Type: multipart/form-data; boundary=----WebKitFormBoundary7MA4YWxkTrZu0gW
User-Agent: PostmanRuntime/7.16.3
Accept: */*
Cache-Control: no-cache
Postman-Token: eb844f1d-f48a-4468-87ff-6261eb82c002,e86603e7-9aa7-4575-b657-dcef52cb79aa
Host: testtraining.org
Accept-Encoding: gzip, deflate
Cookie: __cfduid=de24ad4643f94b9a19b65033b11fc545a1567918565
Content-Length: 279
Connection: keep-alive
cache-control: no-cache


Content-Disposition: form-data; name="firstname"

Martin
------WebKitFormBoundary7MA4YWxkTrZu0gW--,
Content-Disposition: form-data; name="firstname"

Martin
------WebKitFormBoundary7MA4YWxkTrZu0gW--
Content-Disposition: form-data; name="city"

Boston
------WebKitFormBoundary7MA4YWxkTrZu0gW--
```

If you choose "HTTP" from the drop-down at the top left, you will see the *raw http request* – in the exact form the server will receive it. There are also options to let Postman generate code snippets for various programming languages like PHP, Java, Python…



DELETE

The last HTTP method in this list is *DELETE*.

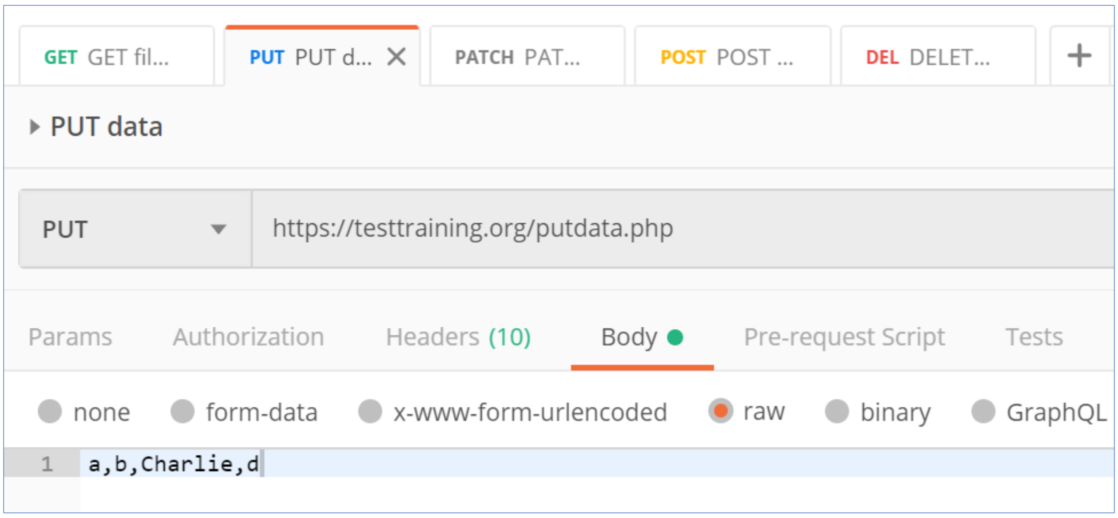
For this example, you can either call the DELETE endpoint with an *index* as described before, to delete a specific entry from the list. Or just call it without any parameters to delete the entire list!

Create a new request and save it in your collection as "DELETE data". Again, use a raw Body and enter the index of the list-entry that you want to delete.

From the previous example the data in the list should be "a,b,Charlie,d".

If you submit the DELETE request with the index *1*, that should remove the *second* entry, so that the resulting list will become: "a,Charlie,d".

In case that you have a different result, submit the previous PUT request first, to restore the original values:

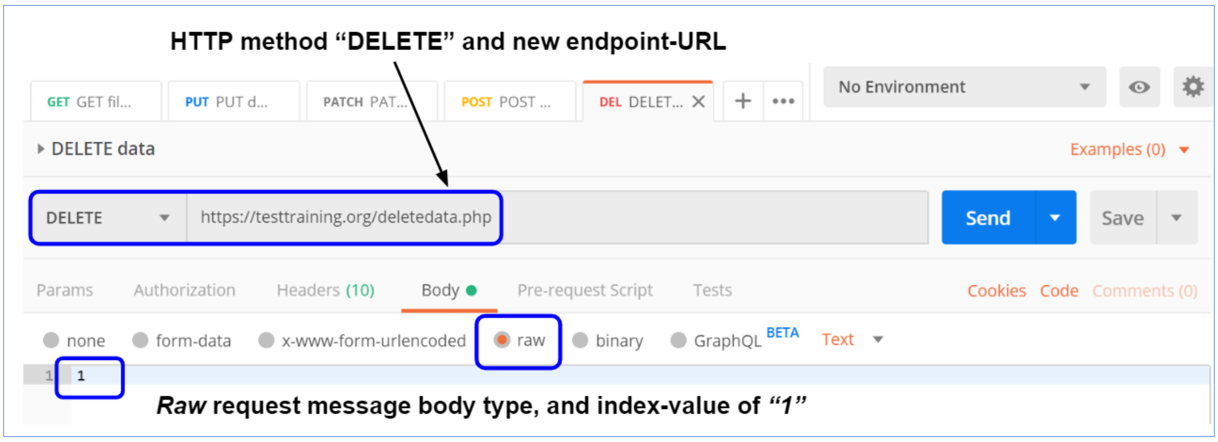


PUT https://testtraining.org/putdata.php

Body: raw

```
a,b,Charlie,d
```

The complete DELETE request:

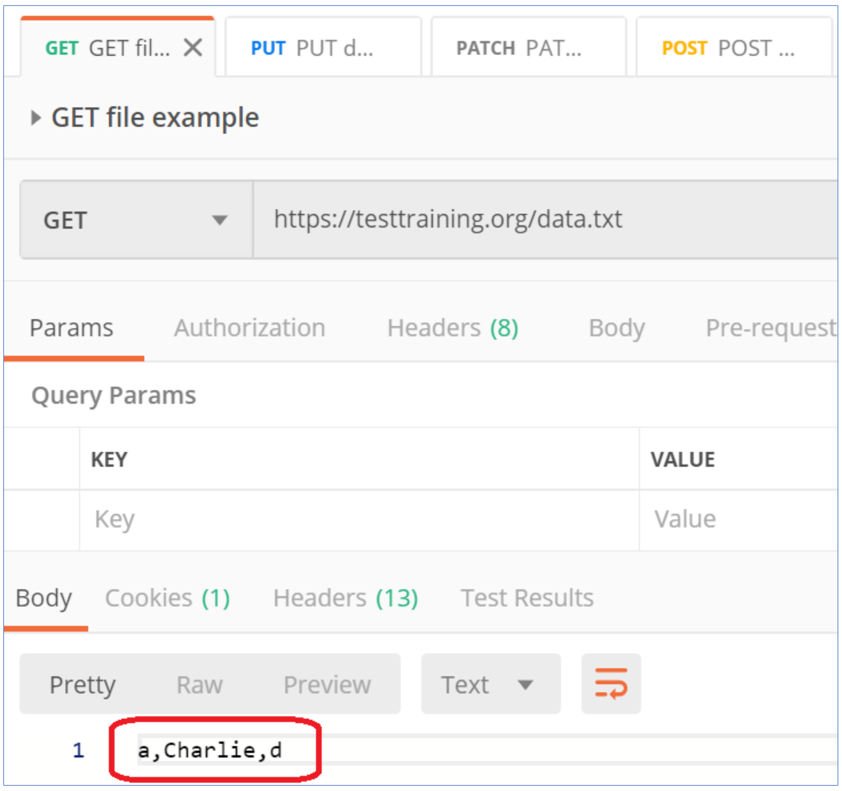


DELETE https://testtraining.org/deletedata.php

Body: raw

```
1
```

To verify that the request was successful, call the previous **GET** request again:



https://testtraining.org/data.txt

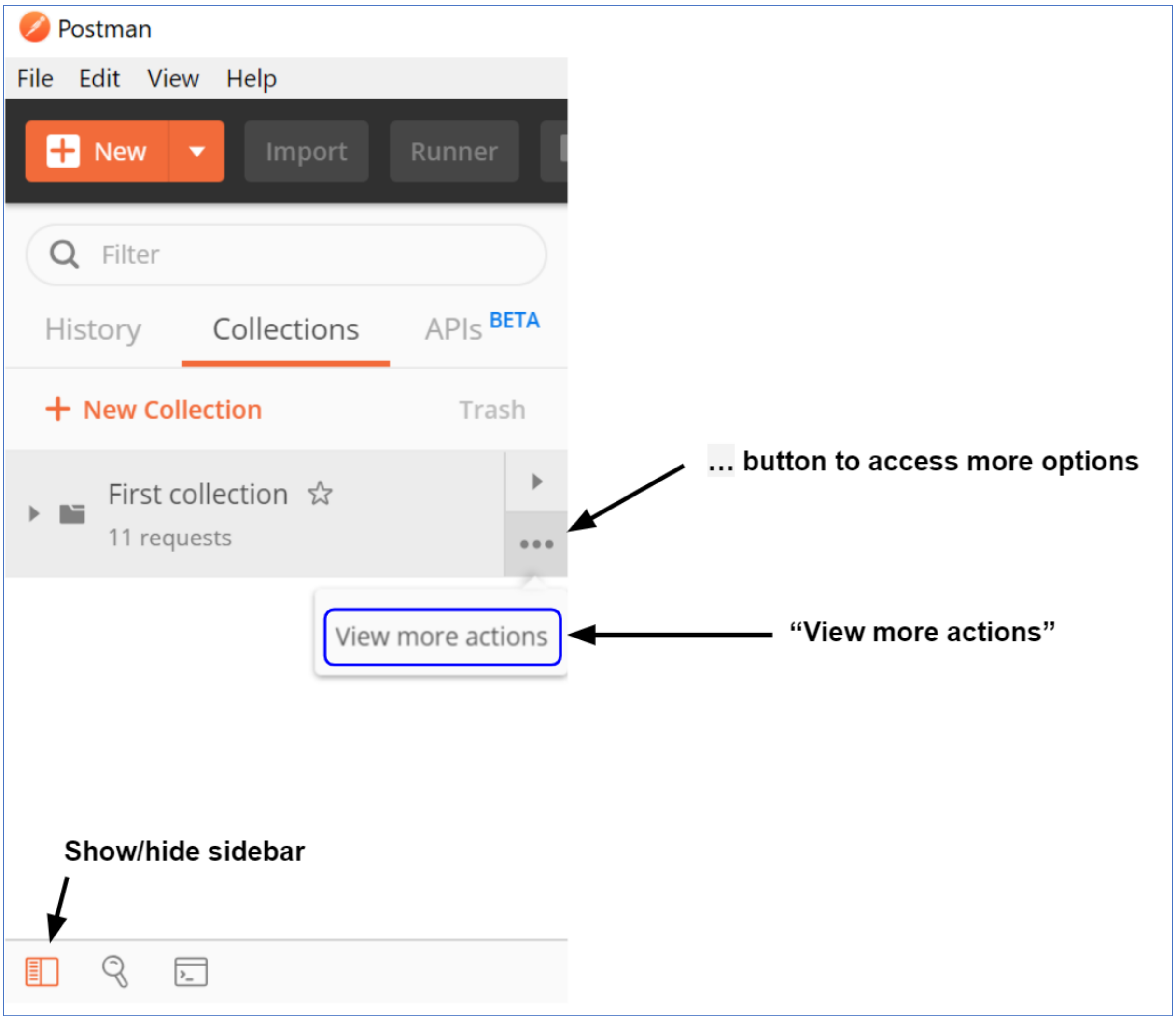
```
a,Charlie,d
```

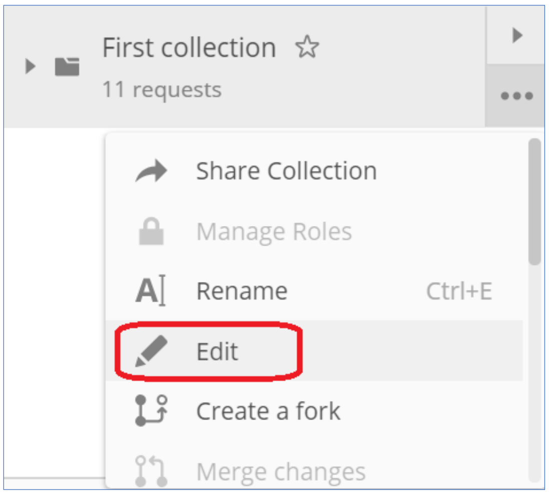
Authorization

In all the previous examples it was possible to request or post data without any need of an authentication. Which is quite uncommon for web services or websites.

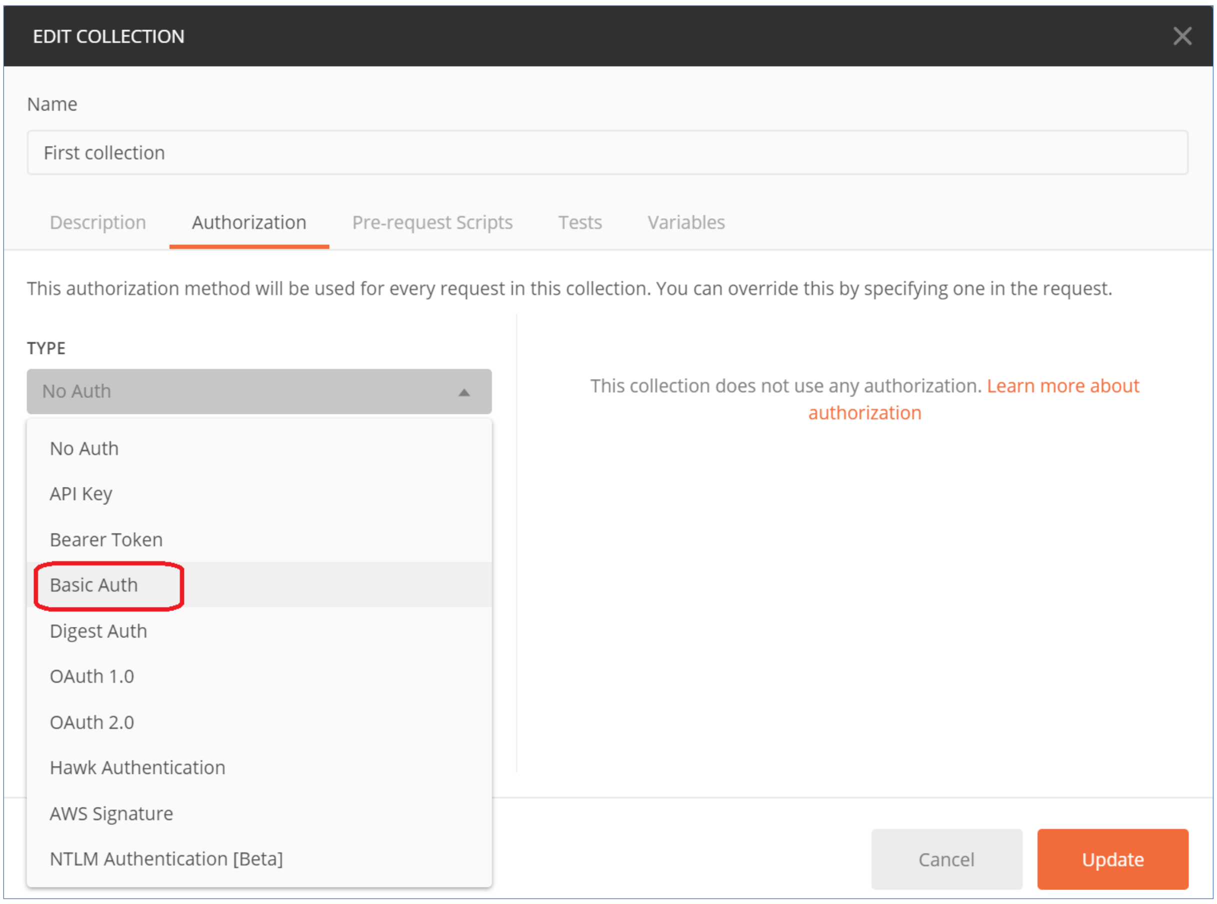
There are lots of options and technologies for authorization / authentication, let's look at least at a few of them! In Postman you can set the required authentication- variables and -settings individually for every request. But when you are testing an API and have a collection, it will usually make sense to make global settings for all the requests in the collection.

When you open the sidebar in Postman and hover with your cursor over the stored collection, a new **"…"** button will appear with access to the collection's settings:





Select "Edit" from the "more actions" menu to access the collection settings:


EDIT COLLECTION

Name

First collection

Description | Authorization | Pre-request Scripts | Tests | Variables

This authorization method will be used for every request in this collection. You can override this by specifying one in the request.

TYPE

No Auth

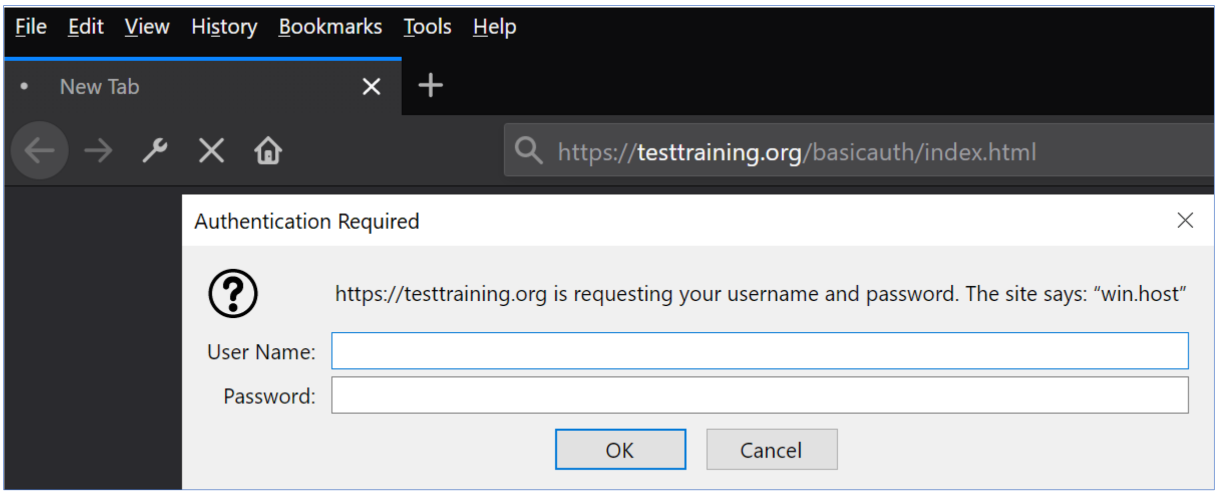
- No Auth
- API Key
- Bearer Token
- Basic Auth
- Digest Auth
- OAuth 1.0
- OAuth 2.0
- Hawk Authentication
- AWS Signature
- NTLM Authentication [Beta]

This collection does not use any authorization. Learn more about authorization

Cancel | Update

Basic Auth

Please select “Basic Auth” from the list. *Basic access authentication* is an authorization method you probably have encountered already. In your web browser calling any Basic Auth- protected page or directory will generate a popup with a login form:


File Edit View History Bookmarks Tools Help

New Tab

https://testtraining.org/basicauth/index.html

Authentication Required

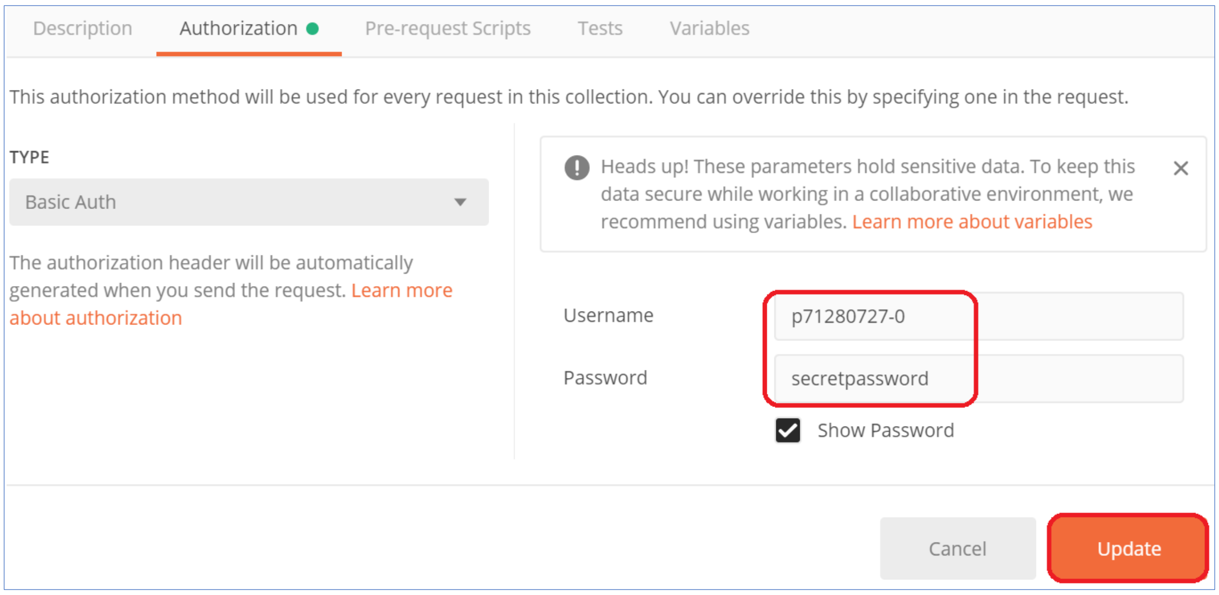
https://testtraining.org is requesting your username and password. The site says: “win.host”

User Name:

Password:

OK | Cancel

Enter the username and password as shown, then click **Update**:


This authorization method will be used for every request in this collection. You can override this by specifying one in the request.

TYPE: Basic Auth

The authorization header will be automatically generated when you send the request. Learn more about authorization

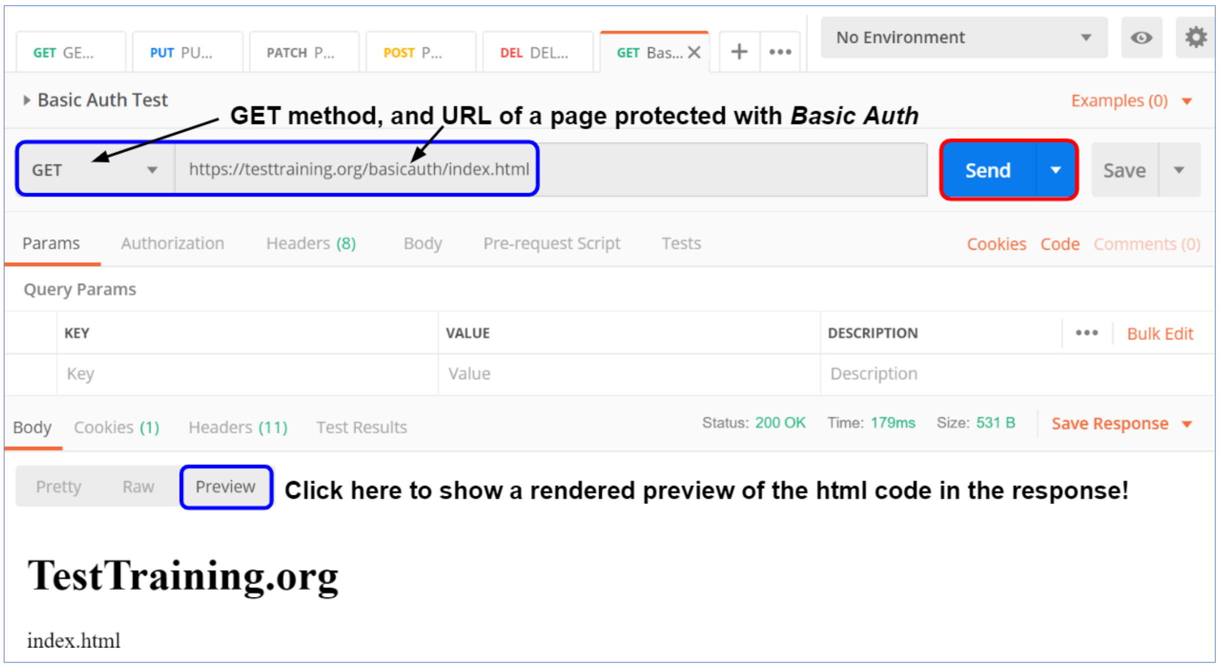
Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. Learn more about variables

Username: p71280727-0

Password: secretpassword

Show Password ✓

To test the configuration, make a GET request to a protected page:



GET https://testtraining.org/basicauth/index.html

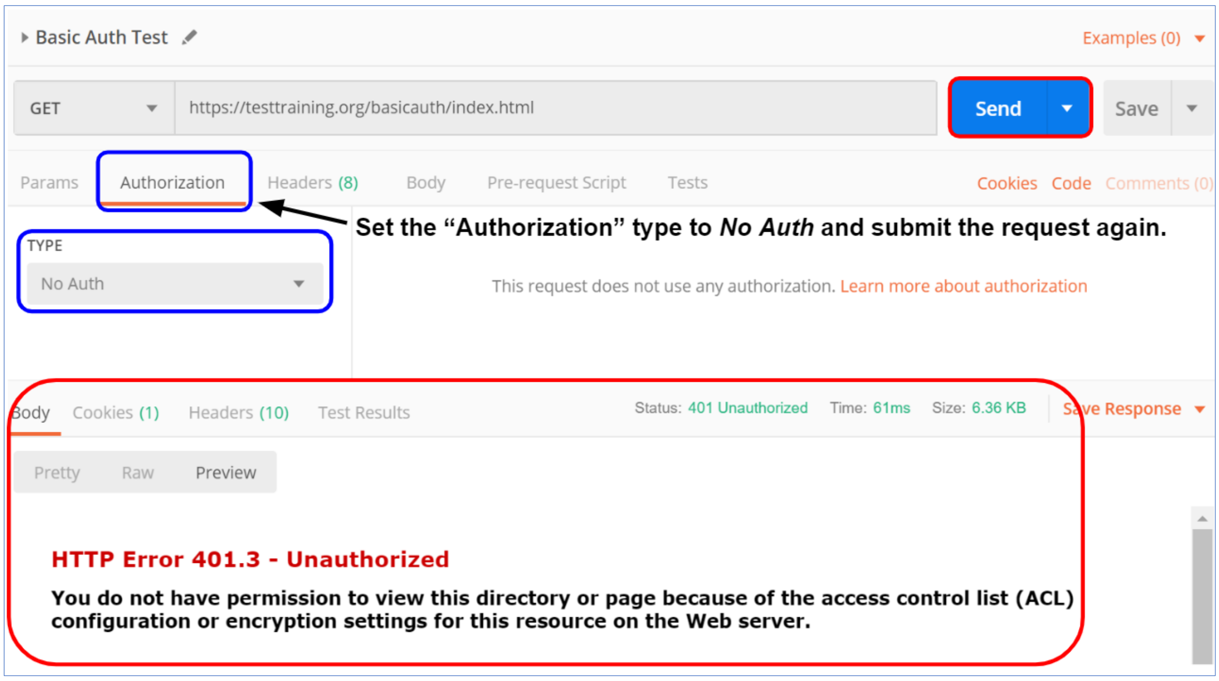
Status: 200 OK Time: 179ms Size: 531 B

TestTraining.org

index.html

To verify that the authorization configuration is working, *turn off* Basic Auth and request the same page again.

Go to the **Authorization** tab of the request, and select “No Auth” from the dropdown:



Set the "Authorization" type to *No Auth* and submit the request again.

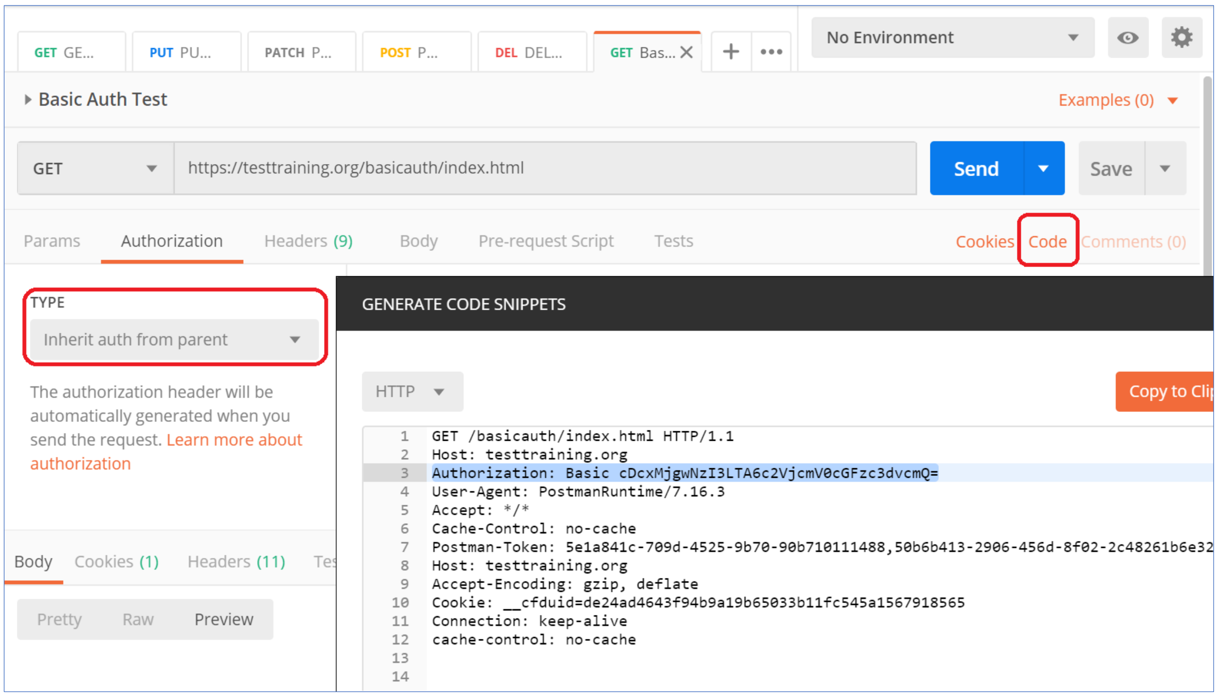
This request does not use any authorization. Learn more about authorization

HTTP Error 401.3 - Unauthorized

You do not have permission to view this directory or page because of the access control list (ACL) configuration or encryption settings for this resource on the Web server.

The web server will now display an error because the request is not authenticated, but the folder requires *Basic Auth* authentication.

After setting the Authorization *type* back to the default value: "Inherit auth from parent", looking at the **code** you can see that this authentication method works by submitting a new *request header* which contains the username and password in Base64-encoded form:



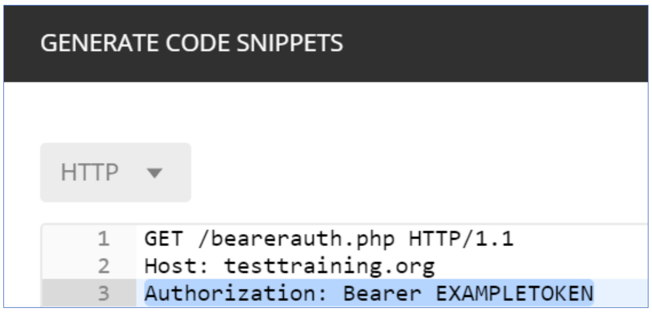
The authorization header will be automatically generated when you send the request. Learn more about authorization

```
GET /basicauth/index.html HTTP/1.1
Host: testtraining.org
Authorization: Basic cDcxMjgwNzI3LTA6c2VjcmV0cGFzc3dvcmQ=
User-Agent: PostmanRuntime/7.16.3
Accept: */*
Cache-Control: no-cache
Postman-Token: 5e1a841c-709d-4525-9b70-90b710111488,50b6b413-2906-456d-8f02-2c48261b6e32
Host: testtraining.org
Accept-Encoding: gzip, deflate
Cookie: __cfduid=de24ad4643f94b9a19b65033b11fc545a1567918565
Connection: keep-alive
cache-control: no-cache
```

Bearer Token

A bearer token is quite similar to the *basic auth*, in that it will be transmitted in the form of a request header, but instead of an “Authorization: Basic” header it will use: “Authorization: Bearer EXAMPLETOKEN”.

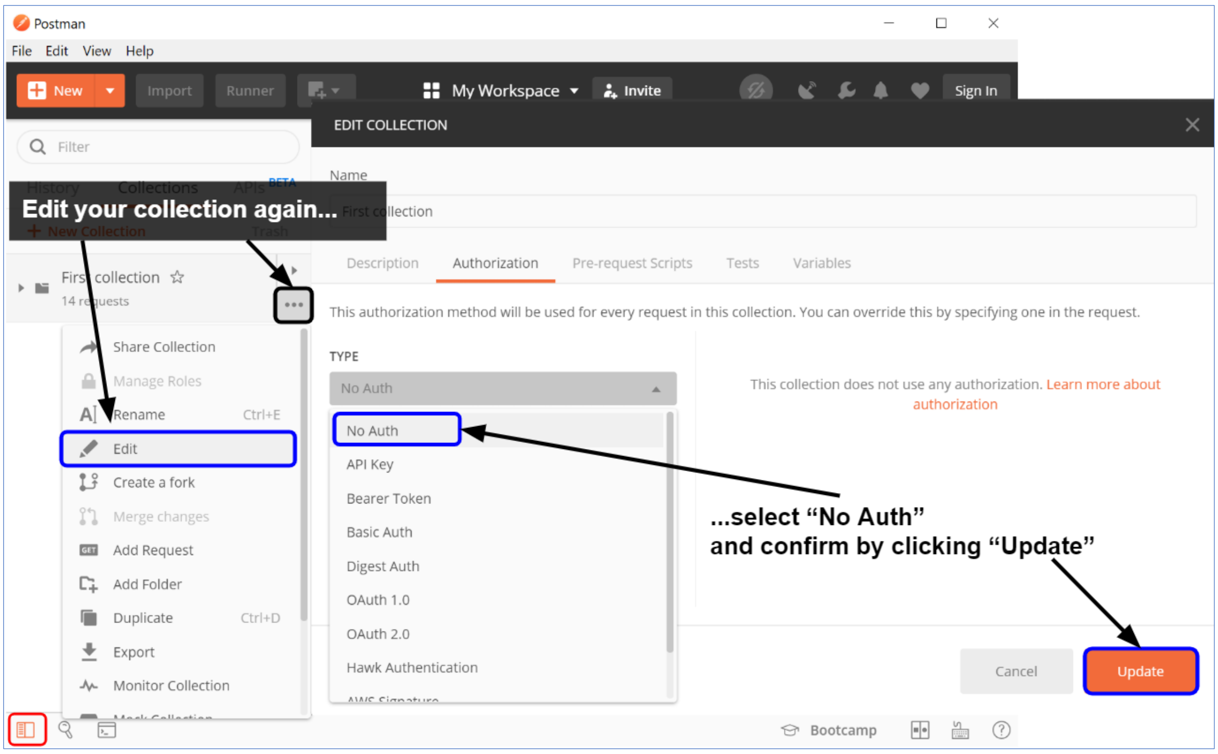
Example:



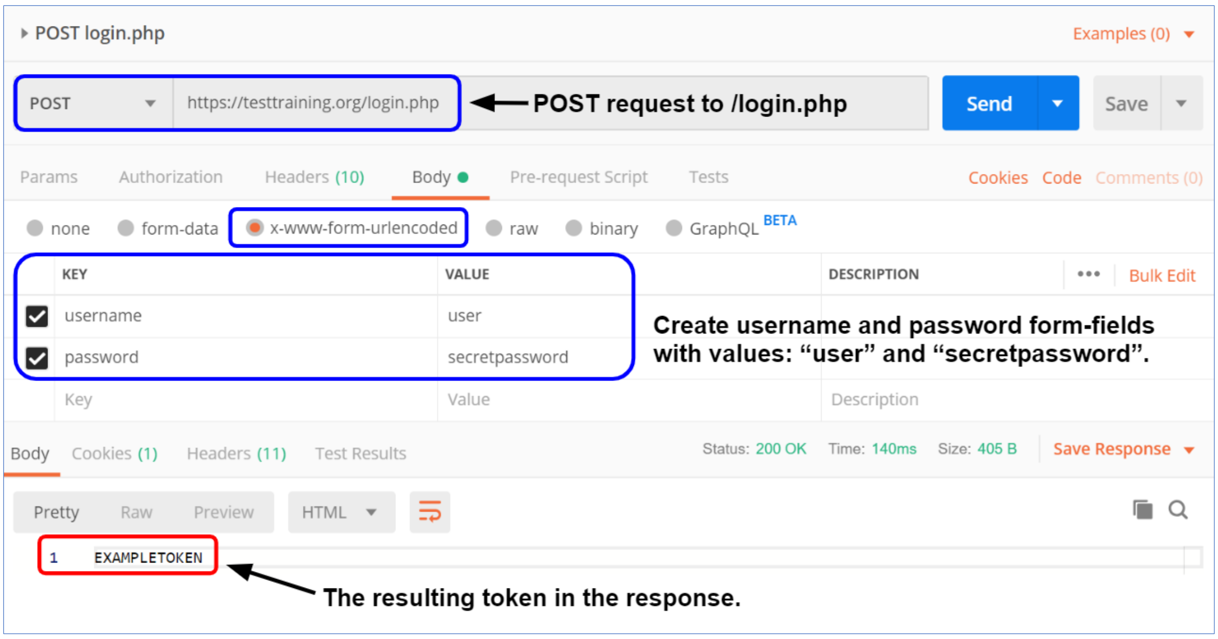
```
GET /bearerauth.php HTTP/1.1
Host: testtraining.org
Authorization: Bearer EXAMPLETOKEN
```

Bearer token is a security token that implies that any party in possession of it (a “bearer”) has access to the protected resource. Normally it will be generated by the server in response to a login request.

Before you can test the Bearer token authorization, it will be required to *undo the collection’s authorization-settings* first. Otherwise the server will attempt to apply the Basic Auth to **all** directories that you attempt to access and fail, because they don’t have the same username/password combination enabled for other folders:



Now, first create a POST login request to request the Bearer token:



POST https://testtraining.org/login.php ← POST request to /login.php

Body: x-www-form-urlencoded

KEY	VALUE
✓ username	user
✓ password	secretpassword
Key	Value

Create username and password form-fields with values: “user” and “secretpassword”.

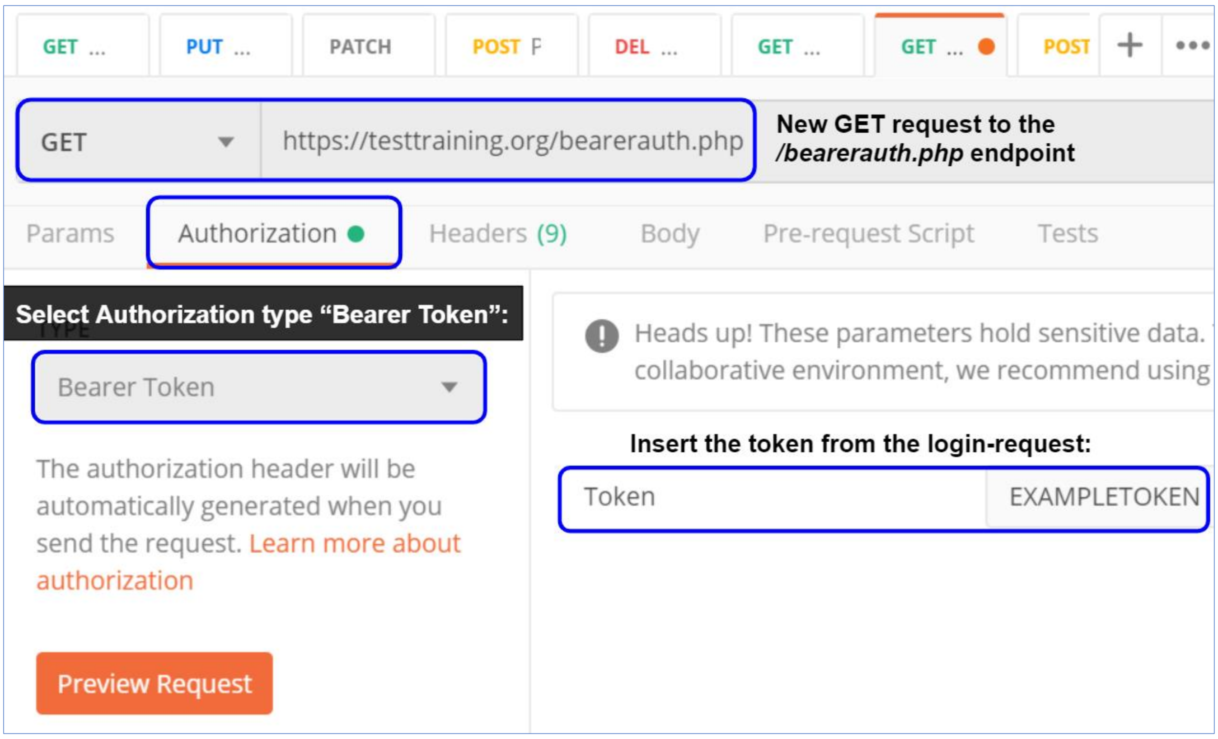
Status: 200 OK Time: 140ms Size: 405 B

```
1   EXAMPLETOKEN
```

The resulting token in the response.

Once you click **Send**, the token “EXAMPLETOKEN” should be displayed in the response. If you submit an invalid username/password combination, the server will present you with a login form instead.

Use the returned token in a new GET request:



GET https://testtraining.org/bearerauth.php

New GET request to the */bearerauth.php* endpoint

Select Authorization type “Bearer Token”:

Bearer Token

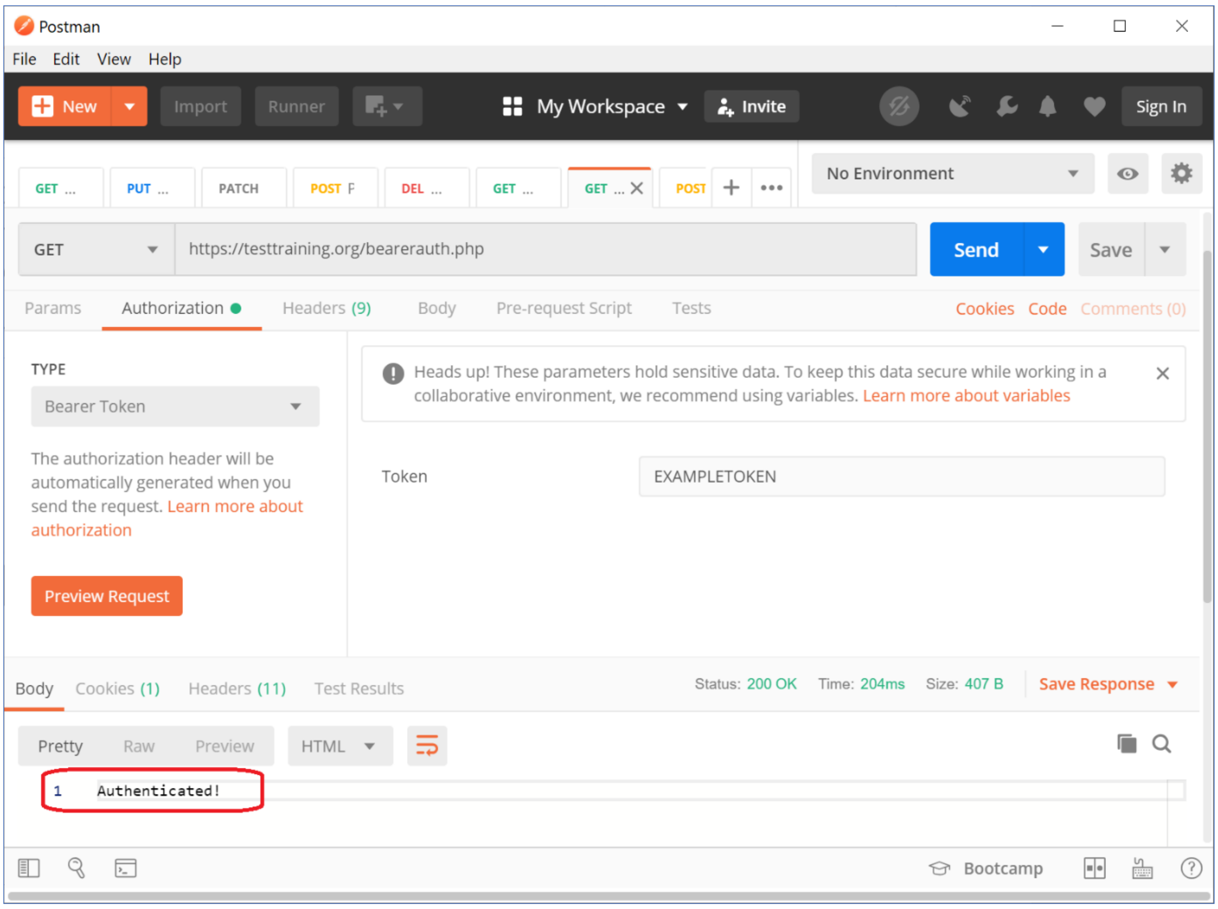
The authorization header will be automatically generated when you send the request. Learn more about authorization

Heads up! These parameters hold sensitive data. collaborative environment, we recommend using

Insert the token from the login-request:

Token EXAMPLETOKEN

When you submit this request (don’t forget to save it to your collection as well) the response will tell you whether the request was successful:



https://testtraining.org/bearerauth.php

TYPE: Bearer Token

The authorization header will be automatically generated when you send the request. Learn more about authorization

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. Learn more about variables

Token: EXAMPLETOKEN

Status: 200 OK Time: 204ms Size: 407 B

```
Authenticated!
```

Success!

What to test for – and how

Most of the testing methods from functional website testing can also be applied to API functional testing.

- Input validation (*Negative testing*):
  After **testing for the expected result first**, try out different values in a request. E.g. submit a random text in a date-field and inspect the response. Is there an unexpected error? Or can the web service handle the failed attempt?
  Submit a request with missing required fields, or too large numbers.
- Roles and permissions:
  How is the API handling requests when you are not properly authenticated? Is there an unexpected error or even a stack trace output?
- Verify the API is updating/deleting data structures:
  When you POST data and inspect the updated data (in a corresponding GET request, or the returned response message), did all the input make its way through? Is something truncated or missing?

After deleting a resource verify that it’s not existing anymore and has been properly deleted.

- Verify all limitations:
  A lot of actions are only allowed to be performed once. For example, you can usually only mark *one* shipping address active at a time in an online shop, or only submit a final order completion *once*. Make sure that limits cannot be circumvented, and that nothing breaks when you still attempt to.
- Performance (if allowed):
  If the customer does not prohibit performance testing, you can also test what happens when you send the same request multiple times in a short period of time, especially larger requests. Or other performance-related items.
- Is the API working according to the specification:
  Verify with the API’s documentation that all requests work as specified.

Common types of APIs / web services

When talking about the “types” of web services, one must differentiate between protocols, architectural styles, languages and *description* languages, as they are easy to confuse.

A **protocol** like *SOAP* or *XML-RPC* describes the valid **syntax** of the API requests and responses.

An **architectural style** or paradigm like *REST* sets rules and goals of an API but does not necessarily need to cover the exact syntax or details of the communication.

Description languages like *WADL* (for RESTful web services) or *WSDL* (for SOAP-based web services) only **describe** the API in a standardized way. They list all API-methods and parameters, and the format of requests and responses. Description languages often use the markup language **XML**.

And **languages** like *GraphQL*, *JSON* or *XML* are used to define the format and syntax of API calls (requests and responses), **or** of the API’s description. That means that an API which has a documentation written in XML, can use JSON or simple query-strings for the actual API-calls. And doesn’t necessarily have to also use XML for that.

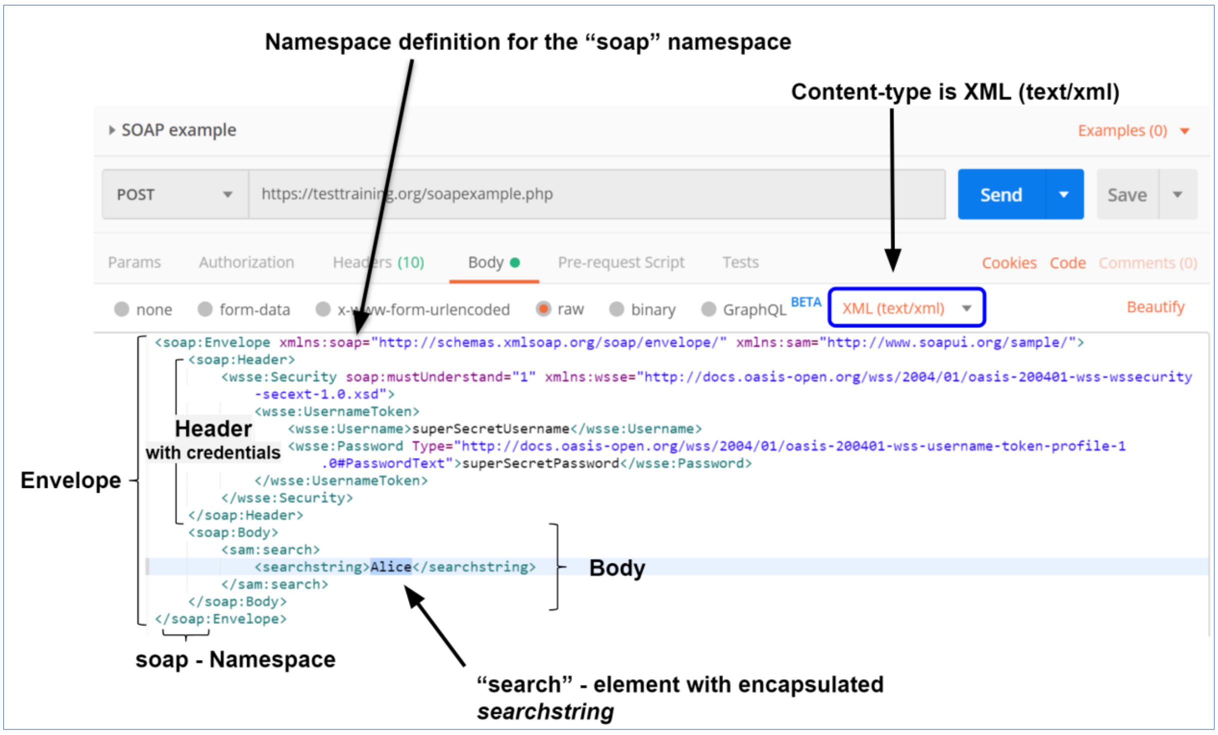
API protocol by example: SOAP

Simple **O**bject **A**ccess **P**rotocol or **SOAP** is a “lightweight protocol intended for exchanging structured information in a decentralized, distributed environment” (https://www.w3.org/TR/soap12-part1/), and a successor of XML-RPC.

Its API-calls are using a variant of the XML markup language for standardized communication.

SOAP APIs are typically described with WSDL.

The *data encapsulation concept* specifies **messages** consisting of an envelope, which contains the **SOAP header** and the **SOAP body**.



POST https://testtraining.org/soapexample.php

XML (text/xml)

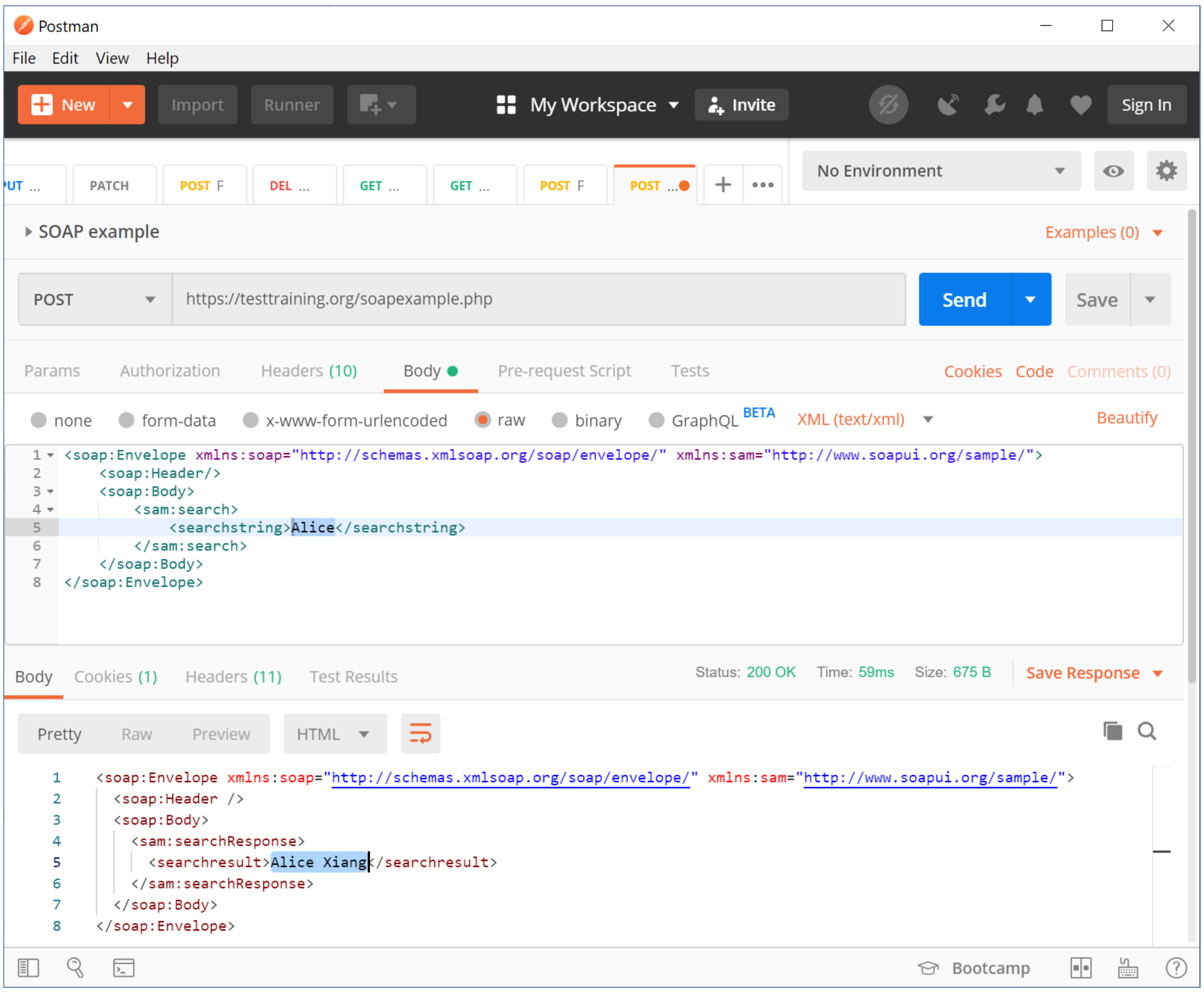
```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" xmlns:sam="http://www.soapui.org/sample/">
  <soap:Header>
    <wsse:Security soap:mustUnderstand="1" xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity
        -secext-1.0.xsd">
      <wsse:UsernameToken>
        <wsse:Username>superSecretUsername</wsse:Username>
        <wsse:Password Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1
            .0#PasswordText">superSecretPassword</wsse:Password>
      </wsse:UsernameToken>
    </wsse:Security>
  </soap:Header>
  <soap:Body>
    <sam:search>
      <searchstring>Alice</searchstring>
    </sam:search>
  </soap:Body>
</soap:Envelope>
```

The raw request (text/xml format) for the SOAP POST call:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:sam="http://www.soapui.org/sample/">
   <soap:Header/>
   <soap:Body>
      <sam:search>
         <searchstring>Alice</searchstring>
      </sam:search>
   </soap:Body>
</soap:Envelope>
```

Once submitted, the API endpoint will search for persons whose first name equals the one defined inside the <searchstring> - tag.

The **response** is also XML (the search result is highlighted):



POST https://testtraining.org/soapexample.php

none form-data x-www-form-urlencoded [✓ raw] binary GraphQL BETA XML (text/xml)

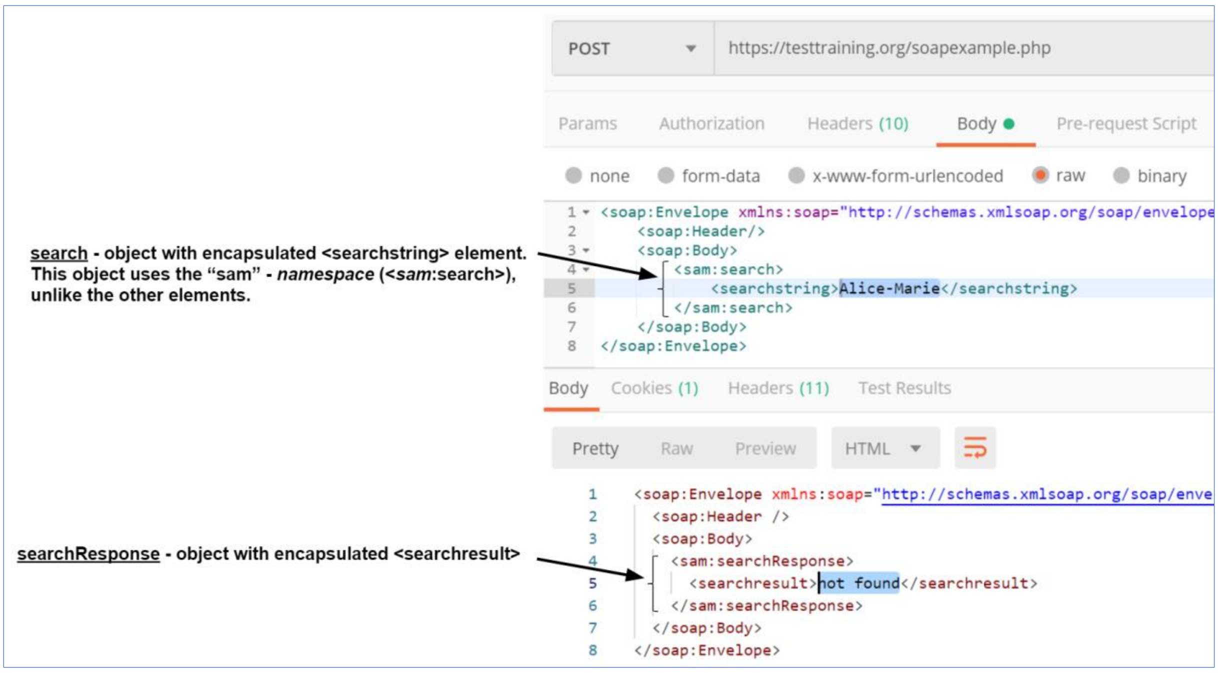
```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" xmlns:sam="http://www.soapui.org/sample/">
    <soap:Header/>
    <soap:Body>
        <sam:search>
            <searchstring>Alice</searchstring>
        </sam:search>
    </soap:Body>
</soap:Envelope>
```

Status: 200 OK Time: 59ms Size: 675 B

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" xmlns:sam="http://www.soapui.org/sample/">
  <soap:Header />
  <soap:Body>
    <sam:searchResponse>
      <searchresult>Alice Xiang</searchresult>
    </sam:searchResponse>
  </soap:Body>
</soap:Envelope>
```

The SOAP response follows the same rules and syntax as the request. But instead of a *<sam:search>* - object it will use *<sam:searchResponse>* to serve the *searchresult*.

If nothing was found, the search-result will show an error message:



search - object with encapsulated <searchstring> element. This object uses the “sam” - *namespace* (<*sam*:search>), unlike the other elements.

searchResponse - object with encapsulated <searchresult>

POST https://testtraining.org/soapexample.php

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope
    <soap:Header/>
    <soap:Body>
        <sam:search>
            <searchstring>Alice-Marie</searchstring>
        </sam:search>
    </soap:Body>
</soap:Envelope>
```

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/enve
  <soap:Header />
  <soap:Body>
    <sam:searchResponse>
      <searchresult>not found</searchresult>
    </sam:searchResponse>
  </soap:Body>
</soap:Envelope>
```

Description language by example: WADL

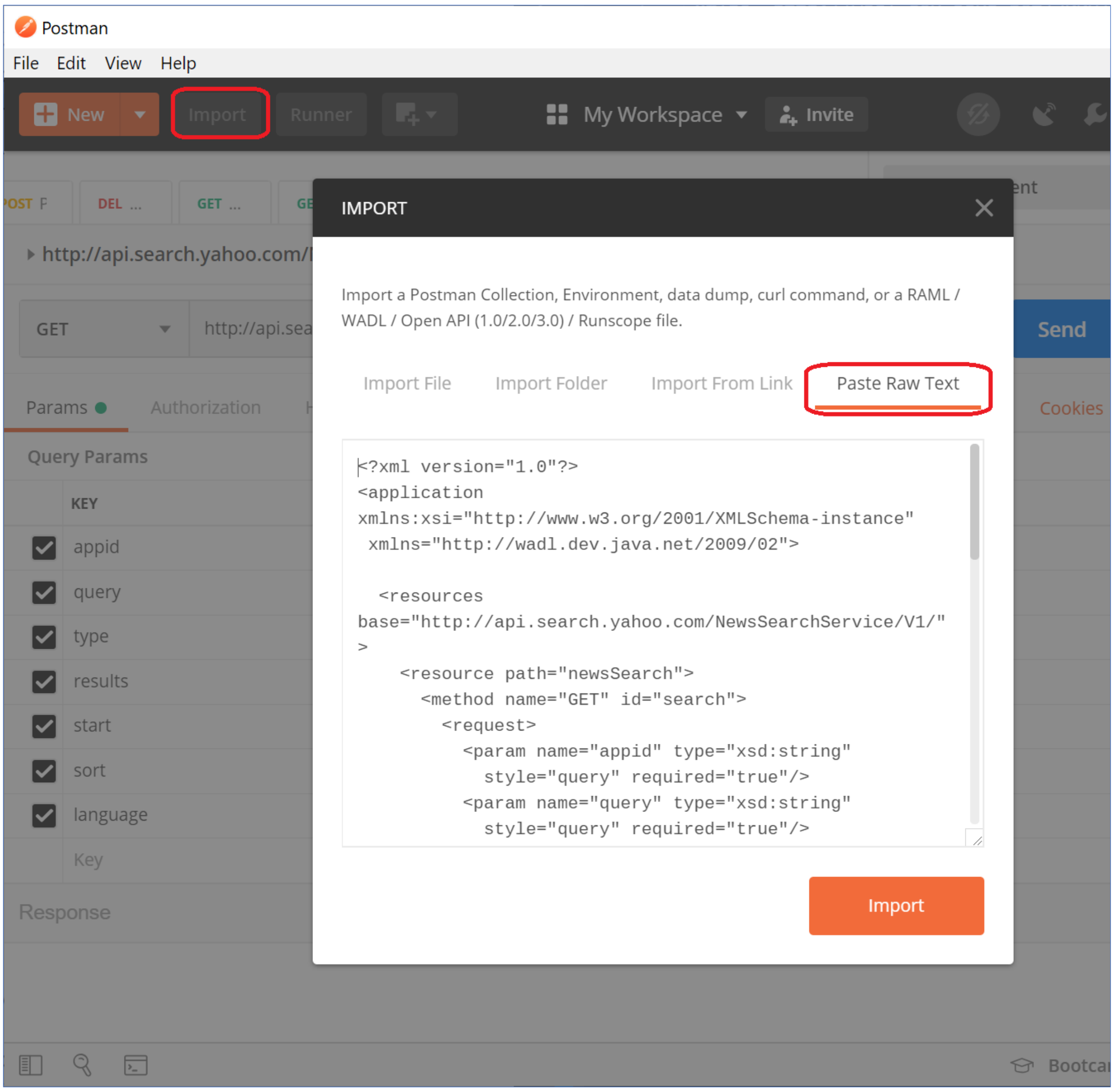
The *Web Application Description Language* is a standardized method of describing HTTP-based web applications. It has been designed to support application modelling and visualization, code generation as well as the configuration of client and server.

WADL is based on XML - just like SOAP. A WADL description file consists of the required root element “application”, the “resources” elements and a couple of other optional elements:

```
<?xml version="1.0"?>
<application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns="http://wadl.dev.java.net/2009/02">

  <resources base="http://api.search.yahoo.com/NewsSearchService/V1/">
    <resource path="newsSearch">
      <method name="GET" id="search">
        <request>
          <param name="appid" type="xsd:string"
            style="query" required="true"/>
          <param name="query" type="xsd:string"
            style="query" required="true"/>
          <param name="type" style="query" default="all">
            <option value="all"/>
            <option value="any"/>
            <option value="phrase"/>
          </param>
          <param name="results" style="query" type="xsd:int" default="10"/>
          <param name="start" style="query" type="xsd:int" default="1"/>
          <param name="sort" style="query" default="rank">
            <option value="rank"/>
            <option value="date"/>
          </param>
          <param name="language" style="query" type="xsd:string"/>
        </request>
        <response status="200">
          <representation mediaType="application/xml"
            element="yn:ResultSet"/>
        </response>
        <response status="400">
          <representation mediaType="application/xml"
            element="ya:Error"/>
        </response>
      </method>
    </resource>
  </resources>

</application>
```

Also required: the highlighted namespace attribute
xmlns="http://wadl.dev.java.net/2009/02"

Postman allows you to import valid WADL directly, and generates a new collection from it. This example is taken from the official WADL specification at https://www.w3.org/Submission/wadl/.

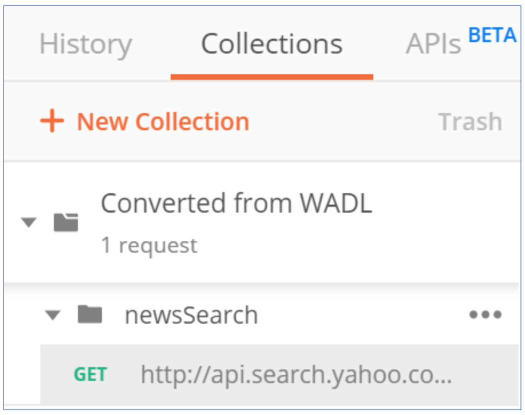


Import a Postman Collection, Environment, data dump, curl command, or a RAML / WADL / Open API (1.0/2.0/3.0) / Runscope file.

```
<?xml version="1.0"?>
<application
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns="http://wadl.dev.java.net/2009/02">

  <resources
base="http://api.search.yahoo.com/NewsSearchService/V1/"
>
    <resource path="newsSearch">
      <method name="GET" id="search">
        <request>
          <param name="appid" type="xsd:string"
            style="query" required="true"/>
          <param name="query" type="xsd:string"
            style="query" required="true"/>
```

Click the **Import** button at the top of the Postman UI, and then either open the WADL file from the “Import File” dialog, or choose the “Paste Raw Text” option to copy-paste the code directly.

Provided that the syntax of the WADL was correct, Postman will store all requests from the documentation into a new collection:



Language by example: JSON

JSON is an acronym for "JavaScript Object Notation". Like the name suggests, each piece of JSON 'code' is actually a valid javascript object.

It has many advantages: it can easily be converted into other languages, it has very little overhead, a JSON object consists only of very few characters – apart from the actual payload (the data that you submit to or from the API). And therefore, its readability is also great – once you have learned the pretty simple syntax:

```
{ "name": "John", "age": 31, "city": "New York" }
```

- The entire JSON object is enclosed in curly brackets.
- The individual parameters are separated with commas.
- Key-value pairs are separated with colons as in *"key": "value"*.
- Text (*strings*) have to be in quotes: *"New York"*.

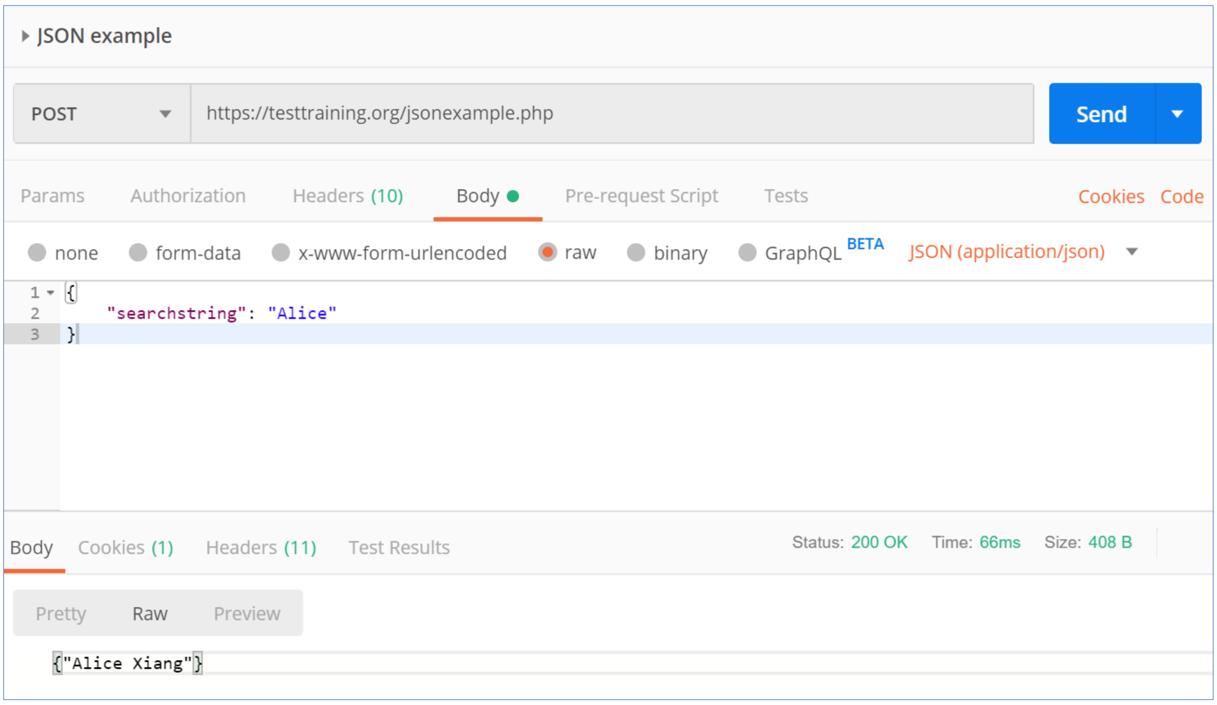
Multiple data types are allowed, e.g. strings, numbers, arrays, and other JSON objects (enclosed in curly brackets).

<u>Example:</u>

```
{"gigs":[
        {"name":"pop concert", "description":"a wonderful evening"},
        {"name":"festival", "description":"a weekend full of fun"}
]}
```

There is no requirement for spacing or padding, but to make it more readable it is recommended to use some indentation and to put each object into its own line. All white space that is not part of a value (inside quotes) will usually be cropped and dropped by the server.

This is the same request as before (searching for persons by first name) but with JSON this time:



POST https://testtraining.org/jsonexample.php

raw JSON (application/json)

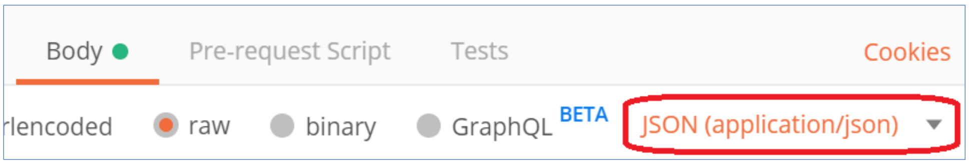
```
{
    "searchstring": "Alice"
}
```

Status: 200 OK Time: 66ms Size: 408 B

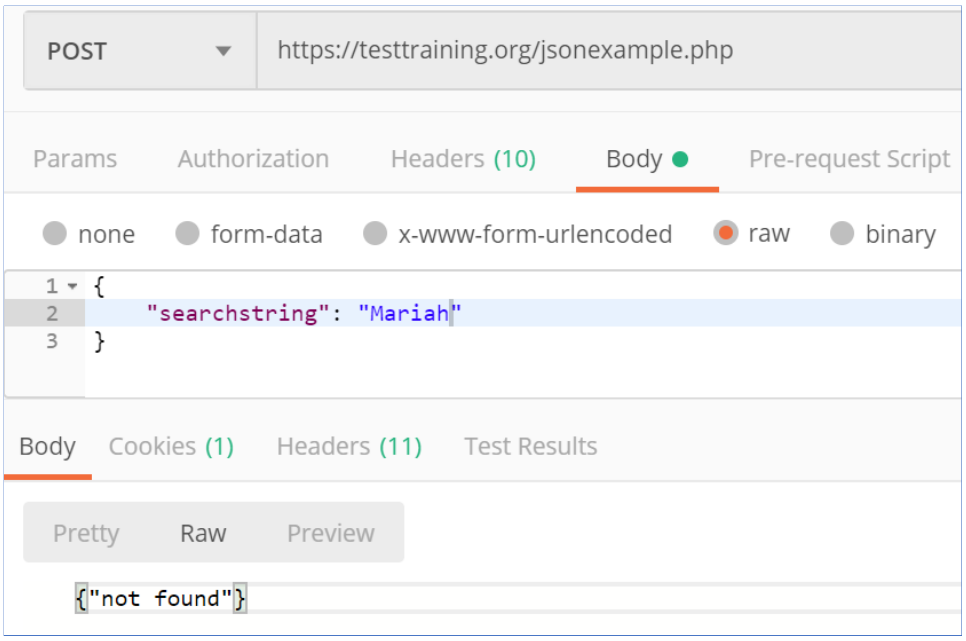
```
{"Alice Xiang"}
```

The result is also a JSON object: *{"Alice Xiang"}.*

Make sure to select the proper Content-Type: *application/json* for the request.



When there is no valid result for the given *searchstring*, you will get this "*not found*" - response:



https://testtraining.org/jsonexample.php

Body: raw

```
{
    "searchstring": "Mariah"
}
```

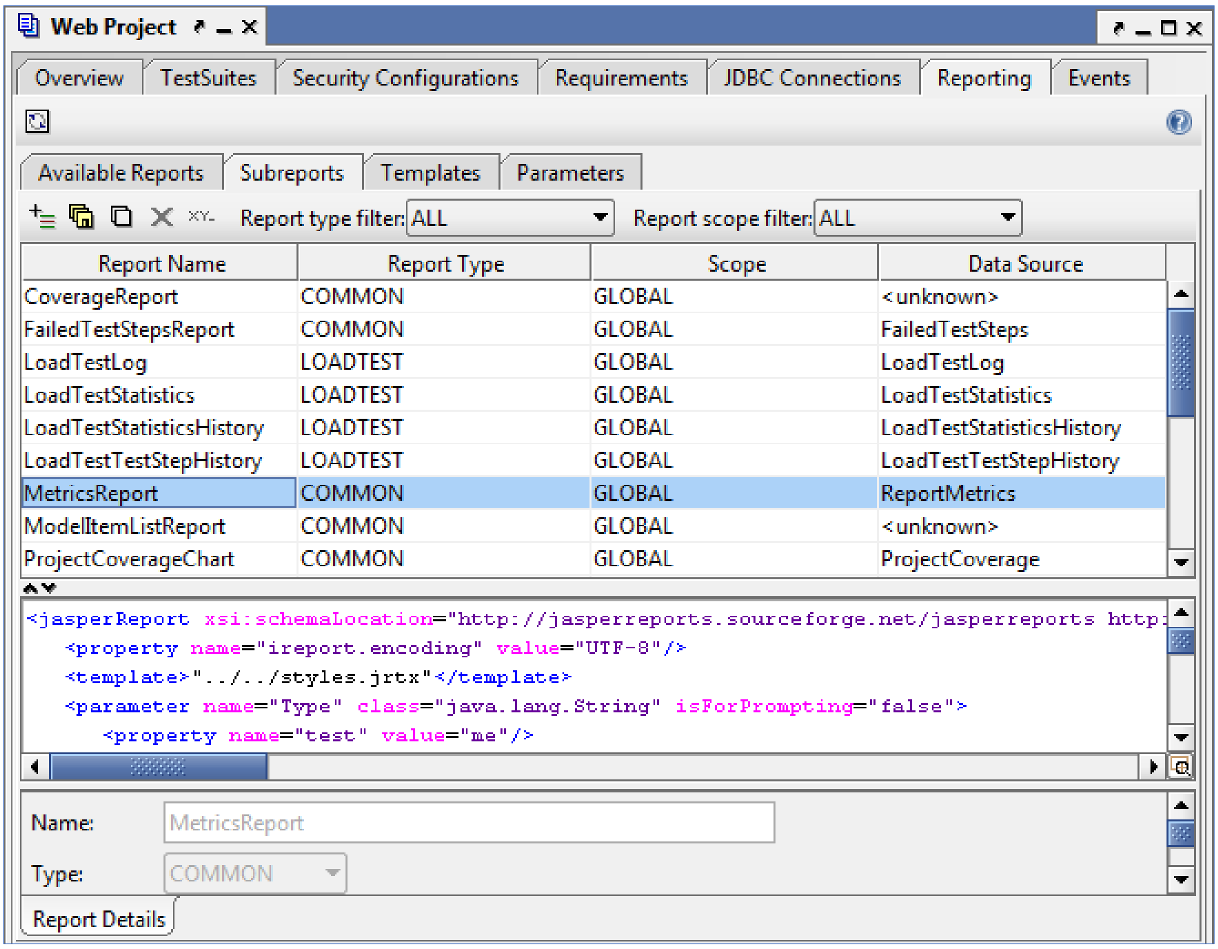
```
{"not found"}
```

Other interesting API testing tools and technology

In this course you learned how to test web services using Postman. There are, of course, other tools as well that you should at least have heard of.

SoapUI

SoapUI is a testing application for both, REST APIs and SOAP APIs. The basic version is open source and free. It's main use is for the design and execution of automated tests, but it can also be used for manual testing.



Report Name	Report Type	Scope	Data Source
CoverageReport	COMMON	GLOBAL	<unknown>
FailedTestStepsReport	COMMON	GLOBAL	FailedTestSteps
LoadTestLog	LOADTEST	GLOBAL	LoadTestLog
LoadTestStatistics	LOADTEST	GLOBAL	LoadTestStatistics
LoadTestStatisticsHistory	LOADTEST	GLOBAL	LoadTestStatisticsHistory
LoadTestTestStepHistory	LOADTEST	GLOBAL	LoadTestTestStepHistory
MetricsReport	COMMON	GLOBAL	ReportMetrics
ModelItemListReport	COMMON	GLOBAL	<unknown>
ProjectCoverageChart	COMMON	GLOBAL	ProjectCoverage

```
<jasperReport xsi:schemaLocation="http://jasperreports.sourceforge.net/jasperreports http:
   <property name="ireport.encoding" value="UTF-8"/>
   <template>"../../styles.jrtx"</template>
   <parameter name="Type" class="java.lang.String" isForPrompting="false">
      <property name="test" value="me"/>
```

Name: MetricsReport

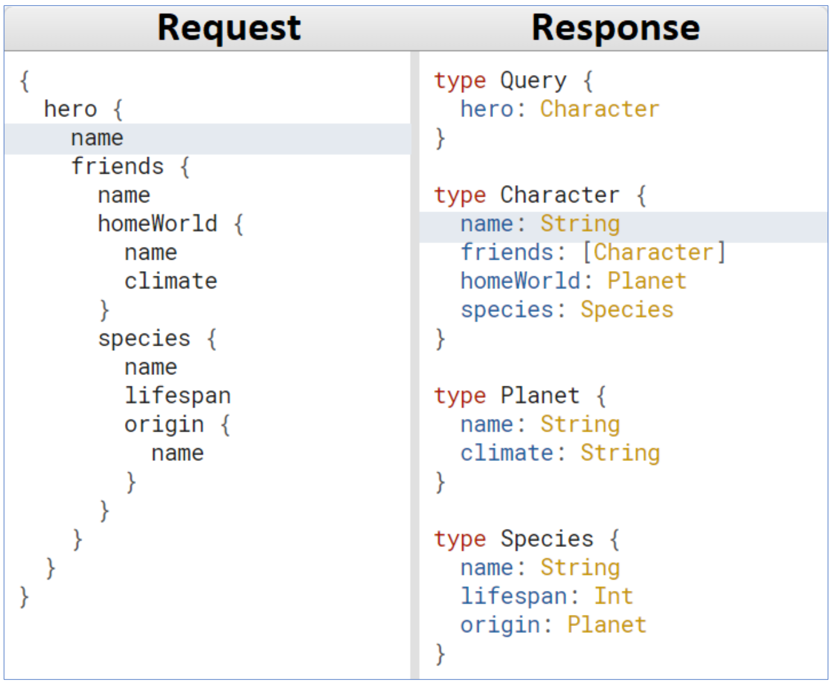
Type: COMMON

The Soap UI interface

GraphQL

Originally designed by and for Facebook, GraphQL quickly became a popular language for web services. This open source technology is hosted by the non-profit *Linux Foundation*.

The language has its own GraphQL Schema Definition Language. It looks somewhat similar to JSON. But in addition to being a pure format for the transmission of data, it has its own very powerful query and data-modification language.


Request
Response

```
{
  hero {
    name
    friends {
      name
      homeWorld {
        name
        climate
      }
      species {
        name
        lifespan
        origin {
          name
        }
      }
    }
  }
}
```

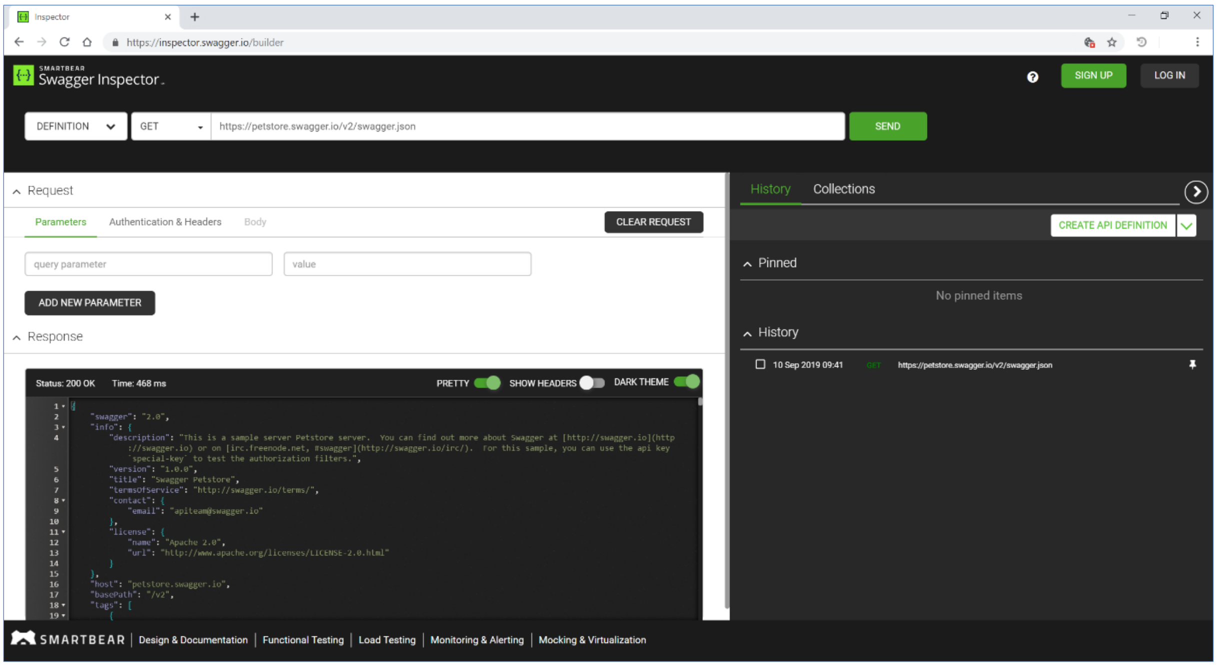
```
type Query {
  hero: Character
}

type Character {
  name: String
  friends: [Character]
  homeWorld: Planet
  species: Species
}

type Planet {
  name: String
  climate: String
}

type Species {
  name: String
  lifespan: Int
  origin: Planet
}
```

A GraphQL request and response

Swagger UI

Another open source project that helps rendering web services documentation.

It consists of a collection of HTML, Javascript and CSS assets, and can automatically generate a browsable web documentation from any compliant API.



Status: 200 OK   Time: 468 ms

```
{
    "swagger": "2.0",
    "info": {
        "description": "This is a sample server Petstore server.  You can find out more about Swagger at [http://swagger.io](http://swagger.io) or on [irc.freenode.net, #swagger](http://swagger.io/irc/).  For this sample, you can use the api key `special-key` to test the authorization filters.",
        "version": "1.0.0",
        "title": "Swagger Petstore",
        "termsOfService": "http://swagger.io/terms/",
        "contact": {
            "email": "apiteam@swagger.io"
        },
        "license": {
            "name": "Apache 2.0",
            "url": "http://www.apache.org/licenses/LICENSE-2.0.html"
        }
    },
    "host": "petstore.swagger.io",
    "basePath": "/v2",
    "tags": [
        {
```

Swagger UI example