Hurl Documentation

Version 6.1.0 - 12-03-2025

Table of Contents

Introduction

What's Hurl?

Hurl is a command line tool that runs **HTTP requests** defined in a simple **plain text format**.

It can chain requests, capture values and evaluate queries on headers and body response. Hurl is very versatile: it can be used for both **fetching data** and **testing HTTP** sessions.

Hurl makes it easy to work with **HTML** content, **REST / SOAP / GraphQL** APIs, or any other **XML / JSON** based APIs.

```
# Get home:
GET https://example.org
HTTP 200
[Captures]
csrf_token: xpath "string(//meta[@name='_csrf_token']/@content)"


# Do login!
POST https://example.org/login?user=toto&password=1234
X-CSRF-TOKEN: {{csrf_token}}
HTTP 302
```

Chaining multiple requests is easy:

```
GET https://example.org/api/health
GET https://example.org/api/step1
GET https://example.org/api/step2
GET https://example.org/api/step3
```

Also an HTTP Test Tool

Hurl can run HTTP requests but can also be used to **test HTTP responses**. Different types of queries and predicates are supported, from [XPath](#) and [JSONPath](#) on body response, to assert on status code and response headers.

It is well adapted for **REST / JSON APIs**

```
POST https://example.org/api/tests
{
    "id": "4568",
    "evaluate": true
}
HTTP 200
[Asserts]
header "X-Frame-Options" == "SAMEORIGIN"
jsonpath "$.status" == "RUNNING"    # Check the status code
jsonpath "$.tests" count == 25      # Check the number of items
jsonpath "$.id" matches /\d{4}/     # Check the format of the id
```

HTML content

```
GET https://example.org
HTTP 200
[Asserts]
xpath "normalize-space(//head/title)" == "Hello world!"
```

GraphQL

```
POST https://example.org/graphql
```graphql
{
  human(id: "1000") {
    name
    height(unit: FOOT)
  }
}
```
HTTP 200
```

and even **SOAP APIs**

```
POST https://example.org/InStock
Content-Type: application/soap+xml; charset=utf-8
SOAPAction: "http://www.w3.org/2003/05/soap-envelope"
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope" xmlns:m="http
  <soap:Header></soap:Header>
  <soap:Body>
    <m:GetStockPrice>
      <m:StockName>GOOG</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
HTTP 200
```

Hurl can also be used to test the **performance** of HTTP endpoints

```
GET https://example.org/api/v1/pets
HTTP 200
[Asserts]
duration < 1000  # Duration in ms
```

And check response bytes

```
GET https://example.org/data.tar.gz
HTTP 200
[Asserts]
sha256 == hex,039058c6f2c0cb492c533b0a4d14ef77cc0f78abccced5287d84a1a2011cfb81;
```

Finally, Hurl is easy to **integrate in CI/CD**, with text, JUnit, TAP and HTML reports

Why Hurl?

Text Format

For both devops and developers

Fast CLI

A command line for local dev and continuous integration

Single Binary

Easy to install, with no runtime required

Powered by curl

Hurl is a lightweight binary written in [Rust](#). Under the hood, Hurl HTTP engine is powered by [libcurl](#), one of the most powerful and reliable file transfer libraries. With its text file format, Hurl adds syntactic sugar to run and test HTTP requests, but it's still the [curl](#) that we love: **fast**, **efficient** and **IPv6 / HTTP/3 ready**.

Feedbacks

To support its development, [star Hurl on GitHub](#)!

[Feedback, suggestion, bugs or improvements](#) are welcome.

```
POST https://hurl.dev/api/feedback
{
  "name": "John Doe",
  "feedback": "Hurl is awesome!"
}
HTTP 200
```

Resources

[License](#)

[Blog](#)

[Tutorial](#)

[Documentation](#)

[GitHub](#)

Getting Started

Installation

Binaries Installation

Linux

Precompiled binary is available at [Hurl latest GitHub release](#):

```
$ INSTALL_DIR=/tmp
$ VERSION=6.1.0
$ curl --silent --location https://github.com/Orange-OpenSource/hurl/releases/dow
$ export PATH=$INSTALL_DIR/hurl-$VERSION-x86_64-unknown-linux-gnu/bin:$PATH
```

Debian / Ubuntu

For Debian / Ubuntu, Hurl can be installed using a binary .deb file provided in each Hurl release.

```
$ VERSION=6.1.0
$ curl --location --remote-name https://github.com/Orange-OpenSource/hurl/release
$ sudo apt update && sudo apt install ./hurl_${VERSION}_amd64.deb
```

For Ubuntu (bionic, focal, jammy, noble), Hurl can be installed from `ppa:lepapareil/hurl`

```
$ VERSION=6.1.0
$ sudo apt-add-repository -y ppa:lepapareil/hurl
$ sudo apt install hurl="${VERSION}"*
```

Alpine

Hurl is available on `testing` channel.

```
$ apk add --repository http://dl-cdn.alpinelinux.org/alpine/edge/testing hurl
```

Arch Linux / Manjaro

Hurl is available on [extra](#) channel.

```
$ pacman -Sy hurl
```

NixOS / Nix

[NixOS / Nix package](#) is available on stable channel.

macOS

Precompiled binaries for Intel and ARM CPUs are available at [Hurl latest GitHub release](#).

Homebrew

```
$ brew install hurl
```

MacPorts

```
$ sudo port install hurl
```

FreeBSD

```
$ sudo pkg install hurl
```

Windows

Windows requires the [Visual C++ Redistributable Package](#) to be installed manually, as this is not included in the installer.

Zip File

Hurl can be installed from a standalone zip file at [Hurl latest GitHub release](#). You will need to update your PATH variable.

Installer

An executable installer is also available at [Hurl latest GitHub release](#).

Chocolatey

```
$ choco install hurl
```

Scoop

```
$ scoop install hurl
```

Windows Package Manager

```
$ winget install hurl
```

Cargo

If you're a Rust programmer, Hurl can be installed with cargo.

```
$ cargo install hurl
```

conda-forge

```
$ conda install -c conda-forge hurl
```

Hurl can also be installed with [conda-forge](#) powered package manager like [pixi](#).

Docker

```
$ docker pull ghcr.io/orange-opensource/hurl:latest
```

npm

```
$ npm install --save-dev @orangeopensource/hurl
```

Building From Sources

Hurl sources are available in [GitHub](#).

Build on Linux

Hurl depends on libssl, libcurl and libxml2 native libraries. You will need their development files in your platform.

Debian based distributions

```
$ apt install -y build-essential pkg-config libssl-dev libcurl4-openssl-dev libxr
```

Fedora based distributions

```
$ dnf install -y pkgconf-pkg-config gcc openssl-devel libxml2-devel
```

Red Hat based distributions

```
$ yum install -y pkg-config gcc openssl-devel libxml2-devel
```

Arch based distributions

```
$ pacman -S --noconfirm pkgconf gcc glibc openssl libxml2
```

Alpine based distributions

```
$ apk add curl-dev gcc libxml2-dev musl-dev openssl-dev
```

Build on macOS

```
$ xcode-select --install
$ brew install pkg-config
```

Hurl is written in [Rust](#). You should [install](#) the latest stable release.

```
$ curl https://sh.rustup.rs -sSf | sh -s -- -y
$ source $HOME/.cargo/env
$ rustc --version
$ cargo --version
```

Then build hurl:

```
$ git clone https://github.com/Orange-OpenSource/hurl
$ cd hurl
$ cargo build --release
$ ./target/release/hurl --version
```

Build on Windows

Please follow the [contrib on Windows section](#).

Manual

Name

hurl - run and test HTTP requests.

Synopsis

hurl [options] [FILE...]

Description

Hurl is a command line tool that runs HTTP requests defined in a simple plain text format.

It can chain requests, capture values and evaluate queries on headers and body response. Hurl is very versatile, it can be used for fetching data and testing HTTP sessions: HTML content, REST / SOAP / GraphQL APIs, or any other XML / JSON based APIs.

```
$ hurl session.hurl
```

If no input files are specified, input is read from stdin.

```
$ echo GET http://httpbin.org/get | hurl
    {
      "args": {},
      "headers": {
        "Accept": "*/*",
        "Accept-Encoding": "gzip",
        "Content-Length": "0",
        "Host": "httpbin.org",
        "User-Agent": "hurl/0.99.10",
        "X-Amzn-Trace-Id": "Root=1-5eedf4c7-520814d64e2f9249ea44e0"
      },
      "origin": "1.2.3.4",
      "url": "http://httpbin.org/get"
    }
```

Hurl can take files as input, or directories. In the latter case, Hurl will search files with `.hurl` extension recursively.

Output goes to stdout by default. To have output go to a file, use the [-o, --output](#) option:

```
$ hurl -o output input.hurl
```

By default, Hurl executes all HTTP requests and outputs the response body of the last HTTP call.

To have a test oriented output, you can use --test option:

```
$ hurl --test *.hurl
```

Hurl File Format

The Hurl file format is fully documented in https://hurl.dev/docs/hurl-file.html

It consists of one or several HTTP requests

```
GET http://example.org/endpoint1
GET http://example.org/endpoint2
```

Capturing values

A value from an HTTP response can be-reused for successive HTTP requests.

A typical example occurs with CSRF tokens.

```
GET https://example.org
HTTP 200
# Capture the CSRF token value from html body.
[Captures]
csrf_token: xpath "normalize-space(//meta[@name='_csrf_token']/@content)"

# Do the login !
POST https://example.org/login?user=toto&password=1234
X-CSRF-TOKEN: {{csrf_token}}
```

More information on captures can be found here https://hurl.dev/docs/capturing-response.html

Asserts

The HTTP response defined in the Hurl file are used to make asserts. Responses are optional.

At the minimum, response includes assert on the HTTP status code.

```
GET http://example.org
HTTP 301
```

It can also include asserts on the response headers

```
GET http://example.org
HTTP 301
Location: http://www.example.org
```

Explicit asserts can be included by combining a query and a predicate

```
GET http://example.org
HTTP 301
[Asserts]
xpath "string(//title)" == "301 Moved"
```

With the addition of asserts, Hurl can be used as a testing tool to run scenarios.

More information on asserts can be found here https://hurl.dev/docs/asserting-response.html

Options

Options that exist in curl have exactly the same semantics.

Options specified on the command line are defined for every Hurl file's entry, except if they are tagged as cli-only (can not be defined in the Hurl request [Options] entry)

For instance:

```
$ hurl --location foo.hurl
```

will follow redirection for each entry in foo.hurl. You can also define an option only for a particular entry with an [Options] section. For instance, this Hurl file:

```
GET https://example.org
HTTP 301

GET https://example.org
[Options]
location: true
HTTP 200
```

will follow a redirection only for the second entry.

Option	Description
--aws-sigv4 <PROVIDER1[:PROVIDER2[:REGION[:SERVICE]]]>	Generate an Authorization header with an AWS SigV4 signature. Use -u, --user to specify Access Key Id (username) and Secret Key (password). To use temporary session credentials (e.g. for an AWS IAM Role), add the X-Amz-Security-Token header containing the session token.
--cacert <FILE>	Specifies the certificate file for peer verification. The file may contain multiple CA certificates and must be in PEM format. Normally Hurl is built to use a default file for this, so this option is typically used to alter that default

	file.
-E, --cert <CERTIFICATE[:PASSWORD]>	Client certificate file and password. See also --key.
--color	Colorize debug output (the HTTP response output is not colorized). This is a cli-only option.
--compressed	Request a compressed response using one of the algorithms br, gzip, deflate and automatically decompress the content.
--connect-timeout <SECONDS>	Maximum time in seconds that you allow Hurl's connection to take. You can specify time units in the connect timeout expression. Set Hurl to use a connect timeout of 20 seconds with --connect-timeout 20s or set it to 35,000 milliseconds with --connect-timeout 35000ms. No spaces allowed. See also -m, --max-time.
--connect-to <HOST1:PORT1:HOST2:PORT2>	For a request to the given HOST1:PORT1 pair, connect to HOST2:PORT2 instead. This option can be used several times in a command line. See also --resolve.
--continue-on-error	Continue executing requests to the end of the Hurl file even when an assert error occurs. By default, Hurl exits after an assert error in the HTTP response. Note that this option does not affect the behavior with multiple input Hurl files. All the input files are executed independently. The result of one file does not affect the execution of the other Hurl files. This is a cli-only option.
-b, --cookie <FILE>	Read cookies from FILE (using the Netscape cookie file format). Combined with -c, --cookie-jar, you can simulate a cookie storage between successive Hurl runs. This is a cli-only option.
	Write cookies to FILE after running

-c, --cookie-jar <FILE>	the session. The file will be written using the Netscape cookie file format. Combined with -b, --cookie, you can simulate a cookie storage between successive Hurl runs. This is a cli-only option.
--curl <FILE>	Export each request to a list of curl commands. This is a cli-only option.
--delay <MILLISECONDS>	Sets delay before each request (aka sleep). The delay is not applied to requests that have been retried because of --retry. See --retry-interval to space retried requests. You can specify time units in the delay expression. Set Hurl to use a delay of 2 seconds with --delay 2s or set it to 500 milliseconds with --delay 500ms. No spaces allowed.
--error-format <FORMAT>	Control the format of error message (short by default or long) This is a cli-only option.
--file-root <DIR>	Set root directory to import files in Hurl. This is used for files in multipart form data, request body and response output. When it is not explicitly defined, files are relative to the Hurl file's directory. This is a cli-only option.
--from-entry <ENTRY_NUMBER>	Execute Hurl file from ENTRY_NUMBER (starting at 1). This is a cli-only option.
--glob <GLOB>	Specify input files that match the given glob pattern. Multiple glob flags may be used. This flag supports common Unix glob patterns like *, ? and []. However, to avoid your shell accidentally expanding glob patterns before Hurl handles them, you must use single quotes or double quotes around each pattern. This is a cli-only option.

-H, --header <HEADER>	Add an extra header to include in information sent. Can be used several times in a command Do not add newlines or carriage returns
-0, --http1.0	Tells Hurl to use HTTP version 1.0 instead of using its internally preferred HTTP version.
--http1.1	Tells Hurl to use HTTP version 1.1.
--http2	Tells Hurl to use HTTP version 2. For HTTPS, this means Hurl negotiates HTTP/2 in the TLS handshake. Hurl does this by default. For HTTP, this means Hurl attempts to upgrade the request to HTTP/2 using the Upgrade: request header.
--http3	Tells Hurl to try HTTP/3 to the host in the URL, but fallback to earlier HTTP versions if the HTTP/3 connection establishment fails. HTTP/3 is only available for HTTPS and not for HTTP URLs.
--ignore-asserts	Ignore all asserts defined in the Hurl file. This is a cli-only option.
-i, --include	Include the HTTP headers in the output This is a cli-only option.
-k, --insecure	This option explicitly allows Hurl to perform "insecure" SSL connections and transfers.
--interactive	Stop between requests. This is similar to a break point, You can then continue (Press C) or quit (Press Q). This is a cli-only option.
-4, --ipv4	This option tells Hurl to use IPv4 addresses only when resolving host names, and not for example try IPv6.
-6, --ipv6	This option tells Hurl to use IPv6 addresses only when resolving host names, and not for example try IPv4.
	Maximum number of parallel jobs in parallel mode. Default value

--jobs <NUM>	corresponds (in most cases) to the current amount of CPUs. See also --parallel. This is a cli-only option.
--json	Output each Hurl file result to JSON. The format is very closed to HAR format. This is a cli-only option.
--key <KEY>	Private key file name.
--limit-rate <SPEED>	Specify the maximum transfer rate you want Hurl to use, for both downloads and uploads. This feature is useful if you have a limited pipe and you would like your transfer not to use your entire bandwidth. To make it slower than it otherwise would be. The given speed is measured in bytes/second.
-L, --location	Follow redirect. To limit the amount of redirects to follow use the --max-redirs option
--location-trusted	Like -L, --location, but allows sending the name + password to all hosts that the site may redirect to. This may or may not introduce a security breach if the site redirects you to a site to which you send your authentication info (which is plaintext in the case of HTTP Basic authentication).
--max-filesize <BYTES>	Specify the maximum size in bytes of a file to download. If the file requested is larger than this value, the transfer does not start. This is a cli-only option.
--max-redirs <NUM>	Set maximum number of redirection-followings allowed By default, the limit is set to 50 redirections. Set this option to -1 to make it unlimited.
-m, --max-time <SECONDS>	Maximum time in seconds that you allow a request/response to take. This is the standard timeout. You can specify time units in the maximum time expression. Set Hurl to use a maximum time of 20 seconds with --max-time 20s or set it to 35,000 milliseconds with --max-time 35000ms. No spaces

	allowed. See also --connect-timeout. This is a cli-only option.
-n, --netrc	Scan the .netrc file in the user's home directory for the username and password. See also --netrc-file and --netrc-optional.
--netrc-file <FILE>	Like --netrc, but provide the path to the netrc file. See also --netrc-optional.
--netrc-optional	Similar to --netrc, but make the .netrc usage optional. See also --netrc-file.
--no-color	Do not colorize output. This is a cli-only option.
--no-output	Suppress output. By default, Hurl outputs the body of the last response. This is a cli-only option.
--noproxy <HOST(S)>	Comma-separated list of hosts which do not use a proxy. Override value from Environment variable no_proxy.
-o, --output <FILE>	Write output to FILE instead of stdout.
--parallel	Run files in parallel. Each Hurl file is executed in its own worker thread, without sharing anything with the other workers. The default run mode is sequential. Parallel execution is by default in --test mode. See also --jobs. This is a cli-only option.
--path-as-is	Tell Hurl to not handle sequences of /../ or /./ in the given URL path. Normally Hurl will squash or merge them according to standards but with this option set you tell it not to do that.
-x, --proxy <[PROTOCOL://]HOST[:PORT]>	Use the specified proxy.

--repeat <NUM>	Repeat the input files sequence NUM times, -1 for infinite loop. Given a.hurl, b.hurl, c.hurl as input, repeat two times will run a.hurl, b.hurl, c.hurl, a.hurl, b.hurl, c.hurl. This is a cli-only option.
--report-html <DIR>	Generate HTML report in DIR. If the HTML report already exists, it will be updated with the new test results. This is a cli-only option.
--report-json <DIR>	Generate JSON report in DIR. If the JSON report already exists, it will be updated with the new test results. This is a cli-only option.
--report-junit <FILE>	Generate JUnit File. If the FILE report already exists, it will be updated with the new test results. This is a cli-only option.
--report-tap <FILE>	Generate TAP report. If the FILE report already exists, it will be updated with the new test results. This is a cli-only option.
--resolve <HOST:PORT:ADDR>	Provide a custom address for a specific host and port pair. Using this, you can make the Hurl requests(s) use a specified address and prevent the otherwise normally resolved address to be used. Consider it a sort of /etc/hosts alternative provided on the command line.
--retry <NUM>	Maximum number of retries, 0 for no retries, -1 for unlimited retries. Retry happens if any error occurs (asserts, captures, runtimes etc...).
--retry-interval <MILLISECONDS>	Duration in milliseconds between each retry. Default is 1000 ms. You can specify time units in the retry interval expression. Set Hurl to use a retry interval of 2 seconds with --retry-interval 2s or set it to 500 milliseconds with --

	`retry-interval 500ms`. No spaces allowed.
<u>--secret <NAME=VALUE></u>	Define secret value to be redacted from logs and report. When defined, secrets can be used as variable everywhere variables are used.
<u>--ssl-no-revoke</u>	(Windows) This option tells Hurl to disable certificate revocation checks. WARNING: this option loosens the SSL security, and by using this flag you ask for exactly that. This is a cli-only option.
<u>--test</u>	Activate test mode: with this, the HTTP response is not outputted anymore, progress is reported for each Hurl file tested, and a text summary is displayed when all files have been run. In test mode, files are executed in parallel. To run test in a sequential way use `--job 1`. See also <u>--jobs</u>. This is a cli-only option.
<u>--to-entry <ENTRY_NUMBER></u>	Execute Hurl file to ENTRY_NUMBER (starting at 1). Ignore the remaining of the file. It is useful for debugging a session. This is a cli-only option.
<u>--unix-socket <PATH></u>	(HTTP) Connect through this Unix domain socket, instead of using the network.
<u>-u, --user <USER:PASSWORD></u>	Add basic Authentication header to each request.
<u>-A, --user-agent <NAME></u>	Specify the User-Agent string to send to the HTTP server. This is a cli-only option.
<u>--variable <NAME=VALUE></u>	Define variable (name/value) to be used in Hurl templates.
<u>--variables-file <FILE></u>	Set properties file in which your define your variables. Each variable is defined as name=value exactly as with <u>--variable</u> option. Note that defining a variable twice produces an error.

	This is a cli-only option.
-v, --verbose	Turn on verbose output on standard error stream. Useful for debugging. A line starting with '>' means data sent by Hurl. A line staring with '<' means data received by Hurl. A line starting with '*' means additional info provided by Hurl. If you only want HTTP headers in the output, -i, --include might be the option you're looking for.
--very-verbose	Turn on more verbose output on standard error stream. In contrast to --verbose option, this option outputs the full HTTP body request and response on standard error. In addition, lines starting with '**' are libcurl debug logs.
-h, --help	Usage help. This lists all current command line options with a short description.
-V, --version	Prints version information

Environment

Environment variables can only be specified in lowercase.

Using an environment variable to set the proxy has the same effect as using the -x, --proxy option.

Variable	Description
http_proxy [PROTOCOL://] <HOST>[:PORT]	Sets the proxy server to use for HTTP.
https_proxy [PROTOCOL://] <HOST>[:PORT]	Sets the proxy server to use for HTTPS.
all_proxy [PROTOCOL://] <HOST>[:PORT]	Sets the proxy server to use if no protocol-specific proxy is set.
no_proxy <comma-separated list of hosts>	List of host names that shouldn't go through any proxy.
HURL_name value	Define variable (name/value) to be used in Hurl templates. This

	is similar than `--variable` and `--variables-file` options.
`NO_COLOR`	When set to a non-empty string, do not colorize output (see `--no-color` option).

Exit Codes

Value	Description
0	Success.
1	Failed to parse command-line options.
2	Input File Parsing Error.
3	Runtime error (such as failure to connect to host).
4	Assert Error.

WWW

https://hurl.dev

See Also

curl(1) hurlfmt(1)

Samples

To run a sample, edit a file with the sample content, and run Hurl:

```
$ vi sample.hurl

GET https://example.org

$ hurl sample.hurl
```

By default, Hurl behaves like curl and outputs the last HTTP response's entry. To have a test oriented output, you can use --test option:

```
$ hurl --test sample.hurl
```

A particular response can be saved with [Options] section:

```
GET https://example.ord/cats/123
[Options]
output: cat123.txt    # use - to output to stdout
HTTP 200

GET https://example.ord/dogs/567
```

```
HTTP 200
```

Finally, Hurl can take files as input, or directories. In the latter case, Hurl will search files with `.hurl` extension recursively.

```
$ hurl --test integration/*.hurl
$ hurl --test .
```

You can check [Hurl tests suite](#) for more samples.

Getting Data

A simple GET:

```
GET https://example.org
```

Requests can be chained:

```
GET https://example.org/a
GET https://example.org/b
HEAD https://example.org/c
GET https://example.org/c
```

[Doc](#)

HTTP Headers

A simple GET with headers:

```
GET https://example.org/news
User-Agent: Mozilla/5.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
```

[Doc](#)

Query Params

```
GET https://example.org/news
[Query]
order: newest
search: something to search
count: 100
```

Or:

```
GET https://example.org/news?order=newest&search=something%20to%20search&count=10
```

> With `[Query]` section, params don't need to be URL escaped.

[Doc](#)

Basic Authentication

```
GET https://example.org/protected
[BasicAuth]
bob: secret
```

[Doc](#)

This is equivalent to construct the request with a [Authorization](#) header:

```
# Authorization header value can be computed with `echo -n 'bob:secret' | base64`
GET https://example.org/protected
Authorization: Basic Ym9iOnNlY3JldA==
```

Basic authentication section allows per request authentication. If you want to add basic authentication to all the requests of a Hurl file you could use [-u/--user option](#):

```
$ hurl --user bob:secret login.hurl
```

[--user](#) option can also be set per request:

```
GET https://example.org/login
[Options]
user: bob:secret
HTTP 200

GET https://example.org/login
[Options]
user: alice:secret
HTTP 200
```

Passing Data between Requests

[Captures](#) can be used to pass data from one request to another:

```
POST https://sample.org/orders
HTTP 201
[Captures]
order_id: jsonpath "$.order.id"

GET https://sample.org/orders/{{order_id}}
HTTP 200
```

[Doc](#)

Sending Data

Sending HTML Form Data

```
POST https://example.org/contact
[Form]
default: false
token: {{token}}
email: john.doe@rookie.org
number: 33611223344
```

[Doc](#)

Sending Multipart Form Data

```
POST https://example.org/upload
[Multipart]
field1: value1
field2: file,example.txt;
# One can specify the file content type:
field3: file,example.zip; application/zip
```

[Doc](#)

Multipart forms can also be sent with a [multiline string body](#):

```
POST https://example.org/upload
Content-Type: multipart/form-data; boundary="boundary"
```
--boundary
Content-Disposition: form-data; name="key1"

value1
--boundary
Content-Disposition: form-data; name="upload1"; filename="data.txt"
Content-Type: text/plain

Hello World!
--boundary
Content-Disposition: form-data; name="upload2"; filename="data.html"
Content-Type: text/html

<div>Hello World!</div>
--boundary--
```
```

In that case, files have to be inlined in the Hurl file.

[Doc](#)

Posting a JSON Body

With an inline JSON:

```
POST https://example.org/api/tests
{
    "id": "456",
    "evaluate": true
}
```

[Doc](#)

With a local file:

```
POST https://example.org/api/tests
Content-Type: application/json
file,data.json;
```

[Doc](#)

Templating a JSON Body

```
PUT https://example.org/api/hits
Content-Type: application/json
{
    "key0": "{{a_string}}",
    "key1": {{a_bool}},
    "key2": {{a_null}},
    "key3": {{a_number}}
}
```

Variables can be initialized via command line:

```
$ hurl --variable a_string=apple \
       --variable a_bool=true \
       --variable a_null=null \
       --variable a_number=42 \
       test.hurl
```

Resulting in a PUT request with the following JSON body:

```
{
    "key0": "apple",
    "key1": true,
    "key2": null,
    "key3": 42
}
```

[Doc](#)

Templating a XML Body

Using templates with [XML body](#) is not currently supported in Hurl. You can use templates in [XML multiline string body](#) with variables to send a variable XML body:

```
POST https://example.org/echo/post/xml
```xml
<?xml version="1.0" encoding="utf-8"?>
<Request>
    <Login>{{login}}</Login>
    <Password>{{password}}</Password>
</Request>
```
```

[Doc](#)

Using GraphQL Query

A simple GraphQL query:

```
POST https://example.org/starwars/graphql
```graphql
```

```
  {
    human(id: "1000") {
      name
      height(unit: FOOT)
    }
  }
  ```
```

A GraphQL query with variables:

```
  POST https://example.org/starwars/graphql
  ```graphql
  query Hero($episode: Episode, $withFriends: Boolean!) {
    hero(episode: $episode) {
      name
      friends @include(if: $withFriends) {
        name
      }
    }
  }

  variable {
    "episode": "JEDI",
    "withFriends": false
  }
  ```
```

GraphQL queries can also use [Hurl templates](#).

[Doc](#)

Using Dynamic Datas

[Functions](#) like `newUuid` and `newDate` can be used in templates to create dynamic datas:

A file that creates a dynamic email (i.e `0531f78f-7f87-44be-a7f2-969a1c4e6d97@test.com`):

```
  POST https://example.org/api/foo
  {
    "name": "foo",
    "email": "{{newUuid}}@test.com"
  }
```

A file that creates a dynamic query parameter (i.e `2024-12-02T10:35:44.461731Z`):

```
  GET https://example.org/api/foo
  [Query]
  date: {{newDate}}
  HTTP 200
```

[Doc](#)

Testing Response

Responses are optional, everything after `HTTP` is part of the response asserts.

```
  # A request with (almost) no check:
  GET https://foo.com
```

```
# A status code check:
GET https://foo.com
HTTP 200

# A test on response body
GET https://foo.com
HTTP 200
[Asserts]
jsonpath "$.state" == "running"
```

Testing Status Code

```
GET https://example.org/order/435
HTTP 200
```

Doc

```
GET https://example.org/order/435
# Testing status code is in a 200-300 range
HTTP *
[Asserts]
status >= 200
status < 300
```

Doc

Testing Response Headers

Use implicit response asserts to test header values:

```
GET https://example.org/index.html
HTTP 200
Set-Cookie: theme=light
Set-Cookie: sessionToken=abc123; Expires=Wed, 09 Jun 2021 10:18:14 GMT
```

Doc

Or use explicit response asserts with predicates:

```
GET https://example.org
HTTP 302
[Asserts]
header "Location" contains "www.example.net"
```

Doc

Implicit and explicit asserts can be combined:

```
GET https://example.org/index.html
HTTP 200
Set-Cookie: theme=light
Set-Cookie: sessionToken=abc123; Expires=Wed, 09 Jun 2021 10:18:14 GMT
[Asserts]
header "Location" contains "www.example.net"
```

Testing REST APIs

Asserting JSON body response (node values, collection count etc...) with [JSONPath](#):

```
GET https://example.org/order
screencapability: low
HTTP 200
[Asserts]
jsonpath "$.validated" == true
jsonpath "$.userInfo.firstName" == "Franck"
jsonpath "$.userInfo.lastName" == "Herbert"
jsonpath "$.hasDevice" == false
jsonpath "$.links" count == 12
jsonpath "$.state" != null
jsonpath "$.order" matches "^order-\\d{8}$"
jsonpath "$.order" matches /^order-\d{8}$/     # Alternative syntax with regex l:
jsonpath "$.created" isIsoDate
```

[Doc](#)

Testing HTML Response

```
GET https://example.org
HTTP 200
Content-Type: text/html; charset=UTF-8
[Asserts]
xpath "string(/html/head/title)" contains "Example" # Check title
xpath "count(//p)" == 2  # Check the number of p
xpath "//p" count == 2  # Similar assert for p
xpath "boolean(count(//h2))" == false  # Check there is no h2
xpath "//h2" not exists  # Similar assert for h2
xpath "string(//div[1])" matches /Hello.*/
```

[Doc](#)

Testing Set-Cookie Attributes

```
GET https://example.org/home
HTTP 200
[Asserts]
cookie "JSESSIONID" == "8400BAFE2F66443613DC38AE3D9D6239"
cookie "JSESSIONID[Value]" == "8400BAFE2F66443613DC38AE3D9D6239"
cookie "JSESSIONID[Expires]" contains "Wed, 13 Jan 2021"
cookie "JSESSIONID[Secure]" exists
cookie "JSESSIONID[HttpOnly]" exists
cookie "JSESSIONID[SameSite]" == "Lax"
```

[Doc](#)

Testing Bytes Content

Check the SHA-256 response body hash:

```
GET https://example.org/data.tar.gz
HTTP 200
[Asserts]
sha256 == hex,039058c6f2c0cb492c533b0a4d14ef77cc0f78abccced5287d84a1a2011cfb81;
```

[Doc](#)

SSL Certificate

Check the properties of a SSL certificate:

```
GET https://example.org
HTTP 200
[Asserts]
certificate "Subject" == "CN=example.org"
certificate "Issuer" == "C=US, O=Let's Encrypt, CN=R3"
certificate "Expire-Date" daysAfterNow > 15
certificate "Serial-Number" matches /[\da-f]+/
```

Doc

Checking Full Body

Use implicit body to test an exact JSON body match:

```
GET https://example.org/api/cats/123
HTTP 200
{
  "name" : "Purrsloud",
  "species" : "Cat",
  "favFoods" : ["wet food", "dry food", "<strong>any</strong> food"],
  "birthYear" : 2016,
  "photo" : "https://learnwebcode.github.io/json-example/images/cat-2.jpg"
}
```

Doc

Or an explicit assert file:

```
GET https://example.org/index.html
HTTP 200
[Asserts]
body == file,cat.json;
```

Doc

Implicit asserts supports XML body:

```
GET https://example.org/api/catalog
HTTP 200
<?xml version="1.0" encoding="UTF-8"?>
<catalog>
    <book id="bk101">
        <author>Gambardella, Matthew</author>
        <title>XML Developer's Guide</title>
        <genre>Computer</genre>
        <price>44.95</price>
        <publish_date>2000-10-01</publish_date>
        <description>An in-depth look at creating applications with XML.</descript:
    </book>
</catalog>
```

Doc

Plain text:

```
GET https://example.org/models
HTTP 200
```
Year,Make,Model,Description,Price
```

```
1997,Ford,E350,"ac, abs, moon",3000.00
1999,Chevy,"Venture ""Extended Edition""","",4900.00
1999,Chevy,"Venture ""Extended Edition, Very Large""",,5000.00
1996,Jeep,Grand Cherokee,"MUST SELL! air, moon roof, loaded",4799.00
```
```

[Doc](#)

One line:

```
POST https://example.org/helloworld
HTTP 200
`Hello world!`
```

[Doc](#)

File:

```
GET https://example.org
HTTP 200
file,data.bin;
```

[Doc](#)

Reports

HTML Report

```
$ hurl --test --report-html build/report/ *.hurl
```

[Doc](#)

JSON Report

```
$ hurl --test --report-json build/report/ *.hurl
```

[Doc](#)

JUnit Report

```
$ hurl --test --report-junit build/report.xml *.hurl
```

[Doc](#)

TAP Report

```
$ hurl --test --report-tap build/report.txt *.hurl
```

[Doc](#)

JSON Output

A structured output of running Hurl files can be obtained with [--json option](#). Each file will produce a JSON export of the run.

```
$ hurl --json *.hurl
```

Others

HTTP Version

Testing HTTP version (HTTP/1.0, HTTP/1.1, HTTP/2 or HTTP/3) can be done using implicit asserts:

```
GET https://foo.com
HTTP/3 200

GET https://bar.com
HTTP/2 200
```

[Doc](#)

Or explicit:

```
GET https://foo.com
HTTP 200
[Asserts]
version == "3"

GET https://bar.com
HTTP 200
[Asserts]
version == "2"
version toFloat > 1.1
```

[Doc](#)

IP Address

Testing the IP address of the response, as a string. This string may be IPv6 address:

```
GET https://foo.com
HTTP 200
[Asserts]
ip == " 2001:0db8:85a3:0000:0000:8a2e:0370:733"
ip startsWith "2001"
ip isIpv6
```

Polling and Retry

Retry request on any errors (asserts, captures, status code, runtime etc...):

```
# Create a new job
POST https://api.example.org/jobs
HTTP 201
[Captures]
job_id: jsonpath "$.id"
[Asserts]
```

```
jsonpath "$.state" == "RUNNING"


# Pull job status until it is completed
GET https://api.example.org/jobs/{{job_id}}
[Options]
retry: 10   # maximum number of retry, -1 for unlimited
retry-interval: 500ms
HTTP 200
[Asserts]
jsonpath "$.state" == "COMPLETED"
```

Doc

Delaying Requests

Add delay for every request, or a particular request:

```
# Delaying this request by 5 seconds (aka sleep)
GET https://example.org/turtle
[Options]
delay: 5s
HTTP 200

# No delay!
GET https://example.org/turtle
HTTP 200
```

Doc

Skipping Requests

```
# a, c, d are run, b is skipped
GET https://example.org/a

GET https://example.org/b
[Options]
skip: true

GET https://example.org/c

GET https://example.org/d
```

Doc

Testing Endpoint Performance

```
GET https://sample.org/helloworld
HTTP *
[Asserts]
duration < 1000   # Check that response time is less than one second
```

Doc

Using SOAP APIs

```
POST https://example.org/InStock
Content-Type: application/soap+xml; charset=utf-8
SOAPAction: "http://www.w3.org/2003/05/soap-envelope"
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope" xmlns:m="http
```

```
    <soap:Header></soap:Header>
    <soap:Body>
      <m:GetStockPrice>
        <m:StockName>GOOG</m:StockName>
      </m:GetStockPrice>
    </soap:Body>
  </soap:Envelope>
HTTP 200
```

Doc

Capturing and Using a CSRF Token

```
GET https://example.org
HTTP 200
[Captures]
csrf_token: xpath "string(//meta[@name='_csrf_token']/@content)"


POST https://example.org/login?user=toto&password=1234
X-CSRF-TOKEN: {{csrf_token}}
HTTP 302
```

Doc

Redacting Secrets

Using command-line for known values:

```
$ hurl --secret token=1234 file.hurl
```

```
POST https://example.org
X-Token: {{token}}
{
  "name": "Alice",
  "value": 100
}
HTTP 200
```

Doc

Using redact for dynamic values:

```
# Get an authorization token:
GET https://example.org/token
HTTP 200
[Captures]
token: header "X-Token" redact

# Send an authorized request:
POST https://example.org
X-Token: {{token}}
{
  "name": "Alice",
  "value": 100
}
HTTP 200
```

Doc

Checking Byte Order Mark (BOM) in Response Body

```
GET https://example.org/data.bin
HTTP 200
[Asserts]
bytes startsWith hex,efbbbf;
```

Doc

AWS Signature Version 4 Requests

Generate signed API requests with AWS Signature Version 4, as used by several cloud providers.

```
POST https://sts.eu-central-1.amazonaws.com/
[Options]
aws-sigv4: aws:amz:eu-central-1:sts
[Form]
Action: GetCallerIdentity
Version: 2011-06-15
```

The Access Key is given per --user, either with command line option or within the [Options] section:

```
POST https://sts.eu-central-1.amazonaws.com/
[Options]
aws-sigv4: aws:amz:eu-central-1:sts
user: bob=secret
[Form]
Action: GetCallerIdentity
Version: 2011-06-15
```

Doc

Using curl Options

curl options (for instance --resolve or --connect-to) can be used as CLI argument. In this case, they're applicable to each request of an Hurl file.

```
$ hurl --resolve foo.com:8000:127.0.0.1 foo.hurl
```

Use [Options] section to configure a specific request:

```
GET http://bar.com
HTTP 200


GET http://foo.com:8000/resolve
[Options]
resolve: foo.com:8000:127.0.0.1
HTTP 200
`Hello World!`
```

Doc

Running Tests

Use --test Option

Hurl is run by default as an HTTP client, returning the body of the last HTTP response.

```
$ hurl hello.hurl
Hello World!
```

When multiple input files are provided, Hurl outputs the body of the last HTTP response of each file.

```
$ hurl hello.hurl assert_json.hurl
Hello World![
  { "id": 1, "name": "Bob"},
  { "id": 2, "name": "Bill"}
]
```

For testing, we are only interested in the asserts results, we don't need the HTTP body response. To use Hurl as a test tool with an adapted output, you can use --test option:

```
$ hurl --test hello.hurl assert_json.hurl
hello.hurl: Success (6 request(s) in 245 ms)
assert_json.hurl: Success (8 request(s) in 308 ms)
--------------------------------------------------------------------------------
Executed files:    2
Executed requests: 10 (17.82/s)
Succeeded files:   2 (100.0%)
Failed files:      0 (0.0%)
Duration:          561 ms
```

Or, in case of errors:

```
$ hurl --test hello.hurl error_assert_status.hurl
hello.hurl: Success (4 request(s) in 5 ms)
error: Assert status code
  --> error_assert_status.hurl:9:6
   |
   | GET http://localhost:8000/not_found
   | ...
 9 | HTTP 200
   |      ^^^ actual value is <404>
   |

error_assert_status.hurl: Failure (1 request(s) in 2 ms)
--------------------------------------------------------------------------------
Executed files:    2
Executed requests: 5 (500.0/s)
Succeeded files:   1 (50.0%)
Failed files:      1 (50.0%)
Duration:          10 ms
```

In test mode, files are executed in parallel to speed-ud the execution. If a sequential run is needed, you can use --jobs 1 option to execute tests one by one.

```
$ hurl --test --jobs 1 *.hurl
```

--repeat option can be used to repeat run files and do performance check. For instance, this call will run 1000 tests in parallel:

```
$ hurl --test --repeat 1000 stress.hurl
```

Selecting Tests

Hurl can take multiple files into inputs:

```
$ hurl --test test/integration/a.hurl test/integration/b.hurl test/integration/c.
```

```
$ hurl --test test/integration/*.hurl
```

Or you can simply give a directory and Hurl will find files with `.hurl` extension recursively:

```
$ hurl --test test/integration/
```

Finally, you can use [--glob option](#) to test files that match a given pattern:

```
$ hurl --test --glob "test/integration/**/*.hurl"
```

Debugging

Debug Logs

If you need more error context, you can use [--error-format long option](#) to print HTTP bodies for failed asserts:

```
$ hurl --test --error-format long hello.hurl error_assert_status.hurl
hello.hurl: Success (4 request(s) in 6 ms)
HTTP/1.1 404
Server: Werkzeug/3.0.3 Python/3.12.4
Date: Wed, 10 Jul 2024 15:42:41 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 207
Server: Flask Server
Connection: close

<!doctype html>
<html lang=en>
<title>404 Not Found</title>
<h1>Not Found</h1>
<p>The requested URL was not found on the server. If you entered the URL manually

error: Assert status code
  --> error_assert_status.hurl:9:6
   |
   | GET http://localhost:8000/not_found
   | ...
 9 | HTTP 200
   |      ^^^ actual value is <404>
   |

error_assert_status.hurl: Failure (1 request(s) in 2 ms)
--------------------------------------------------------------------------------
Executed files:   2
Executed requests: 5 (454.5/s)
Succeeded files:   1 (50.0%)
```

```
Failed files:      1 (50.0%)
Duration:          11 ms
```

Individual requests can be modified with [[Options] section]options to turn on logs for a particular request, using verbose and very-verbose option.

With this Hurl file:

```
GET https://foo.com
HTTP 200

GET https://bar.com
[Options]
very-verbose: true
HTTP 200

GET https://baz.com
HTTP 200
```

Running hurl --test . will output debug logs for the request to https://bar.com.

--verbose / --very-verbose can also be enabled globally, for every requests of every tested files:

```
$ hurl --test --very-verbose .
```

HTTP Responses

In test mode, HTTP responses are not displayed. One way to get HTTP responses even in test mode is to use --output option of [Options] section: --output file allows to save a particular response to a file, while --output - allows to redirect HTTP responses to standard output.

```
GET http://foo.com
HTTP 200

GET https://bar.com
[Options]
output: -
HTTP 200
```

```
$ hurl --test .
<html><head><meta http-equiv="content-type" content="text/html;charset=utf-8">
<title>301 Moved</TITLE></head><body>
<h1>301 Moved</h1>
The document has moved
<a HREF="https://www.bar.com/">here</a>.
</body></html>
/tmp/test.hurl: Success (2 request(s) in 184 ms)
--------------------------------------------------------------------------------
Executed files:    1
Executed requests: 2 (10.7/s)
Succeeded files:   1 (100.0%)
Failed files:      0 (0.0%)
Duration:          187 ms
```

Generating Report

In the different reports, files are always referenced in the input order (which, as tests are executed in

parallel, can be different from the execution order).

HTML Report

Hurl can generate an HTML report by using the `--report-html DIR` option.

If the HTML report already exists, the test results will be appended to it.

Test Report

Tue, 27 Dec 2022 12:36:52 +0100

Executed: 3 (100%)
Succeeded: 2 (66.7%)
Failed: 1 (33.3%)

File	Status	Duration
Users/jc/Documents/Dev/hurl/integration/tests_ok/assert_base64.hurl	success	0.006
Users/jc/Documents/Dev/hurl/integration/tests_ok/assert_header.hurl	success	0.007
Users/jc/Documents/Dev/hurl/integration/tests_ok/assert_json.hurl	failure	0.01

The input Hurl files (HTML version) are also included and are easily accessed from the main page.

```
1   GET http://localhost:8000/assert-json
2   HTTP/1.0 200
3   [Asserts]
4   jsonpath "$.count" equals 5
5   jsonpath "$.count" == 5
6   jsonpath "$.count" equals 5.0
7   jsonpath "$.count" notEquals 4
8   jsonpath "$.count" != 4
9   jsonpath "$.count" greaterThan 1
10  jsonpath "$.count" greaterThan 1.0
11  jsonpath "$.count" >= 1.0
```

JSON Report

A JSON report can be produced by using the `--report-json DIR`. The report directory will contain a `report.json` file, including each test file executed with `--json` option and a reference to each HTTP response of the run dumped to disk.

If the JSON report already exists, it will be updated with the new test results.

JUnit Report

A JUnit report can be produced by using the `--report-junit FILE` option.

If the JUnit report already exists, it will be updated with the new test results.

TAP Report

A TAP report (Test Anything Protocol) can be produced by using the `--report-tap FILE` option.

If the TAP report already exists, it will be updated with the new test results.

Use Variables in Tests

To use variables in your tests, you can:

- use `--variable option`

- use [--variables-file option](#)
- define environment variables, for instance HURL_foo=bar

You will find a detailed description in the [Injecting Variables](#) section of the docs.

Frequently Asked Questions

General

Why "Hurl"?

The name Hurl is a tribute to the awesome [curl](#), with a focus on the HTTP protocol. While it may have an informal meaning not particularly elegant, [other eminent tools](#) have set a precedent in naming.

Yet Another Tool, I already use X

We think that Hurl has some advantages compared to similar tools.

Hurl is foremost a command line tool and should be easy to use on a local computer, or in a CI/CD pipeline. Some tools in the same space as Hurl ([Postman](#) for instance), are GUI oriented, and we find it less attractive than CLI. As a command line tool, Hurl can be used to get HTTP data (like [curl](#)), but also as a test tool for HTTP sessions, or even as documentation.

Having a text based [file format](#) is another advantage. The Hurl format is simple, focused on the HTTP domain, can serve as documentation and can be read or written by non-technical people.

For instance putting JSON data with Hurl can be done with this simple file:

```
PUT http://localhost:3000/api/login
{
  "username": "xyz",
  "password": "xyz"
}
```

With [curl](#):

```
curl --header "Content-Type: application/json" \
     --request PUT \
     --data '{"username": "xyz","password": "xyz"}' \
     http://localhost:3000/api/login
```

[Karate](#), a tool combining API test automation, mocking, performance-testing, has similar features but offers also much more at a cost of an increased complexity.

Comparing Karate file format:

```
Scenario: create and retrieve a cat

Given url 'http://myhost.com/v1/cats'
And request { name: 'Billie' }
When method post
Then status 201
And match response == { id: '#notnull', name: 'Billie' }

Given path response.id
When method get
```

```
    Then status 200
```

And Hurl:

```
# Scenario: create and retrieve a cat

POST http://myhost.com/v1/cats
{ "name": "Billie" }
HTTP 201
[Captures]
cat_id: jsonpath "$.id"
[Asserts]
jsonpath "$.name" == "Billie"

GET http://myshost.com/v1/cats/{{cat_id}}
HTTP 200
```

A key point of Hurl is to work on the HTTP domain. In particular, there is no JavaScript runtime, Hurl works on the raw HTTP requests/responses, and not on a DOM managed by a HTML engine. For security, this can be seen as a feature: let's say you want to test backend validation, you want to be able to bypass the browser or javascript validations and directly test a backend endpoint.

Finally, with no headless browser and working on the raw HTTP data, Hurl is also really reliable with a very small probability of false positives. Integration tests with tools like Selenium can, in this regard, be challenging to maintain.

Just use what is convenient for you. In our case, it's Hurl!

Hurl is build on top of libcurl, but what is added?

Hurl has two main functionalities on top of curl:

1. Chain several requests:

   With its captures, it enables to inject data received from a response into following requests. CSRF tokens are typical examples in a standard web session.

2. Test HTTP responses:

   With its asserts, responses can be easily tested.

Hurl benefits from the features of the `libcurl` against it is linked. You can check `libcurl` version with `hurl --version`.

For instance on macOS:

```
$ hurl --version
hurl 2.0.0 libcurl/7.79.1 (SecureTransport) LibreSSL/3.3.6 zlib/1.2.11 nghttp2/1.
Features (libcurl):  alt-svc AsynchDNS HSTS HTTP2 IPv6 Largefile libz NTLM NTLM_W
Features (built-in): brotli
```

You can also check which `libcurl` is used.

On macOS:

```
$ which hurl
/opt/homebrew/bin/hurl
$ otool -L /opt/homebrew/bin/hurl:
        /usr/lib/libxml2.2.dylib (compatibility version 10.0.0, current version
        /System/Library/Frameworks/CoreFoundation.framework/Versions/A/CoreFounda
```

```
        /usr/lib/libcurl.4.dylib (compatibility version 7.0.0, current version 9.
        /usr/lib/libiconv.2.dylib (compatibility version 7.0.0, current version
        /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version
```

On Linux:

```
$ which hurl
/root/.cargo/bin/hurl
$ ldd /root/.cargo/bin/hurl
ldd /root/.cargo/bin/hurl
        linux-vdso.so.1 (0x0000ffff8656a000)
        libxml2.so.2 => /usr/lib/aarch64-linux-gnu/libxml2.so.2 (0x0000ffff85fe8(
        libcurl.so.4 => /usr/lib/aarch64-linux-gnu/libcurl.so.4 (0x0000ffff85f45(
        libgcc_s.so.1 => /lib/aarch64-linux-gnu/libgcc_s.so.1 (0x0000ffff85f2100(
        ...
        libkeyutils.so.1 => /lib/aarch64-linux-gnu/libkeyutils.so.1 (0x0000ffff82
        libffi.so.7 => /usr/lib/aarch64-linux-gnu/libffi.so.7 (0x0000ffff82ebc00(
```

Note that some Hurl features are dependent on `libcurl` capacities: for instance, if your `libcurl` doesn't support HTTP/2 Hurl won't be able to send HTTP/2 request.

Why shouldn't I use Hurl?

If you need a GUI. Currently, Hurl does not offer a GUI version (like [Postman](#)). While we think that it can be useful, we prefer to focus for the time-being on the core, keeping something simple and fast. Contributions to build a GUI are welcome.

I have a large numbers of tests, how to run just specific tests?

By convention, you can organize Hurl files into different folders or prefix them.

For example, you can split your tests into two folders critical and additional.

```
critical/test1.hurl
critical/test2.hurl
additional/test1.hurl
additional/test2.hurl
```

You can simply run your critical tests with

```
$ hurl --test critical/*.hurl
```

How can I use my Hurl files outside Hurl?

Hurl file can be exported to a JSON file with `hurlfmt`. This JSON file can then be easily parsed for converting a different format, getting ad-hoc information,...

For example, the Hurl file

```
GET https://example.org/api/users/1
User-Agent: Custom
HTTP 200
[Asserts]
jsonpath "$.name" == "Bob"
```

will be converted to JSON with the following command:

```
$ hurlfmt test.hurl --out json | jq
{
  "entries": [
    {
      "request": {
        "method": "GET",
        "url": "https://example.org/api/users/1",
        "headers": [
          {
            "name": "User-Agent",
            "value": "Custom"
          }
        ]
      },
      "response": {
        "version": "HTTP",
        "status": 200,
        "asserts": [
          {
            "query": {
              "type": "jsonpath",
              "expr": "$.name"
            },
            "predicate": {
              "type": "==",
              "value": "Bob"
            }
          }
        ]
      }
    }
  ]
}
```

Can I do calculation within a Hurl file?

Currently, the templating is very simple, only accessing variables. Calculations can be done beforehand, before running the Hurl File.

For example, with date calculations, variables `now` and `tomorrow` can be used as param or expected value.

```
$ TODAY=$(date '+%y%m%d')
$ TOMORROW=$(date '+%y%m%d' -d"+1days")
$ hurl --variable "today=$TODAY" --variable "tomorrow=$TOMORROW" test.hurl
```

You can also use environment variables that begins with HURL_ to inject data in an Hurl file. For instance, to inject `today` and `tomorrow` variables:

```
$ export HURL_today=$(date '+%y%m%d')
$ export HURL_tomorrow=$(date '+%y%m%d' -d"+1days")
$ hurl test.hurl
```

You can also use [filters](#) to process HTTP responses in asserts and captures.

macOS

How can I use a custom libcurl (from Homebrew by instance)?

No matter how you've installed Hurl (using the precompiled binary for macOS or with [Homebrew](#)) Hurl is linked against the built-in system libcurl. If you want to use another libcurl (for instance, if you've installed curl with Homebrew and want Hurl to use Homebrew's libcurl), you can patch Hurl

with the following command:

```
$ sudo install_name_tool -change /usr/lib/libcurl.4.dylib PATH_TO_CUSTOM_LIBCURL
```

For instance:

```
# /usr/local/opt/curl/lib/libcurl.4.dylib is installed by `brew install curl`
$ sudo install_name_tool -change /usr/lib/libcurl.4.dylib /usr/local/opt/curl/lik
```

File Format

Hurl File

Character Encoding

Hurl file should be encoded in UTF-8, without a byte order mark at the beginning (while Hurl ignores the presence of a byte order mark rather than treating it as an error)

File Extension

Hurl file extension is `.hurl`

Comments

Comments begin with # and continue until the end of line. Hurl file can serve as a documentation for HTTP based workflows so it can be useful to be very descriptive.

```
# A very simple Hurl file
# with tasty comments...
GET https://www.sample.net
x-app: MY_APP  # Add a dummy header
HTTP 302       # Check that we have a redirection
[Asserts]
header "Location" exists
header "Location" contains "login"  # Check that we are redirected to the login
```

Special Characters in Strings

String can include the following special characters:

- The escaped special characters " (double quotation mark), \ (backslash), \b (backspace), \f (form feed), \n (line feed), \r (carriage return), and \t (horizontal tab)
- An arbitrary Unicode scalar value, written as \u{n}, where n is a 1–8 digit hexadecimal number

```
GET https://example.org/api
HTTP 200
# The following assert are equivalent:
[Asserts]
jsonpath "$.slideshow.title" == "A beautiful ✈!"
jsonpath "$.slideshow.title" == "A beautiful \u{2708}!"
```

In some case, (in headers value, etc..), you will also need to escape # to distinguish it from a comment. In the following example:

```
GET https://example.org/api
x-token: BEEF \#STEACK # Some comment
HTTP 200
```

We're sending a header `x-token` with value BEEF #STEACK

Entry

Definition

A Hurl file is a list of entries, each entry being a mandatory request, optionally followed by a response.

Responses are not mandatory, a Hurl file consisting only of requests is perfectly valid. To sum up, responses can be used to capture values to perform subsequent requests, or add asserts to HTTP responses.

Example

```
# First, test home title.
GET https://acmecorp.net
HTTP 200
[Asserts]
xpath "normalize-space(//head/title)" == "Hello world!"

# Get some news, response description is optional
GET https://acmecorp.net/news

# Do a POST request without CSRF token and check
# that status code is Forbidden 403
POST https://acmecorp.net/contact
[Form]
default: false
email: john.doe@rookie.org
number: 33611223344
HTTP 403
```

Description

Options

Options specified on the command line apply to every entry in an Hurl file. For instance, with --location option, every entry of a given file will follow redirection:

```
$ hurl --location foo.hurl
```

You can use an [[Options] section]options to set option only for a specified request. For instance, in this Hurl file, the second entry will follow location (so we can test the status code to be 200 instead of 301).

```
GET https://google.fr
HTTP 301

GET https://google.fr
[Options]
location: true
HTTP 200

GET https://google.fr
HTTP 301
```

You can use the [Options](#getting-started-manual-options) section to log a specific entry:

```
# ... previous entries

GET https://api.example.org
[Options]
very-verbose: true
HTTP 200



# ... next entries
```

Cookie storage

Requests in the same Hurl file share the cookie storage, enabling, for example, session based scenario.

Redirects

By default, Hurl doesn't follow redirection. To effectively run a redirection, entries should describe each step of the redirection, allowing insertion of asserts in each response.

```
# First entry, test the redirection (status code and 'Location' header)
GET https://google.fr
HTTP 301
Location: https://www.google.fr/

# Second entry, the 200 OK response
GET https://www.google.fr
HTTP 200
```

Alternatively, one can use --location / --location-trusted options to force redirection to be followed. In this case, asserts are executed on the last received response. Optionally, the number of redirections can be limited with --max-redirs.

```
# Running hurl --location google.hurl
GET https://google.fr
HTTP 200
```

Finally, you can force redirection on a particular request with an [[Options] section]options and the --location / --location-trusted options:

```
GET https://google.fr
[Options]
location-trusted: true
HTTP 200
```

Retry

Every entry can be retried upon asserts, captures or runtime errors. Retries allow polling scenarios and effective runs under flaky conditions. Asserts can be explicit (with an [[Asserts] section]asserts), or implicit (like headers or status code).

Retries can be set globally for every request (see --retry and --retry-interval), or activated on a particular request with an [[Options] section]options.

For example, in this Hurl file, first we create a new job then we poll the new job until it's completed:

```
# Create a new job
POST http://api.example.org/jobs
```

```
HTTP 201
[Captures]
job_id: jsonpath "$.id"
[Asserts]
jsonpath "$.state" == "RUNNING"


# Pull job status until it is completed
GET http://api.example.org/jobs/{{job_id}}
[Options]
retry: 10    # maximum number of retry, -1 for unlimited
retry-interval: 300ms
HTTP 200
[Asserts]
jsonpath "$.state" == "COMPLETED"
```

Control flow

In [Options](#getting-started-manual-options) section, skip and repeat can be used to control flow of execution:

- skip: true/false skip this request and execute the next one unconditionally,
- repeat: N loop the request N times. If there are assert or runtime errors, the requests execution is stopped.

```
# This request will be played exactly 3 times
GET https://example.org/foo
[Options]
repeat: 3
HTTP 200

# This request is skipped
GET https://example.org/foo
[Options]
skip: true
HTTP 200
```

Additionally, a delay can be inserted between requests, to add a delay before execution of a request (aka sleep).

```
# A 5 seconds delayed request
GET https://example.org/foo
[Options]
delay: 5s
HTTP 200
```

delay and repeat can also be used globally as command line options:

```
$ hurl --delay 500ms --repeat 3 foo.hurl
```

For complete reference, below is a diagram for the executed entries.

Request

Definition

Request describes an HTTP request: a mandatory [method](#) and [URL](#), followed by optional [headers](#).

Then, [options](#), [query parameters](#), [form parameters](#), [multipart form data](#), [cookies](#), and [basic authentication](#) can be used to configure the HTTP request.

Finally, an optional [body](#) can be used to configure the HTTP request body.

Example

```
GET https://example.org/api/dogs?id=4567
User-Agent: My User Agent
Content-Type: application/json
[BasicAuth]
alice: secret
```

Structure

```
PUT https://sample.net        Method and URL (mandatory)
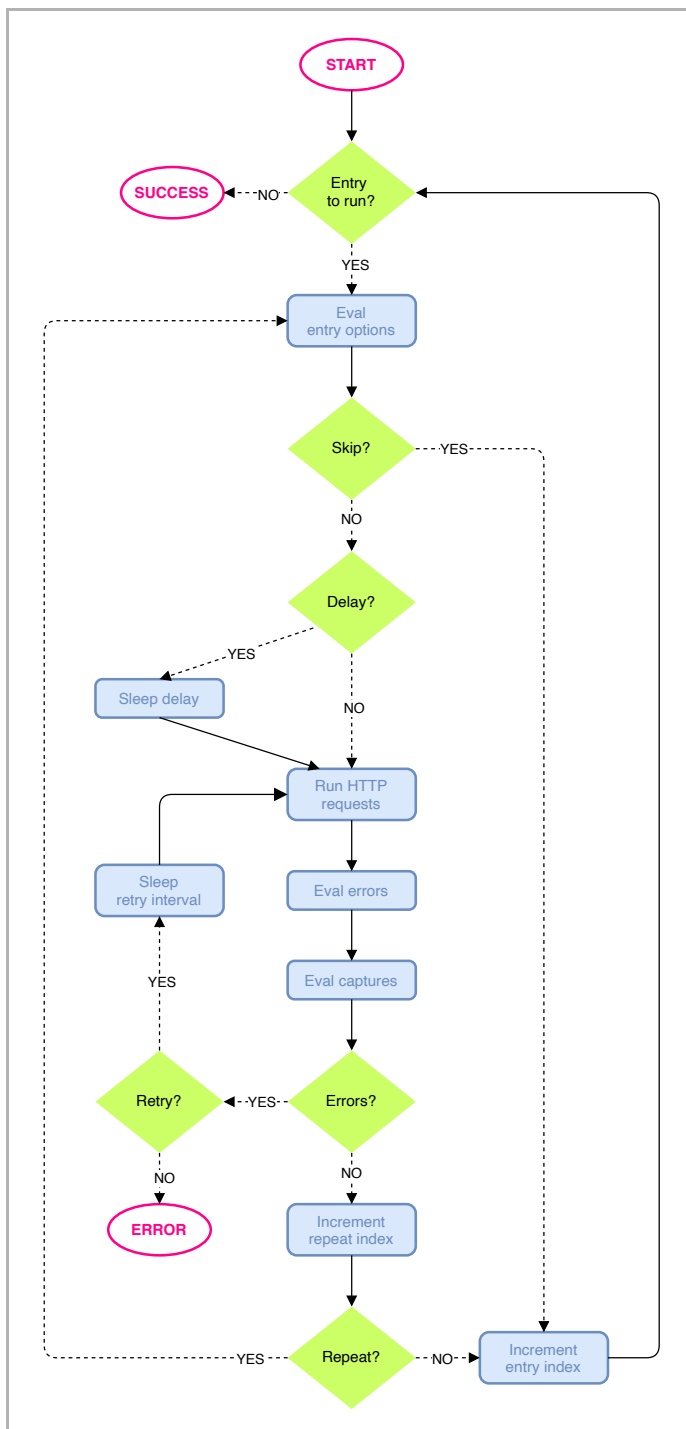
accept: */*
x-powered-by: Express         HTTP request headers (optional)
user-agent: Test
[Options]
...
[Query]
...
[Form]                        Options, query strings, form params, cookies, authentication ...
...                           (optional sections, unordered)
[BasicAuth]
...
[Cookies]
...
...
...                           HTTP request body (optional)
{
  "type": "FOO",
  "value": 356789,
  "ordered": true,
  "index": 10
}
```

Headers, if present, follow directly after the method and URL. This allows Hurl format to 'look like'
the real HTTP format. Contrary to HTTP headers, other parameters are defined in sections
([Cookies], [Query], [Form] etc...) These sections are not ordered and can be mixed in any way:

```
GET https://example.org/api/dogs
User-Agent: My User Agent
[Query]
id: 4567
order: newest
[BasicAuth]
alice: secret
```

```
GET https://example.org/api/dogs
User-Agent: My User Agent
[BasicAuth]
alice: secret
[Query]
id: 4567
order: newest
```

The last optional part of a request configuration is the request body. Request body must be the last
parameter of a request (after headers and request sections). Like headers, body have no explicit
marker:
```

```
POST https://example.org/api/dogs?id=4567
User-Agent: My User Agent
{
 "name": "Ralphy"
}
```

Description

Method

Mandatory HTTP request method, usually one of GET, HEAD, POST, PUT, DELETE, CONNECT, OPTIONS, TRACE and PATCH.

Other methods can be used like QUERY with the constraint of using only uppercase chars.

URL

Mandatory HTTP request URL.

URL can contain query parameters, even if using a [query parameters section](#) is preferred.

```
# A request with URL containing query parameters.
GET https://example.org/forum/questions/?search=Install%20Linux&order=newest

# A request with query parameters section, equivalent to the first request.
GET https://example.org/forum/questions/
[Query]
search: Install Linux
order: newest
```

Query parameters in query parameter section are not URL encoded.

When query parameters are present in the URL and in a query parameters section, the resulting request will have both parameters.

Headers

Optional list of HTTP request headers.

A header consists of a name, followed by a : and a value.

```
GET https://example.org/news
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:70.0) Gecko/2010010
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
```

Headers directly follow URL, without any section name, contrary to query parameters, form parameters or cookies

Note that a header usually doesn't start with double quotes. If a header value starts with double quotes, double quotes will be part of the header value:

```
PATCH https://example.org/file.txt
```

```
If-Match: "e0023aa4e"
```

`If-Match` request header will be sent will the following value "e0023aa4e" (started and ended with double quotes).

Headers must follow directly after the method and [URL](URL).

Options

Options used to execute this request.

Options such as --location, --verbose, --insecure can be used at the command line and applied to every request of an Hurl file. An `[Options]` section can be used to apply option to only one request (without passing options to the command line), while other requests are unaffected.

```
GET https://example.org
# An options section, each option is optional and applied only to this request...
[Options]
aws-sigv4: aws:amz:sts    # generate AWS SigV4 Authorization header
cacert: /etc/cert.pem     # custom certificate file
cert: /etc/client-cert.pem # client authentication certificate
key: /etc/client-cert.key  # client authentication certificate key
compressed: true          # request a compressed response
connect-timeout: 20s      # connect timeout
delay: 3s                 # delay for this request (aka sleep)
http3: true               # use HTTP/3 protocol version
insecure: true            # allow insecure SSL connections and transfers
ipv6: true                # use IPv6 addresses
limit-rate: 32000         # limit this request to the specidied speed (bytes/s)
location: true            # follow redirection for this request
max-redirs: 10            # maximum number of redirections
output: out.html          # dump the response to this file
path-as-is: true          # do not handle sequences of /../ or /./ in URL path
retry: 10                 # number of retry if HTTP/asserts errors
retry-interval: 500ms     # interval between retry
skip: false               # skip this request
unix-socket: sock         # use Unix socket for transfer
user: bob:secret          # use basic authentication
proxy: my.proxy:8012      # define proxy (host:port where host can be an IP addr
variable: country=Italy   # define variable country
variable: planet=Earth    # define variable planet
verbose: true             # allow verbose output
very-verbose: true        # allow more verbose output
```

> Variable defined in an `[Options]` section are defined also for the next entries. This is the exception, all other options are defined only for the current request.

Query parameters

Optional list of query parameters.

A query parameter consists of a field, followed by a `:` and a value. The query parameters section starts with `[Query]`. Contrary to query parameters in the URL, each value in the query parameters section is not URL encoded.

```
GET https://example.org/news
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:70.0) Gecko/2010010
[Query]
order: newest
search: {{custom-search}}
count: 100
```

If there are any parameters in the URL, the resulted request will have both parameters.

Form parameters

A form parameters section can be used to send data, like [HTML form](#).

This section contains an optional list of key values, each key followed by a `:` and a value. Key values will be encoded in key-value tuple separated by '&', with a '=' between the key and the value, and sent in the body request. The content type of the request is `application/x-www-form-urlencoded`. The form parameters section starts with `[Form]`.

```
POST https://example.org/contact
[Form]
default: false
token: {{token}}
email: john.doe@rookie.org
number: 33611223344
```

Form parameters section can be seen as syntactic sugar over body section (values in form parameters section are not URL encoded.). A [oneline string body](#) could be used instead of a forms parameters section.

```
# Run a POST request with form parameters section:
POST https://example.org/test
[Form]
name: John Doe
key1: value1

# Run the same POST request with a body section:
POST https://example.org/test
Content-Type: application/x-www-form-urlencoded
`name=John%20Doe&key1=value1`
```

When both [body section](#) and form parameters section are present, only the body section is taken into account.

Multipart Form Data

A multipart form data section can be used to send data, with key / value and file content (see [multipart/form-data on MDN](#)).

The form parameters section starts with `[Multipart]`.

```
POST https://example.org/upload
[Multipart]
field1: value1
field2: file,example.txt;
# One can specify the file content type:
field3: file,example.zip; application/zip
```

Files are relative to the input Hurl file, and cannot contain implicit parent directory (`..`). You can use [`--file-root` option](#) to specify the root directory of all file nodes.

Content type can be specified or inferred based on the filename extension:

- `.gif`: `image/gif`,
- `.jpg`: `image/jpeg`,
- `.jpeg`: `image/jpeg`,
- `.png`: `image/png`,
- `.svg`: `image/svg+xml`,
- `.txt`: `text/plain`,
- `.htm`: `text/html`,
- `.html`: `text/html`,

- `.pdf`: `application/pdf`,
- `.xml`: `application/xml`

By default, content type is `application/octet-stream`.

As an alternative to a `[Multipart]` section, multipart forms can also be sent with a [multiline string body](#):

```
POST https://example.org/upload
Content-Type: multipart/form-data; boundary="boundary"
```
--boundary
Content-Disposition: form-data; name="key1"

value1
--boundary
Content-Disposition: form-data; name="upload1"; filename="data.txt"
Content-Type: text/plain

Hello World!
--boundary
Content-Disposition: form-data; name="upload2"; filename="data.html"
Content-Type: text/html

<div>Hello World!</div>
--boundary--
```
```

> When using a multiline string body to send a multipart form data, files content must be inlined in
> the Hurl file.

Cookies

Optional list of session cookies for this request.

A cookie consists of a name, followed by a `:` and a value. Cookies are sent per request, and are not added to the cookie storage session, contrary to a cookie set in a header response. (for instance `Set-Cookie: theme=light`). The cookies section starts with `[Cookies]`.

```
GET https://example.org/index.html
[Cookies]
theme: light
sessionToken: abc123
```

Cookies section can be seen as syntactic sugar over corresponding request header.

```
# Run a GET request with cookies section:
GET https://example.org/index.html
[Cookies]
theme: light
sessionToken: abc123

# Run the same GET request with a header:
GET https://example.org/index.html
Cookie: theme=light; sessionToken=abc123
```

Basic Authentication

A basic authentication section can be used to perform [basic authentication](#).

Username is followed by a `:` and a password. The basic authentication section starts with

`[BasicAuth]`. Username and password are *not* base64 encoded.

```
# Perform basic authentication with login `bob` and password `secret`.
GET https://example.org/protected
[BasicAuth]
bob: secret
```

> Spaces surrounded username and password are trimmed. If you really want a space in your
> password (!!), you could use Hurl unicode literals \u{20}.

This is equivalent (but simpler) to construct the request with a Authorization header:

```
# Authorization header value can be computed with `echo -n 'bob:secret' | base64`
GET https://example.org/protected
Authorization: Basic Ym9iOnNlY3JldA==
```

Basic authentication allows per request authentication. If you want to add basic authentication to all
the requests of a Hurl file you can use –u/--user option.

Body

Optional HTTP body request.

If the body of the request is a JSON string or a XML string, the value can be directly inserted without
any modification. For a text based body that is neither JSON nor XML, one can use multiline string
body that starts with ``` and ends with ```. Multiline string body support "language hint" and can be
used to create GraphQL queries.

For a precise byte control of the request body, Base64 encoded string, hexadecimal string or
included file can be used to describe exactly the body byte content.

> You can set a body request even with a GET body, even if this is not a common practice.

The body section must be the last section of the request configuration.

JSON body

JSON request body is used to set a literal JSON as the request body.

```
# Create a new doggy thing with JSON body:
POST https://example.org/api/dogs
{
    "id": 0,
    "name": "Frieda",
    "picture": "images/scottish-terrier.jpeg",
    "age": 3,
    "breed": "Scottish Terrier",
    "location": "Lisco, Alabama"
}
```

JSON request body can be templatized with variables:

```
# Create a new catty thing with JSON body:
POST https://example.org/api/cats
{
    "id": 42,
    "lives": {{lives_count}},
    "name": "{{ name }}"
```

```
    }
```

When using JSON request body, the content type `application/json` is automatically set.

JSON request body can be seen as syntactic sugar of [multiline string body](#) with `json` identifier:

```
# Create a new doggy thing with JSON body:
POST https://example.org/api/dogs
```json
{
    "id": 0,
    "name": "Frieda",
    "picture": "images/scottish-terrier.jpeg",
    "age": 3,
    "breed": "Scottish Terrier",
    "location": "Lisco, Alabama"
}
```
```

**XML body**

XML request body is used to set a literal XML as the request body.

```
# Create a new soapy thing XML body:
POST https://example.org/InStock
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 299
SOAPAction: "http://www.w3.org/2003/05/soap-envelope"
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope" xmlns:m="http
  <soap:Header></soap:Header>
  <soap:Body>
    <m:GetStockPrice>
      <m:StockName>GOOG</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
```

XML request body can be seen as syntactic sugar of [multiline string body](#) with `xml` identifier:

```
# Create a new soapy thing XML body:
POST https://example.org/InStock
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 299
SOAPAction: "http://www.w3.org/2003/05/soap-envelope"
```xml
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope" xmlns:m="http
  <soap:Header></soap:Header>
  <soap:Body>
    <m:GetStockPrice>
      <m:StockName>GOOG</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
```
```

> Contrary to JSON body, the succinct syntax of XML body can not use variables. If you need to use variables in your XML body, use a simple [multiline string body](#) with variables.

**GraphQL query**

GraphQL query uses [multiline string body](#) with `graphql` identifier:

```
POST https://example.org/starwars/graphql
```graphql
{
  human(id: "1000") {
    name
    height(unit: FOOT)
  }
}
```
```

GraphQL query body can use [GraphQL variables](#):

```
POST https://example.org/starwars/graphql
```graphql
query Hero($episode: Episode, $withFriends: Boolean!) {
  hero(episode: $episode) {
    name
    friends @include(if: $withFriends) {
      name
    }
  }
}

variable {
  "episode": "JEDI",
  "withFriends": false
}
```
```

GraphQL query, as every multiline string body, can use Hurl variables.

```
POST https://example.org/starwars/graphql
```graphql
{
  human(id: "{{human_id}}") {
    name
    height(unit: FOOT)
  }
}
```
```

> Hurl variables and GraphQL variables can be mixed in the same body.

**Multiline string body**

For text based body that are neither JSON nor XML, one can use multiline string, started and ending with ```.

```
POST https://example.org/models
```
Year,Make,Model,Description,Price
1997,Ford,E350,"ac, abs, moon",3000.00
1999,Chevy,"Venture ""Extended Edition""","",4900.00
1999,Chevy,"Venture ""Extended Edition, Very Large""",,5000.00
1996,Jeep,Grand Cherokee,"MUST SELL! air, moon roof, loaded",4799.00
```
```

The standard usage of a multiline string is:

```
```
line1
line2
line3
```
```

is evaluated as "line1\nline2\nline3\n".

Multiline string body can use language identifier, like `json`, `xml` or `graphql`. Depending on the language identifier, an additional 'Content-Type' request header is sent, and the real body (bytes sent over the wire) can be different from the raw multiline text.

```
POST https://example.org/api/dogs
```json
{
    "id": 0,
    "name": "Frieda",
}
```
```

**Oneline string body**

For text based body that do not contain newlines, one can use oneline string, started and ending with `.

```
POST https://example.org/helloworld
`Hello world!`
```

Base64 body

Base64 body is used to set binary data as the request body.

Base64 body starts with `base64,` and end with `;`. MIME's Base64 encoding is supported (newlines and white spaces may be present anywhere but are to be ignored on decoding), and = padding characters might be added.

```
POST https://example.org
# Some random comments before body
base64,TG9yZW0gaXBzdW0gZG9sb3Igc2l0IGFtZXQsIGNvbnNlY3RldHVyIG
FkaXBpc2NpbmcgZWxpdC4gSW4gbWFsZXN1YWRhLCBuaXNsIHZlbCBkaWN0dW0g
aGVuZHJlcml0LCBlc3QganVzdG8gYmliZW5kdW0gbWV0dXMsIG5lYyBydXydW
0gdG9ydG9yIG1hc3NhIGlkIG1ldHVzLiA=;
```

Hex body

Hex body is used to set binary data as the request body.

Hex body starts with `hex,` and end with `;`.

```
PUT https://example.org
# Send a café, encoded in UTF-8
hex,636166c3a90a;
```

File body

To use the binary content of a local file as the body request, file body can be used. File body starts with `file,` and ends with `;`

```
POST https://example.org
# Some random comments before body
file,data.bin;
```

File are relative to the input Hurl file, and cannot contain implicit parent directory (`..`). You can use
`--file-root option` to specify the root directory of all file nodes.

Response

Definition

Responses can be used to capture values to perform subsequent requests, or add asserts to HTTP
responses. Response on requests are optional, a Hurl file can just consist of a sequence of
requests.

A response describes the expected HTTP response, with mandatory version and status, followed by
optional headers, captures, asserts and body. Assertions in the expected HTTP response describe
values of the received HTTP response. Captures capture values from the received HTTP response
and populate a set of named variables that can be used in the following entries.

Example

```
GET https://example.org
HTTP 200
Last-Modified: Wed, 21 Oct 2015 07:28:00 GMT
[Asserts]
xpath "normalize-space(//head/title)" startsWith "Welcome"
xpath "//li" count == 18
```

Structure

```
HTTP 200                          Version and status (mandatory if response present)
-----------------------
content-length: 206
accept-ranges: bytes              HTTP response headers (optional)
user-agent: Test
-----------------------
[Captures]
...                               Captures and asserts (optional sections, unordered)
[Asserts]
...
-----------------------
{
  "type": "FOO",
  "value": 356789,                HTTP response body (optional)
  "ordered": true,
  "index": 10
}
```

Capture and Assertion

With the response section, one can optionally capture value from headers, body, or add assert on
status code, body or headers.

Body compression

Hurl outputs the raw HTTP body to stdout by default. If response body is compressed (using br, gzip, deflate), the binary stream is output, without any modification. One can use --compressed option to request a compressed response and automatically get the decompressed body.

Captures and asserts work automatically on the decompressed body, so you can request compressed data (using Accept-Encoding header by example) and add assert and captures on the decoded body as if there weren't any compression.

Timings

HTTP response timings are exposed through Hurl structured output (see --json), HTML report (see --report-html) and JSON report (see --report-json).

On each response, libcurl response timings are available:

- **time_namelookup**: the time it took from the start until the name resolving was completed. You can use --resolve to exclude DNS performance from the measure.
- **time_connect**: The time it took from the start until the TCP connect to the remote host (or proxy) was completed.
- **time_appconnect**: The time it took from the start until the SSL/SSH/etc connect/handshake to the remote host was completed. The client is then ready to send its HTTP GET request.
- **time_starttransfer**: The time it took from the start until the first byte was just about to be transferred (just before Hurl reads the first byte from the network). This includes time_pretransfer and also the time the server needed to calculate the result.
- **time_total**: The total time that the full operation lasted.

All timings are in microsecond.

Capturing Response

Captures

Captures are optional values that are **extracted from the HTTP response** and stored in a named variable. These captures may be the response status code, part of or the entire the body, and response headers.

Captured variables can be accessed through a run session; each new value of a given variable overrides the last value.

Captures can be useful for using data from one request in another request, such as when working with [CSRF tokens](#). Variables in a Hurl file can be created from captures or [injected into the session](#).

```
# An example to show how to pass a CSRF token
# from one request to another:
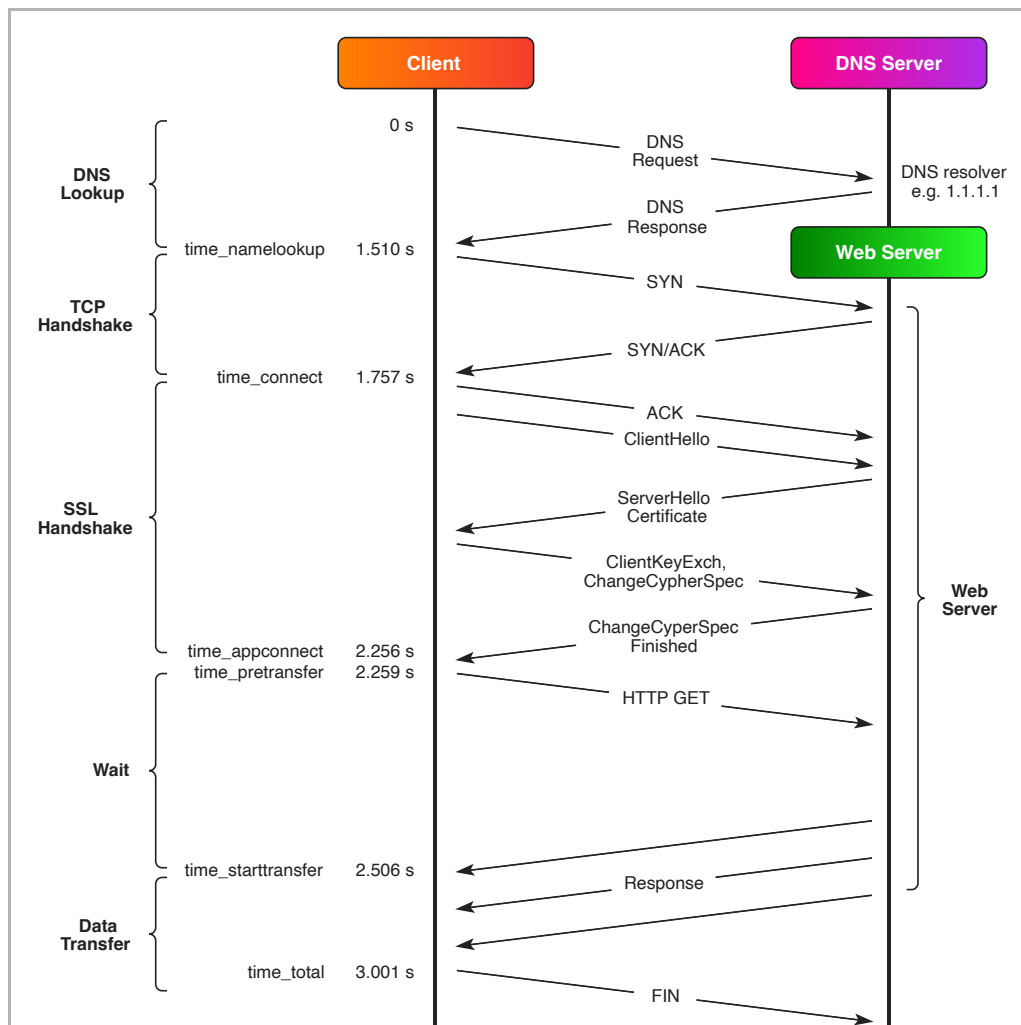
# First GET request to get CSRF token value:
GET https://example.org
HTTP 200
# Capture the CSRF token value from html body.
[Captures]
csrf_token: xpath "normalize-space(//meta[@name='_csrf_token']/@content)"

# Do the login !
POST https://acmecorp.net/login?user=toto&password=1234
X-CSRF-TOKEN: {{csrf_token}}
HTTP 302
```

Structure of a capture:

<div align="center">

my_var : xpath "string(//h1)"

variable   query

</div>

A capture consists of a variable name, followed by **:** and a query. Captures section starts with `[Captures]`.

Query

Queries are used to extract data from an HTTP response.

A query can extract data from

- status line:
  - [status](#)
  - [version](#)

- headers:
  - [header](#)
  - [cookie](#)

- body:
  - [body](#)
  - [bytes](#)

- xpath
- jsonpath
- regex
- sha256
- md5

- others:
  - url
  - ip
  - variable
  - duration
  - certificate

Extracted data can then be further refined using filters.

Status capture

Capture the received HTTP response status code. Status capture consists of a variable name, followed by a `:`, and the keyword `status`.

```
GET https://example.org
HTTP 200
[Captures]
my_status: status
```

Version capture

Capture the received HTTP version. Version capture consists of a variable name, followed by a `:`, and the keyword `version`. The value captured is a string:

```
GET https://example.org
HTTP 200
[Captures]
http_version: version
```

Header capture

Capture a header from the received HTTP response headers. Header capture consists of a variable name, followed by a `:`, then the keyword `header` and a header name.

```
POST https://example.org/login
[Form]
user: toto
password: 12345678
HTTP 302
[Captures]
next_url: header "Location"
```

Cookie capture

Capture a Set-Cookie header from the received HTTP response headers. Cookie capture consists of a variable name, followed by a `:`, then the keyword `cookie` and a cookie name.

```
GET https://example.org/cookies/set
HTTP 200
[Captures]
session-id: cookie "LSID"
```

Cookie attributes value can also be captured by using the following format: `<cookie-name>`

[cookie-attribute]. The following attributes are supported: Value, Expires, Max-Age, Domain, Path, Secure, HttpOnly and SameSite.

```
GET https://example.org/cookies/set
HTTP 200
[Captures]
value1: cookie "LSID"
value2: cookie "LSID[Value]"# Equivalent to the previous capture
expires: cookie "LSID[Expires]"
max-age: cookie "LSID[Max-Age]"
domain: cookie "LSID[Domain]"
path: cookie "LSID[Path]"
secure: cookie "LSID[Secure]"
http-only: cookie "LSID[HttpOnly]"
same-site: cookie "LSID[SameSite]"
```

Body capture

Capture the entire body (decoded as text) from the received HTTP response. The encoding used to decode the body is based on the charset value in the Content-Type header response.

```
GET https://example.org/home
HTTP 200
[Captures]
my_body: body
```

If the Content-Type doesn't include any encoding hint, a decode filter can be used to explicitly decode the body response bytes.

```
# Our HTML response is encoded using GB 2312.
# But, the 'Content-Type' HTTP response header doesn't precise any charset,
# so we decode explicitly the bytes.
GET https://example.org/cn
HTTP 200
[Captures]
my_body: bytes decode "gb2312"
```

Bytes capture

Capture the entire body (as a raw bytestream) from the received HTTP response

```
GET https://example.org/data.bin
HTTP 200
[Captures]
my_data: bytes
```

XPath capture

Capture a XPath query from the received HTTP body decoded as a string. Currently, only XPath 1.0 expression can be used.

```
GET https://example.org/home
# Capture the identifier from the dom node <div id="pet0">5646eaf23</div>
HTTP 200
[Captures]
pet-id: xpath "normalize-space(//div[@id='pet0'])"

# Open the captured page.
GET https://example.org/home/pets/{{pet-id}}
HTTP 200
```

XPath captures are not limited to node values (like string, or boolean); any valid XPath can be captured and asserted with variable asserts.

```
# Test that the XML endpoint return 200 pets
GET https://example.org/api/pets
HTTP 200
[Captures]
pets: xpath "//pets"
[Asserts]
variable "pets" count == 200
```

XPath expression can also be evaluated against part of the body with a xpath filter:

```
GET https://example.org/home_cn
HTTP 200
[Captures]
pet-id: bytes decode "gb2312" xpath "normalize-space(//div[@id='pet0'])"
```

JSONPath capture

Capture a JSONPath query from the received HTTP body.

```
POST https://example.org/api/contact
[Form]
token: {{token}}
email: toto@rookie.net
HTTP 200
[Captures]
contact-id: jsonpath "$['id']"
```

> Explain that the value selected by the JSONPath is coerced to a string when only one node is selected.

As with XPath captures, JSONPath captures can be anything from string, number, to object and collections. For instance, if we have a JSON endpoint that returns the following JSON:

```
{
  "a_null": null,
  "an_object": {
    "id": "123"
  },
  "a_list": [
    1,
    2,
    3
  ],
  "an_integer": 1,
  "a float": 1.1,
  "a_bool": true,
  "a_string": "hello"
}
```

We can capture the following paths:

```
GET https://example.org/captures-json
HTTP 200
```

```
[Captures]
an_object: jsonpath "$['an_object']"
a_list:    jsonpath "$['a_list']"
a_null:    jsonpath "$['a_null']"
an_integer: jsonpath "$['an_integer']"
a_float:   jsonpath "$['a_float']"
a_bool:    jsonpath "$['a_bool']"
a_string:  jsonpath "$['a_string']"
all:       jsonpath "$"
```

Regex capture

Capture a regex pattern from the HTTP received body, decoded as text.

```
GET https://example.org/helloworld
HTTP 200
[Captures]
id_a: regex "id_a:([0-9]+)"
id_b: regex "id_b:(\\d+)"# pattern using double quote
id_c: regex /id_c:(\d+)/# pattern using forward slash
name: regex "Hello ([a-zA-Z]+)"
```

The regex pattern must have at least one capture group, otherwise the capture will fail. When the pattern is a double-quoted string, metacharacters beginning with a backslash in the pattern (like \d, \s) must be escaped; literal pattern enclosed by / can also be used to avoid metacharacters escaping.

SHA-256 capture

Capture the SHA-256 hash of the response body.

```
GET https://example.org/data.tar.gz
HTTP 200
[Captures]
my_hash: sha256
```

Like body assert, sha256 capture works *after* content encoding decompression (so the captured value is not affected by Content-Encoding response header).

MD5 capture

Capture the MD5 hash of the response body.

```
GET https://example.org/data.tar.gz
HTTP 200
[Captures]
my_hash: md5
```

Like sha256 asserts, md5 assert works *after* content encoding decompression (so the predicates values are not affected by Content-Encoding response header)

URL capture

Capture the last fetched URL. This is most meaningful if you have told Hurl to follow redirection (see [[Options] section]options or --location option). URL capture consists of a variable name, followed by a :, and the keyword url.

```
GET https://example.org/redirecting
[Options]
location: true
```

```
HTTP 200
[Captures]
landing_url: url
```

IP address capture

Capture the IP address of the last connection. The value of the ip query is a string.

```
GET https://example.org/hello
HTTP 200
[Captures]
server_ip: ip
```

Variable capture

Capture the value of a variable into another.

```
GET https://example.org/helloworld
HTTP 200
[Captures]
in: body
name: variable "in"
```

Duration capture

Capture the response time of the request in ms.

```
GET https://example.org/helloworld
HTTP 200
[Captures]
duration_in_ms: duration
```

SSL certificate capture

Capture the SSL certificate properties. Certificate capture consists of the keyword certificate, followed by the certificate attribute value.

The following attributes are supported: Subject, Issuer, Start-Date, Expire-Date and Serial-Number.

```
GET https://example.org
HTTP 200
[Captures]
cert_subject: certificate "Subject"
cert_issuer: certificate "Issuer"
cert_expire_date: certificate "Expire-Date"
cert_serial_number: certificate "Serial-Number"
```

Redacting Secrets

When capturing data, you may need to hide captured values from logs and report. To do this, captures can use secrets which are redacted from logs and reports, using [--secret option](#):

```
$ hurl --secret pass=sesame-ouvre-toi file.hurl
```

If the secret value to be redacted is dynamic, or not known before execution, a capture can become a secret using `redact` at the end of the query's capture:

```
GET https://foo.com
HTTP 200
[Captures]
pass: header "token" redact
```

Asserting Response

Asserts

Asserts are used to test various properties of an HTTP response. Asserts can be implicits (such as version, status, headers) or explicit within an `[Asserts]` section. The delimiter of the request / response is `HTTP <STATUS-CODE>`: after this delimiter, you'll find the implicit asserts, then an `[Asserts]` section with all the explicit checks.

```
GET https://example.org/api/cats
HTTP 200
# Implicit assert on `Content-Type` Header
Content-Type: application/json; charset=utf-8
[Asserts]
# Explicit asserts section
bytes count == 120
header "Content-Type" contains "utf-8"
jsonpath "$.cats" count == 49
jsonpath "$.cats[0].name" == "Felix"
jsonpath "$.cats[0].lives" == 9
```

Body responses can be encoded by server (see [Content-Encoding HTTP header](#)) but asserts in Hurl files are not affected by this content compression. All body asserts (`body`, `bytes`, `sha256` etc...) work *after* content decoding.

Finally, body text asserts (`body`, `jsonpath`, `xpath` etc...) are also decoded to strings based on [Content-Type header](#) so these asserts can be written with usual strings.

Implicit asserts

Version - Status

Expected protocol version and status code of the HTTP response.

Protocol version is one of `HTTP/1.0`, `HTTP/1.1`, `HTTP/2`, `HTTP/3` or `HTTP`; `HTTP` describes any version. Note that there are no status text following the status code.

```
GET https://example.org/404.html
HTTP 404
```

Wildcard keywords `HTTP` and `*` can be used to disable tests on protocol version and status:

```
GET https://example.org/api/pets
HTTP *
# Check that response status code is > 400 and <= 500
[Asserts]
```

```
status > 400
status <= 500
```

While `HTTP/1.0`, `HTTP/1.1`, `HTTP/2` and `HTTP/3` explicitly check HTTP version:

```
# Check that our server responds with HTTP/2
GET https://example.org/api/pets
HTTP/2 200
```

Headers

Optional list of the expected HTTP response headers that must be in the received response.

A header consists of a name, followed by a `:` and a value.

For each expected header, the received response headers are checked. If the received header is not equal to the expected, or not present, an error is raised. The comparison is case-insensitive for the name: expecting a `Content-Type` header is equivalent to a `content-type` one. Note that the expected headers list is not fully descriptive: headers present in the response and not in the expected list doesn't raise error.

```
# Check that user toto is redirected to home after login.
POST https://example.org/login
[Form]
user: toto
password: 12345678
HTTP 302
Location: https://example.org/home
```

> Quotes in the header value are part of the value itself.
>
> This is used by the [ETag](#) Header `ETag: W/"<etag_value>"` `ETag: "<etag_value>"`

Testing duplicated headers is also possible.

For example with the `Set-Cookie` header:

```
Set-Cookie: theme=light
Set-Cookie: sessionToken=abc123; Expires=Wed, 09 Jun 2021 10:18:14 GMT
```

You can either test the two header values:

```
GET https://example.org/index.html
Host: example.net
HTTP 200
Set-Cookie: theme=light
Set-Cookie: sessionToken=abc123; Expires=Wed, 09 Jun 2021 10:18:14 GMT
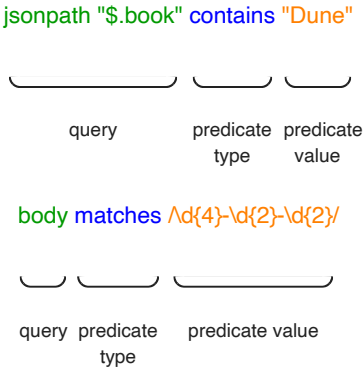```

Or only one:

```
GET https://example.org/index.html
Host: example.net
HTTP 200
Set-Cookie: theme=light
```

If you want to test specifically the number of headers returned for a given header name, or if you want to test header value with [predicates](#) (like `startsWith`, `contains`, `exists`)you can use the explicit [header assert](#).

Explicit asserts

Optional list of assertions on the HTTP response within an `[Asserts]` section. Assertions can describe checks on status code, on the received body (or part of it) and on response headers.

Structure of an assert:

jsonpath "$.book" contains "Dune"

query | predicate type | predicate value

body matches /\d{4}-\d{2}-\d{2}/

query | predicate type | predicate value

An assert consists of a query followed by a predicate. The format of the query is shared with [captures](#), and queries can extract data from

- status line:
  - [status](#)
  - [version](#)

- headers:
  - [header](#)
  - [cookie](#)

- body:
  - [body](#)
  - [bytes](#)
  - [xpath](#)
  - [jsonpath](#)
  - [regex](#)
  - [sha256](#)
  - [md5](#)

- others:
  - [url](#)
  - [ip](#)
  - [variable](#)
  - [duration](#)
  - [certificate](#)

Queries, in asserts and in captures, can be refined with [filters](#), like [count]count to add tests on collections sizes.

Predicates

Predicates consist of a predicate function and a predicate value. Predicate functions are:

Predicate	Description	Example
	Query and	

==	predicate value are equal	`jsonpath "$.book" == "Dune"`
!=	Query and predicate value are different	`jsonpath "$.color" != "red"`
>	Query number is greater than predicate value	`jsonpath "$.year" > 1978`
>=	Query number is greater than or equal to the predicate value	`jsonpath "$.year" >= 1978`
<	Query number is less than that predicate value	`jsonpath "$.year" < 1978`
<=	Query number is less than or equal to the predicate value	`jsonpath "$.year" <= 1978`

startsWith	Query starts with the predicate value Value is string or a binary content	`jsonpath "$.movie" startsWith "The"`
		
		`bytes startsWith hex,efbbbf;`

endsWith	Query ends with the predicate value Value is string or a binary content	`jsonpath "$.movie" endsWith "Back"`
		
		`bytes endsWith hex,ab23456;`

contains	Query contains the predicate value Value is string or a binary content	`jsonpath "$.movie" contains "Empire"`
		
		`bytes contains hex,beef;`

matches	Part of the query string matches the regex pattern described by the predicate value	`jsonpath "$.release" matches "\\d{4}"`
		
		`jsonpath "$.release" matches /\d{4}/`

exists	Query returns a value	`jsonpath "$.book" exists`
isBoolean	Query returns a boolean	`jsonpath "$.succeeded" isBoolean`
isCollection	Query returns a collection	`jsonpath "$.books" isCollection`
isEmpty	Query returns an empty collection	`jsonpath "$.movies" isEmpty`
isFloat	Query returns a float	`jsonpath "$.height" isFloat`
isInteger	Query returns an integer	`jsonpath "$.count" isInteger`

isIsoDate	Query string returns a [RFC 3339](#) date (YYYY-MM-DDTHH:mm:ss.sssZ)	`jsonpath "$.publication_date" isIsoDate`
isNumber	Query returns an integer or a float	`jsonpath "$.count" isNumber`
isString	Query returns a string	`jsonpath "$.name" isString`
isIpv4	Query returns an IPv4 address	`ip isIpv4`
isIpv6	Query returns an IPv6 address	`ip isIpv6`

Each predicate can be negated by prefixing it with `not` (for instance, `not contains` or `not exists`)

<div align="center">

jsonpath "$.book" not contains "Dune"

| query | predicate type | predicate value |

</div>

A predicate value is typed, and can be a string, a boolean, a number, a bytestream, `null` or a collection. Note that `"true"` is a string, whereas `true` is a boolean.

For instance, to test the presence of a h1 node in an HTML response, the following assert can be used:

```
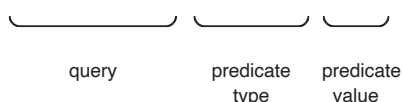GET https://example.org/home
HTTP 200
[Asserts]
xpath "boolean(count(//h1))" == true
xpath "//h1" exists # Equivalent but simpler
```

As the XPath query `boolean(count(//h1))` returns a boolean, the predicate value in the assert must be either `true` or `false` without double quotes. On the other side, say you have an article node and you want to check the value of some [data attributes](#):

```
<article
  id="electric-cars"
  data-visible="true"
...
</article>
```

The following assert will check the value of the `data-visible` attribute:

```
GET https://example.org/home
HTTP 200
[Asserts]
xpath "string(//article/@data-visible)" == "true"
```

In this case, the XPath query `string(//article/@data-visible)` returns a string, so the predicate value must be a string.

The predicate function `==` can be used with string, numbers or booleans; `startWith` and `contains` can only be used with strings and bytes, while `matches` only works on string. If a query returns a number, using a `matches` predicate will cause a runner error.

```
# A really well tested web page...
GET https://example.org/home
HTTP 200
[Asserts]
header "Content-Type" contains "text/html"
header "Last-Modified" == "Wed, 21 Oct 2015 07:28:00 GMT"
xpath "//h1" exists  # Check we've at least one h1
xpath "normalize-space(//h1)" contains "Welcome"
xpath "//h2" count == 13
xpath "string(//article/@data-id)" startsWith "electric"
```

Status assert

Check the received HTTP response status code. Status assert consists of the keyword `status` followed by a predicate function and value.

```
GET https://example.org
HTTP *
[Asserts]
status < 300
```

Version assert

Check the received HTTP version. Version assert consists of the keyword `version` followed by a predicate function and value. The value returns by `version` is a string:

```
GET https://example.org
HTTP *
[Asserts]
version == "2"
```

Header assert

Check the value of a received HTTP response header. Header assert consists of the keyword `header` followed by the value of the header, a predicate function and a predicate value. Like [headers implicit asserts](), the check is case-insensitive for the name: comparing a `Content-Type` header is equivalent to a `content-type` one.

```
GET https://example.org
HTTP 302
[Asserts]
header "Location" contains "www.example.net"
header "Last-Modified" matches /\d{2} [a-z-A-Z]{3} \d{4}/
```

If there are multiple headers with the same name, the header assert returns a collection, so `count`, `contains` can be used in this case to test the header list.

Let's say we have this request and response:

```
> GET /hello HTTP/1.1
> Host: example.org
> Accept: */*
> User-Agent: hurl/2.0.0-SNAPSHOT
>
* Response: (received 12 bytes in 11 ms)
```

```
*
< HTTP/1.0 200 OK
< Vary: Content-Type
< Vary: User-Agent
< Content-Type: text/html; charset=utf-8
< Content-Length: 12
< Server: Flask Server
< Date: Fri, 07 Oct 2022 20:53:35 GMT
```

One can use explicit header asserts:

```
GET https://example.org/hello
HTTP 200
[Asserts]
header "Vary" count == 2
header "Vary" contains "User-Agent"
header "Vary" contains "Content-Type"
```

Or implicit header asserts:

```
GET https://example.org/hello
HTTP 200
Vary: User-Agent
Vary: Content-Type
```

Cookie assert

Check value or attributes of a Set-Cookie response header. Cookie assert consists of the keyword
cookie, followed by the cookie name (and optionally a cookie attribute), a predicate function and
value.

Cookie attributes value can be checked by using the following format:<cookie-name>[cookie-
attribute]. The following attributes are supported: Value, Expires, Max-Age, Domain, Path,
Secure, HttpOnly and SameSite.

```
GET http://localhost:8000/cookies/set
HTTP 200

# Explicit check of Set-Cookie header value. If the attributes are
# not in this exact order, this assert will fail.
Set-Cookie: LSID=DQAAAKEaem_vYg; Expires=Wed, 13 Jan 2021 22:23:01 GMT; Secure; I
Set-Cookie: HSID=AYQEVnDKrdst; Domain=localhost; Expires=Wed, 13 Jan 2021 22:23:(
Set-Cookie: SSID=Ap4PGTEq; Domain=localhost; Expires=Wed, 13 Jan 2021 22:23:01 GN

# Using cookie assert, one can check cookie value and various attributes.
[Asserts]
cookie "LSID" == "DQAAAKEaem_vYg"
cookie "LSID[Value]" == "DQAAAKEaem_vYg"
cookie "LSID[Expires]" exists
cookie "LSID[Expires]" contains "Wed, 13 Jan 2021"
cookie "LSID[Max-Age]" not exists
cookie "LSID[Domain]" not exists
cookie "LSID[Path]" == "/accounts"
cookie "LSID[Secure]" exists
cookie "LSID[HttpOnly]" exists
cookie "LSID[SameSite]" == "Lax"
```

> Secure and HttpOnly attributes can only be tested with exists or not exists predicates to
> reflect the Set-Cookie header semantics (in other words, queries <cookie-name>[HttpOnly]
> and <cookie-name>[Secure] don't return boolean).

Body assert

Check the value of the received HTTP response body when decoded as a string. Body assert consists of the keyword `body` followed by a predicate function and value.

```
GET https://example.org
HTTP 200
[Asserts]
body contains "<h1>Welcome!</h1>"
```

The encoding used to decode the response body bytes to a string is based on the `charset` value in the `Content-Type` header response.

```
# Our HTML response is encoded with GB 2312 (see https://en.wikipedia.org/wiki/GB
GET https://example.org/cn
HTTP 200
[Asserts]
header "Content-Type" == "text/html; charset=gb2312"
# bytes of the response, without any text decoding:
bytes contains hex,c4e3bac3cac0bde7; # 你好世界 encoded in GB 2312
# text of the response, decoded with GB 2312:
body contains "你好世界"
```

If the `Content-Type` response header doesn't include any encoding hint, a [decode filter](#) can be used to explicitly decode the response body bytes.

```
# Our HTML response is encoded using GB 2312.
# But, the 'Content-Type' HTTP response header doesn't precise any charset,
# so we decode explicitly the bytes.
GET https://example.org/cn
HTTP 200
[Asserts]
header "Content-Type" == "text/html"
bytes contains hex,c4e3bac3cac0bde7; # 你好世界 encoded in GB2312
bytes decode "gb2312" contains "你好世界"
```

Body asserts are automatically decompressed based on the value of `Content-Encoding` response header. So, whatever is the response compression (`gzip`, `brotli`) etc... asserts values don't depend on the content encoding.

```
# Request a gzipped reponse, the `body` asserts works with ungzipped response
GET https://example.org
Accept-Encoding: gzip
HTTP 200
[Asserts]
header "Content-Encoding" == "gzip"
body contains "<h1>Welcome!</h1>"

# Without content encoding, asserts remains identical
GET https://example.org
HTTP 200
[Asserts]
header "Content-Encoding" not exists
body contains "<h1>Welcome!</h1>"
```

Bytes assert

Check the value of the received HTTP response body as a bytestream. Body assert consists of the keyword `bytes` followed by a predicate function and value.

```
GET https://example.org/data.bin
HTTP 200
[Asserts]
bytes startsWith hex,efbbbf;
bytes count == 12424
header "Content-Length" == "12424"
```

Like body assert, bytes assert works *after* content encoding decompression (so the predicates values are not affected by Content-Encoding response header value).

XPath assert

Check the value of a XPath query on the received HTTP body decoded as a string (using the charset value in the Content-Type header response). Currently, only XPath 1.0 expression can be used. Body assert consists of the keyword xpath followed by a predicate function and value. Values can be string, boolean or number depending on the XPath query.

Let's say we want to check this HTML response:

```
$ curl -v https://example.org

< HTTP/1.1 200 OK
< Content-Type: text/html; charset=UTF-8
...
<!doctype html>
<html>
  <head>
    <title>Example Domain</title>
    ...
  </head>

  <body>
    <div>
      <h1>Example</h1>
      <p>This domain is for use in illustrative examples in documents. You may u;
      <p><a href="https://www.iana.org/domains/example">More information...</a></
    </div>
  </body>
</html>
```

With Hurl, we can write multiple XPath asserts describing the DOM content:

```
GET https://example.org
HTTP 200
Content-Type: text/html; charset=UTF-8
[Asserts]
xpath "string(/html/head/title)" contains "Example"  # Check title
xpath "count(//p)" == 2                               # Check the number of <p>
xpath "//p" count == 2                                # Similar assert for <p>
xpath "boolean(count(//h2))" == false                 # Check there is no <h2>
xpath "//h2" not exists                                # Similar assert for <h2>
```

XML Namespaces are also supported. Let's say you want to check this XML response:

```
<?xml version="1.0"?>
<!-- both namespace prefixes are available throughout -->
<bk:book xmlns:bk='urn:loc.gov:books'
         xmlns:isbn='urn:ISBN:0-395-36341-6'>
    <bk:title>Cheaper by the Dozen</bk:title>
    <isbn:number>1568491379</isbn:number>
</bk:book>
```

This XML response can be tested with the following Hurl file:

```
GET http://localhost:8000/assert-xpath
HTTP 200
[Asserts]

xpath "string(//bk:book/bk:title)" == "Cheaper by the Dozen"
xpath "string(//*[name()='bk:book']/*[name()='bk:title'])" == "Cheaper by the Doz
xpath "string(//*[local-name()='book']/*[local-name()='title'])" == "Cheaper by
xpath "string(//bk:book/isbn:number)" == "1568491379"
xpath "string(//*[name()='bk:book']/*[name()='isbn:number'])" == "1568491379"
xpath "string(//*[local-name()='book']/*[local-name()='number'])" == "1568491379"
```

The XPath expressions `string(//bk:book/bk:title)` and `string(//bk:book/isbn:number)` are
written with `bk` and `isbn` namespaces.

> For convenience, the first default namespace can be used with _

JSONPath assert

Check the value of a [JSONPath](#) query on the received HTTP body decoded as a JSON document.
JSONPath assert consists of the keyword `jsonpath` followed by a predicate function and value.

Let's say we want to check this JSON response:

```
curl -v http://httpbin.org/json

< HTTP/1.1 200 OK
< Content-Type: application/json
...

{
  "slideshow": {
    "author": "Yours Truly",
    "date": "date of publication",
    "slides": [
      {
        "title": "Wake up to WonderWidgets!",
        "type": "all"
      },
      ...
    ],
    "title": "Sample Slide Show"
  }
}
```

With Hurl, we can write multiple JSONPath asserts describing the DOM content:

```
GET http://httpbin.org/json
HTTP 200
[Asserts]
jsonpath "$.slideshow.author" == "Yours Truly"
jsonpath "$.slideshow.slides[0].title" contains "Wonder"
jsonpath "$.slideshow.slides" count == 2
jsonpath "$.slideshow.date" != null
jsonpath "$.slideshow.slides[*].title" contains "Mind Blowing!"
```

> Explain that the value selected by the JSONPath is coerced to a string when only one node is
> selected.

In `matches` predicates, metacharacters beginning with a backslash (like `\d`, `\s`) must be escaped. Alternatively, `matches` predicate support [JavaScript-like Regular expression syntax](#) to enhance the readability:

```
GET https://example.org/hello
HTTP 200
[Asserts]

# Predicate value with matches predicate:
jsonpath "$.date" matches "^\\d{4}-\\d{2}-\\d{2}$"
jsonpath "$.name" matches "Hello [a-zA-Z]+!"

# Equivalent syntax:
jsonpath "$.date" matches /^\d{4}-\d{2}-\d{2}$/
jsonpath "$.name" matches /Hello [a-zA-Z]+!/
```

Regex assert

Check that the HTTP received body, decoded as text, matches a regex pattern.

```
GET https://example.org/hello
HTTP 200
[Asserts]
regex "^(\\d{4}-\\d{2}-\\d{2})$" == "2018-12-31"
# Same assert as previous using regex literals
regex /^(\d{4}-\d{2}-\d{2})$/ == "2018-12-31"
```

The regex pattern must have at least one capture group, otherwise the assert will fail. The assertion is done on the captured group value. When the regex pattern is a double-quoted string, metacharacters beginning with a backslash in the pattern (like `\d`, `\s`) must be escaped; literal pattern enclosed by `/` can also be used to avoid metacharacters escaping.

SHA-256 assert

Check response body [SHA-256](#) hash.

```
GET https://example.org/data.tar.gz
HTTP 200
[Asserts]
sha256 == hex,039058c6f2c0cb492c533b0a4d14ef77cc0f78abccced5287d84a1a2011cfb81;
```

Like `body` assert, `sha256` assert works *after* content encoding decompression (so the predicates values are not affected by `Content-Encoding` response header). For instance, if we have a resource `a.txt` on a server with a given hash `abcdef`, `sha256` value is not affected by `Content-Encoding`:

```
# Without content encoding compression:
GET https://example.org/a.txt
HTTP 200
[Asserts]
sha256 == hex,abcdef;

# With content encoding compression:
GET https://example.org/a.txt
Accept-Encoding: brotli
HTTP 200
[Asserts]
header "Content-Encoding" == "brotli"
sha256 == hex,abcdef;
```

MD5 assert

Check response body [MD5](#) hash.

```
GET https://example.org/data.tar.gz
HTTP 200
[Asserts]
md5 == hex,ed076287532e86365e841e92bfc50d8c;
```

Like `sha256` asserts, `md5` assert works *after* content encoding decompression (so the predicates values are not affected by `Content-Encoding` response header)

URL assert

Check the last fetched URL. This is most meaningful if you have told Hurl to follow redirection (see [[Options]section]options or [--location option](#)). URL assert consists of the keyword `url` followed by a predicate function and value.

```
GET https://example.org/redirecting
[Options]
location: true
HTTP 200
[Asserts]
url == "https://example.org/redirected"
```

IP address assert

Check the IP address of the last connection. The value of the `ip` query is a string.

> Predicates `isIpv4` and `isIpv6` are available to check if a particular string matches an IPv4 or IPv6 address and can use with `ip` queries.

```
GET https://example.org/hello
HTTP 200
[Asserts]
ip isIpv4
ip not isIpv6
ip == "172.16.45.87"
```

Variable assert

```
# Test that the XML endpoint return 200 pets
GET https://example.org/api/pets
HTTP 200
[Captures]
pets: xpath "//pets"
[Asserts]
variable "pets" count == 200
```

Duration assert

Check the total duration (sending plus receiving time) of the HTTP transaction.

```
GET https://example.org/helloworld
HTTP 200
[Asserts]
duration < 1000   # Check that response time is less than one second
```

SSL certificate assert

Check the SSL certificate properties. Certificate assert consists of the keyword `certificate`, followed by the certificate attribute value.

The following attributes are supported: `Subject`, `Issuer`, `Start-Date`, `Expire-Date` and `Serial-Number`.

```
GET https://example.org
HTTP 200
[Asserts]
certificate "Subject" == "CN=example.org"
certificate "Issuer" == "C=US, O=Let's Encrypt, CN=R3"
certificate "Expire-Date" daysAfterNow > 15
certificate "Serial-Number" matches "[0-9af]+"
```

Body

Optional assertion on the received HTTP response body. Body section can be seen as syntactic sugar over body asserts (with == predicate). If the body of the response is a JSON string or a XML string, the body assertion can be directly inserted without any modification. For a text based body that is neither JSON nor XML, one can use multiline string that starts with ``` ``` ``` and ends with ``` ``` ```. For a precise byte control of the response body, a Base64 encoded string or an input file can be used to describe exactly the body byte content to check.

Like explicit body assert, the body section is automatically decompressed based on the value of `Content-Encoding` response header. So, whatever is the response compression (`gzip`, `brotli`, etc...) body section doesn't depend on the content encoding. For textual body sections (JSON, XML, multiline, etc...), content is also decoded to string, based on the value of `Content-Type` response header.

JSON body

```
# Get a doggy thing:
GET https://example.org/api/dogs/{{dog-id}}
HTTP 200
{
    "id": 0,
    "name": "Frieda",
    "picture": "images/scottish-terrier.jpeg",
    "age": 3,
    "breed": "Scottish Terrier",
    "location": "Lisco, Alabama"
}
```

JSON response body can be seen as syntactic sugar of multiline string body with `json` identifier:

```
# Get a doggy thing:
GET https://example.org/api/dogs/{{dog-id}}
HTTP 200
```json
{
    "id": 0,
    "name": "Frieda",
    "picture": "images/scottish-terrier.jpeg",
    "age": 3,
    "breed": "Scottish Terrier",
    "location": "Lisco, Alabama"
}
```
```

XML body

```
GET https://example.org/api/catalog
HTTP 200
<?xml version="1.0" encoding="UTF-8"?>
<catalog>
   <book id="bk101">
      <author>Gambardella, Matthew</author>
      <title>XML Developer's Guide</title>
      <genre>Computer</genre>
      <price>44.95</price>
      <publish_date>2000-10-01</publish_date>
      <description>An in-depth look at creating applications with XML.</descripti
   </book>
</catalog>
```

XML response body can be seen as syntactic sugar of [multiline string body](#) with `xml` identifier:

```
GET https://example.org/api/catalog
HTTP 200
```xml
<?xml version="1.0" encoding="UTF-8"?>
<catalog>
   <book id="bk101">
      <author>Gambardella, Matthew</author>
      <title>XML Developer's Guide</title>
      <genre>Computer</genre>
      <price>44.95</price>
      <publish_date>2000-10-01</publish_date>
      <description>An in-depth look at creating applications with XML.</descripti
   </book>
</catalog>
```
```

Multiline string body

```
GET https://example.org/models
HTTP 200
```
Year,Make,Model,Description,Price
1997,Ford,E350,"ac, abs, moon",3000.00
1999,Chevy,"Venture ""Extended Edition""","",4900.00
1999,Chevy,"Venture ""Extended Edition, Very Large""",,5000.00
1996,Jeep,Grand Cherokee,"MUST SELL! air, moon roof, loaded",4799.00
```
```

The standard usage of a multiline string is :

```
```
line1
line2
line3
```
```

Oneline string body

For text based response body that do not contain newlines, one can use oneline string, started and ending with `` ` ``.

```
POST https://example.org/helloworld
HTTP 200
```

```
    `Hello world!`
```

Base64 body

Base64 response body assert starts with `base64,` and end with `;`. MIME's Base64 encoding is supported (newlines and white spaces may be present anywhere but are to be ignored on decoding), and = padding characters might be added.

```
GET https://example.org
HTTP 200
base64,TG9yZW0gaXBzdW0gZG9sb3Igc2l0IGFtZXQsIGNvbnNlY3RldHVyIG
FkaXBpc2NpbmcgZWxpdC4gSW4gbWFsZXN1YWRhLCBuaXNsIHZlbCBkaWN0dW0g
aGVuZHJlcml0LCBlc3QganVzdG8gYmliZW5kdW0gbWV0dXMsIG5lYyBydW0
0gdG9ydG9yIG1hc3NhIGlkIG1ldHVzLiA=;
```

File body

To use the binary content of a local file as the body response assert, file body can be used. File body starts with `file,` and ends with `;`

```
GET https://example.org
HTTP 200
file,data.bin;
```

File are relative to the input Hurl file, and cannot contain implicit parent directory (`..`). You can use [`--file-root option`](#) to specify the root directory of all file nodes.

Filters

Definition

[Captures](#) and [asserts](#) share a common structure: query. A query is used to extract data from an HTTP response; this data can come from the HTTP response body, the HTTP response headers or from the HTTP meta-information (like `duration` for instance)...

In this example, the query **jsonpath "$.books[0].name"** is used in a capture to save data and in an assert to test the HTTP response body.

Capture:

name : jsonpath "$.books[0].name"

variable   query

Assert:

jsonpath "$.books[0].name" == "Dune"

query   predicate

In both case, the query is exactly the same: queries are the core structure of asserts and captures.

Sometimes, you want to process data extracted by queries: that's the purpose of **filters**.

Filters are used to transform value extracted by a query and can be used in asserts and captures to refine data. Filters **can be chained**, allowing for fine-grained data extraction.

jsonpath "$.name" split "," nth 0 == "Herbert"

| query | 2 filters | predicate |

Example

```
GET https://example.org/api
HTTP 200
[Captures]
name: jsonpath "$user.id" replace /\d/ "x"
[Asserts]
header "x-servers" split "," count == 2
header "x-servers" split "," nth 0 == "rec1"
header "x-servers" split "," nth 1 == "rec3"
jsonpath "$.books" count == 12
```

Description

base64Decode

Decode a base 64 encoded string into bytes.

```
GET https://example.org/api
HTTP 200
[Asserts]
jsonpath "$.token" base64Decode == hex,e4bda0e5a5bde4b896e7958c;
```

base64Encode

Encode bytes into base 64 encoded string.

```
GET https://example.org/api
HTTP 200
[Asserts]
bytes base64Encode == "5L2g5aW95LiW55WM"
```

count

Counts the number of items in a collection.

```
GET https://example.org/api
HTTP 200
[Asserts]
jsonpath "$.books" count == 12
```

daysAfterNow

Returns the number of days between now and a date in the future.

```
GET https://example.org
HTTP 200
[Asserts]
certificate "Expire-Date" daysAfterNow > 15
```

daysBeforeNow

Returns the number of days between now and a date in the past.

```
GET https://example.org
HTTP 200
[Asserts]
certificate "Start-Date" daysBeforeNow < 100
```

decode

Decode bytes to string using encoding.

```
# The 'Content-Type' HTTP response header does not precise the charset 'gb2312'
# so body must be decoded explicitly by Hurl before processing any text based ass
GET https://example.org/hello_china
HTTP 200
[Asserts]
header "Content-Type" == "text/html"
# Content-Type has no encoding clue, we must decode ourselves the body response.
bytes decode "gb2312" xpath "string(//body)" == "你好世界"
```

format

Formats a date to a string given [a specification format](#).

```
GET https://example.org
HTTP 200
[Asserts]
cookie "LSID[Expires]" format "%a, %d %b %Y %H:%M:%S" == "Wed, 13 Jan 2021 22:23
```

htmlEscape

Converts the characters &, < and > to HTML-safe sequence.

```
GET https://example.org/api
HTTP 200
[Asserts]
jsonpath "$.text" htmlEscape == "a &gt; b"
```

htmlUnescape

Converts all named and numeric character references (e.g. >, >, >) to the corresponding Unicode characters.

```
GET https://example.org/api
HTTP 200
[Asserts]
jsonpath "$.escaped_html[1]" htmlUnescape == "Foo © bar ☰"
```

jsonpath

Evaluates a [JSONPath](#) expression.

```
GET https://example.org/api
HTTP 200
[Captures]
books: xpath "string(//body/@data-books)"
[Asserts]
variable "books" jsonpath "$[0].name" == "Dune"
variable "books" jsonpath "$[0].author" == "Franck Herbert"
```

nth

Returns the element from a collection at a zero-based index.

```
GET https://example.org/api
HTTP 200
[Asserts]
jsonpath "$.books" nth 2 == "Children of Dune"
```

regex

Extracts regex capture group. Pattern must have at least one capture group.

```
GET https://example.org/foo
HTTP 200
[Captures]
param1: header "header1"
param2: header "header2" regex "Hello (.*)!"
param3: header "header2" regex /Hello (.*)!/
```

replace

Replaces all occurrences of old string with new string.

```
GET https://example.org/foo
HTTP 200
[Captures]
url: jsonpath "$.url" replace "http://" "https://"
[Asserts]
jsonpath "$.ips" replace ", " "|" == "192.168.2.1|10.0.0.20|10.0.0.10"
```

split

Splits to a list of strings around occurrences of the specified delimiter.

```
GET https://example.org/foo
HTTP 200
[Asserts]
jsonpath "$.ips" split ", " count == 3
```

toDate

Converts a string to a date given [a specification format](#).

```
GET https:///example.org
```

```
HTTP 200
[Asserts]
header "Expires" toDate "%a, %d %b %Y %H:%M:%S GMT" daysBeforeNow > 1000
```

ISO 8601 / RFC 3339 date and time format have shorthand format %+:

```
GET https://example.org/api/books
HTTP 200
[Asserts]
jsonpath "$.published" == "2023-01-23T18:25:43.511Z"
jsonpath "$.published" toDate "%Y-%m-%dT%H:%M:%S%.fZ" format "%A" == "Monday"
jsonpath "$.published" toDate "%+" format "%A" == "Monday" # %+ can be used to pa
```

toFloat

Converts value to float number.

```
GET https://example.org/foo
HTTP 200
[Asserts]
jsonpath "$.pi" toFloat == 3.14
```

toInt

Converts value to integer number.

```
GET https://example.org/foo
HTTP 200
[Asserts]
jsonpath "$.id" toInt == 123
```

toString

Converts value to string.

```
GET https://example.org/foo
HTTP 200
[Asserts]
jsonpath "$.count" toString == "42"
```

urlDecode

Replaces %xx escapes with their single-character equivalent.

```
GET https://example.org/foo
HTTP 200
[Asserts]
jsonpath "$.encoded_url" urlDecode == "https://mozilla.org/?x=шеллы"
```

urlEncode

Percent-encodes all the characters which are not included in unreserved chars (see [RFC3986](RFC3986)) with the exception of forward slash (/).

```
GET https://example.org/foo
HTTP 200
```

```
   [Asserts]
   jsonpath "$.url" urlEncode == "https%3A//mozilla.org/%3Fx%3D%D1%88%D0%B5%D0%BB%D(
```

xpath

Evaluates a [XPath](#) expression.

```
GET https://example.org/hello_gb2312
HTTP 200
[Asserts]
bytes decode "gb2312" xpath "string(//body)" == "你好世界"
```

Templates

Variables

In Hurl file, you can generate value using two curly braces, i.e {{my_variable}}. For instance, if you want to reuse a value from an HTTP response in the next entries, you can capture this value in a variable and reuse it in a placeholder.

In this example, we capture the value of a [CSRF token](#) from the body of the first response, and inject it as a header in the next POST request:

```
GET https://example.org
HTTP 200
[Captures]
csrf_token: xpath "string(//meta[@name='_csrf_token']/@content)"

# Do the login !
POST https://acmecorp.net/login?user=toto&password=1234
X-CSRF-TOKEN: {{csrf_token}}
HTTP 302
```

In this second example, we capture the body in a variable index, and reuse this value in the query jsonpath "$.errors[{{index}}].id":

```
GET https://example.org/api/index
HTTP 200
[Captures]
index: body

GET https://example.org/api/status
HTTP 200
[Asserts]
jsonpath "$.errors[{{index}}].id" == "error"
```

Functions

Besides variables, functions can be used to generate dynamic values. Current functions are:

Function	Description

newUuid	Generates an [UUID v4 random string](#)
newDate	Generates an [RFC 3339](#) UTC date string, at the current time

In the following example, we use `newDate` to generate a dynamic query parameter:

```
GET https://example.org/api/foo
[Query]
date: {{newDate}}
HTTP 200
```

We run a `GET` request to `https://example.org/api/foo?`
`date=2024%2D12%2D02T10%3A35%3A44%2E461731Z` where the `date` query parameter value is `2024-`
`12-02T10:35:44.461731Z` URL encoded.

In this second example, we use `newUuid` function to generate an email dynamically:

```
POST https://example.org/api/foo
{
  "name": "foo",
  "email": "{{newUuid}}@test.com"
}
```

When run, the request body will be:

```
{
  "name": "foo",
  "email": "0531f78f-7f87-44be-a7f2-969a1c4e6d97@test.com"
}
```

Types

Values generated from function and variables are typed, and can be either string, bool, number, `null` or collections. Depending on the value type, templates can be rendered differently. Let's say we have captured an integer value into a variable named `count`:

```
GET https://sample/counter

HTTP 200
[Captures]
count: jsonpath "$.results[0]"
```

The following entry:

```
GET https://sample/counter/{{count}}

HTTP 200
[Asserts]
jsonpath "$.id" == "{{count}}"
```

will be rendered at runtime to:

```
GET https://sample/counter/458

HTTP 200
[Asserts]
jsonpath "$.id" == "458"
```

resulting in a comparison between the JSONPath expression and a string value.

On the other hand, the following assert:

```
GET https://sample/counter/{{count}}

HTTP 200
[Asserts]
jsonpath "$.index" == {{count}}
```

will be rendered at runtime to:

```
GET https://sample/counter/458

HTTP 200
[Asserts]
jsonpath "$.index" == 458
```

resulting in a comparison between the JSONPath expression and an integer value.

So if you want to use typed values (in asserts for instances), you can use {{my_var}}. If you're interested in the string representation of a variable, you can surround the variable with double quotes , as in "{{my_var}}".

> When there is no possible ambiguities, like using a variable in an URL, or in a header, you can omit the double quotes. The value will always be rendered as a string.

Injecting Variables

Variables can be injected in a Hurl file:

- by using --variable option
- by using --variables-file option
- by defining environment variables, for instance HURL_foo=bar
- by defining variables in an [[Options] section]options

Lets' see how to inject variables, given this test.hurl:

```
GET https://{{host}}/{{id}}/status
HTTP 304

GET https://{{host}}/health
HTTP 200
```

variable option

Variable can be defined with command line option:

```
$ hurl --variable host=example.net --variable id=1234 test.hurl
```

`variables-file` option

We can also define all injected variables in a file:

```
$ hurl --variables-file vars.env test.hurl
```

where `vars.env` is

```
host=example.net
id=1234
```

Environment variable

We can use environment variables in the form of `HURL_name=value`:

```
$ export HURL_host=example.net
$ export HURL_id=1234
$ hurl test.hurl
```

Options sections

We can define variables in `[Options]` section. Variables defined in a section are available for the next requests.

```
GET https://{{host}}/{{id}}/status
[Options]
variable: host=example.net
variable: id=1234
HTTP 304

GET https://{{host}}/health
HTTP 200
```

Secrets

Secrets are variables which value is redacted from standard error logs (for instance using [--very-verbose](#)) and [reports](#). Secrets are injected through command-line with [--secret option](#):

```
$ hurl --secret token=FooBar test.hurl
```

Values are redacted by *exact matching*: if a secret value is transformed, and you want to redact also the transformed value, you can add as many secrets as there are transformed values. Even if a secret is not used as a variable, all secrets values will be redacted from messages and logs.

```
$ hurl --secret token=FooBar \
       --secret token_alt_0=FOOBAR \
       --secret token_alt_1=foobar \
       test.hurl
```

> Secrets **are not redacted** from HTTP responses outputted on standard output as Hurl considers the standard output as the correct unaltered output of a run. With this call `$ hurl --secret token=FooBar test.hurl`, the HTTP response is outputted unaltered and `FooBar` can appear in

Templating Body

Variables and functions can be used in JSON body:

```
PUT https://example.org/api/hits
{
    "key0": "{{a_string}}",
    "key1": {{a_bool}},
    "key2": {{a_null}},
    "key3": {{a_number}},
    "key4": "{{newDate}}"
}
```

Note that we're writing a kind of JSON body directly without any delimitation marker. For the moment, XML body can't use variables directly. In order to templatize a XML body, you can use multiline string body with variables and functions. The multiline string body allows to templatize any text based body (JSON, XML, CSV etc...):

Multiline string body delimited by `:

```
PUT https://example.org/api/hits
Content-Type: application/json
```
{
    "key0": "{{a_string}}",
    "key1": {{a_bool}},
    "key2": {{a_null}},
    "key3": {{a_number}},
    "key4: "{{newDate}}"
}
```
```

Variables can be initialized via command line:

```
$ hurl --variable a_string=apple --variable a_bool=true --variable a_null=null --
```

Resulting in a PUT request with the following JSON body:

```
{
    "key0": "apple",
    "key1": true,
    "key2": null,
    "key3": 42,
    "key4": "2024-12-02T13:39:45.936643Z"
}
```

Grammar

Definitions

Short description:

- operator | denotes alternative,
- operator * denotes iteration (zero or more),
- operator + denotes iteration (one or more),

Syntax Grammar

General

hurl-file
  entry*
  lt*

entry(used by hurl-file)
  request
  response?

request(used by entry)
  lt*
  method sp value-string lt
  header*
  request-section*
  body?

response(used by entry)
  lt*
  version sp status lt
  header*
  response-section*
  body?

method(used by request)
  [A-Z]+

version(used by response)
   HTTP/1.0
  |HTTP/1.1
  |HTTP/2
  |HTTP

status(used by response)
  [0-9]+

header(used by request, response)
  lt*
  key-value lt

body(used by request, response)
  lt*
  bytes lt

Sections

request-section(used by request)
   basic-auth-section
  |query-string-params-section
  |form-params-section
  |multipart-form-data-section
  |cookies-section
  |options-section

response-section(used by [response](#))
  [captures-section](#)
  |[asserts-section](#)

query-string-params-section(used by [request-section](#))
  [lt](#)*
  ([QueryStringParams]|[Query]) [lt](#)
  [key-value](#)*

form-params-section(used by [request-section](#))
  [lt](#)*
  ([FormParams]|[Form]) [lt](#)
  [key-value](#)*

multipart-form-data-section(used by [request-section](#))
  [lt](#)*
  ([MultipartFormData]|[Multipart]) [lt](#)
  [multipart-form-data-param](#)*

cookies-section(used by [request-section](#))
  [lt](#)*
  [Cookies] [lt](#)
  [key-value](#)*

captures-section(used by [response-section](#))
  [lt](#)*
  [Captures] [lt](#)
  [capture](#)*

asserts-section(used by [response-section](#))
  [lt](#)*
  [Asserts] [lt](#)
  [assert](#)*

basic-auth-section(used by [request-section](#))
  [lt](#)*
  [BasicAuth] [lt](#)
  [key-value](#)*

options-section(used by [request-section](#))
  [lt](#)*
  [Options] [lt](#)
  [option](#)*

key-value(used by [header](#), [query-string-params-section](#), [form-params-section](#), [cookies-section](#), [basic-auth-section](#), [multipart-form-data-param](#))
  [key-string](#) : [value-string](#)

multipart-form-data-param(used by [multipart-form-data-section](#))
  [file-param](#)|[key-value](#)

file-param(used by [multipart-form-data-param](#))
  [lt](#)*
  [key-string](#) : [file-value](#) [lt](#)

file-value(used by [file-param](#))
  file, [filename](#) ; ([file-contenttype](#))?

file-contenttype(used by [file-value](#))
  [value-string](#)

capture(used by [captures-section](#))
  [lt](#)*
  [key-string](#) : [query](#) ([sp](#) [filter](#))* ([sp](#) redact)? [lt](#)

assert(used by [asserts-section](#))
  [lt](#)*
  [query](#) ([sp](#) [filter](#))* [sp](#) [predicate](#) [lt](#)

option(used by options-section)
  lt*
  (aws-sigv4-option|ca-certificate-option|client-certificate-
  option|client-key-option|compressed-option|connect-to-option|connect-
  timeout-option|delay-option|follow-redirect-option|follow-redirect-
  trusted-option|header-option|http10-option|http11-option|http2-
  option|http3-option|insecure-option|ipv4-option|ipv6-option|limit-rate-
  option|max-redirs-option|netrc-option|netrc-file-option|netrc-optional-
  option|output-option|path-as-is-option|proxy-option|repeat-
  option|resolve-option|retry-option|retry-interval-option|skip-
  option|unix-socket-option|user-option|variable-option|verbose-
  option|very-verbose-option)

aws-sigv4-option(used by option)
  aws-sigv4 : value-string lt

ca-certificate-option(used by option)
  cacert : filename lt

client-certificate-option(used by option)
  cert : filename-password lt

client-key-option(used by option)
  key : value-string lt

compressed-option(used by option)
  compressed : boolean-option lt

connect-to-option(used by option)
  connect-to : value-string lt

connect-timeout-option(used by option)
  connect-timeout : duration-option lt

delay-option(used by option)
  delay : duration-option lt

follow-redirect-option(used by option)
  location : boolean-option lt

follow-redirect-trusted-option(used by option)
  location-trusted : boolean-option lt

header-option(used by option)
  header : value-string lt

http10-option(used by option)
  http1.0 : boolean-option lt

http11-option(used by option)
  http1.1 : boolean-option lt

http2-option(used by option)
  http2 : boolean-option lt

http3-option(used by option)
  http3 : boolean-option lt

insecure-option(used by option)
  insecure : boolean-option lt

ipv4-option(used by option)
  ipv4 : boolean-option lt

ipv6-option(used by option)
  ipv6 : boolean-option lt

limit-rate-option(used by option)

```
          limit-rate : integer-option lt

max-redirs-option(used by option)
  max-redirs : integer-option lt

netrc-option(used by option)
  netrc : boolean-option lt

netrc-file-option(used by option)
  netrc-file : value-string lt

netrc-optional-option(used by option)
  netrc-optional : boolean-option lt

output-option(used by option)
  output : value-string lt

path-as-is-option(used by option)
  path-as-is : boolean-option lt

proxy-option(used by option)
  proxy : value-string lt

resolve-option(used by option)
  resolve : value-string lt

repeat-option(used by option)
  repeat : integer-option lt

retry-option(used by option)
  retry : integer-option lt

retry-interval-option(used by option)
  retry-interval : duration-option lt

skip-option(used by option)
  skip : boolean-option lt

unix-socket-option(used by option)
  unix-socket : value-string lt

user-option(used by option)
  user : value-string lt

variable-option(used by option)
  variable : variable-definition lt

verbose-option(used by option)
  verbose : boolean-option lt

very-verbose-option(used by option)
  very-verbose : boolean-option lt

variable-definition(used by variable-option)
  variable-name = variable-value

boolean-option(used by compressed-option, follow-redirect-option, follow-
redirect-trusted-option, http10-option, http11-option, http2-option, http3-
option, insecure-option, ipv4-option, ipv6-option, netrc-option, netrc-optional-
option, path-as-is-option, skip-option, verbose-option, very-verbose-option)
  boolean|placeholder

integer-option(used by limit-rate-option, max-redirs-option, repeat-
option, retry-option)
  integer|placeholder

duration-option(used by connect-timeout-option, delay-option, retry-interval-
option)
  (integer duration-unit?)|placeholder
```

duration-unit(used by [duration-option](#))
  ms|s|m

variable-value(used by [variable-definition](#))
   [null](#)
  |[boolean](#)
  |[integer](#)
  |[float](#)
  |[key-string](#)
  |[quoted-string](#)

Query

query(used by [capture](#), [assert](#))
   [status-query](#)
  |[version-query](#)
  |[url-query](#)
  |[ip-query](#)
  |[header-query](#)
  |[certificate-query](#)
  |[cookie-query](#)
  |[body-query](#)
  |[xpath-query](#)
  |[jsonpath-query](#)
  |[regex-query](#)
  |[variable-query](#)
  |[duration-query](#)
  |[bytes-query](#)
  |[sha256-query](#)
  |[md5-query](#)

status-query(used by [query](#))
  status

version-query(used by [query](#))
  version

url-query(used by [query](#))
  url

ip-query(used by [query](#))
  ip

header-query(used by [query](#))
  header [sp](#) [quoted-string](#)

certificate-query(used by [query](#))
  certificate [sp](#) (Subject|Issuer|Start-Date|Expire-Date|Serial-Number)

cookie-query(used by [query](#))
  cookie [sp](#) [quoted-string](#)

body-query(used by [query](#))
  body

xpath-query(used by [query](#))
  xpath [sp](#) [quoted-string](#)

jsonpath-query(used by [query](#))
  jsonpath [sp](#) [quoted-string](#)

regex-query(used by [query](#))
  regex [sp](#) ([quoted-string](#)|[regex](#))

variable-query(used by [query](#))
  variable [sp](#) [quoted-string](#)

duration-query(used by [query](#))
  duration

sha256-query(used by [query](#))
  sha256

md5-query(used by [query](#))
  md5

bytes-query(used by [query](#))
  bytes

Predicates

predicate(used by [assert](#))
  (not [sp](#))? [predicate-func](#)

predicate-func(used by [predicate](#))
   [equal-predicate](#)
  |[not-equal-predicate](#)
  |[greater-predicate](#)
  |[greater-or-equal-predicate](#)
  |[less-predicate](#)
  |[less-or-equal-predicate](#)
  |[start-with-predicate](#)
  |[end-with-predicate](#)
  |[contain-predicate](#)
  |[match-predicate](#)
  |[exist-predicate](#)
  |[is-empty-predicate](#)
  |[include-predicate](#)
  |[integer-predicate](#)
  |[float-predicate](#)
  |[boolean-predicate](#)
  |[string-predicate](#)
  |[collection-predicate](#)
  |[date-predicate](#)
  |[iso-date-predicate](#)
  |[is-ipv4-predicate](#)
  |[is-ipv6-predicate](#)

equal-predicate(used by [predicate-func](#))
  == [sp](#) [predicate-value](#)

not-equal-predicate(used by [predicate-func](#))
  != [sp](#) [predicate-value](#)

greater-predicate(used by [predicate-func](#))
  > [sp](#) ([number](#)|[quoted-string](#))

greater-or-equal-predicate(used by [predicate-func](#))
  >= [sp](#) [sp](#)* ([number](#)|[quoted-string](#))

less-predicate(used by [predicate-func](#))
  < [sp](#) ([number](#)|[quoted-string](#))

less-or-equal-predicate(used by [predicate-func](#))
  <= [sp](#) ([number](#)|[quoted-string](#))

start-with-predicate(used by [predicate-func](#))
  startsWith [sp](#) ([quoted-string](#)|[oneline-hex](#)|[oneline-base64](#))

end-with-predicate(used by [predicate-func](#))
  endsWith [sp](#) ([quoted-string](#)|[oneline-hex](#)|[oneline-base64](#))

contain-predicate(used by [predicate-func](#))
  contains [sp](#) [quoted-string](#)

match-predicate(used by [predicate-func](#))
  matches [sp](#) ([quoted-string](#)|[regex](#))

exist-predicate(used by [predicate-func](#))
  exists

is-empty-predicate(used by [predicate-func](#))
  isEmpty

include-predicate(used by [predicate-func](#))
  includes [sp](#) [predicate-value](#)

integer-predicate(used by [predicate-func](#))
  isInteger

float-predicate(used by [predicate-func](#))
  isFloat

boolean-predicate(used by [predicate-func](#))
  isBoolean

string-predicate(used by [predicate-func](#))
  isString

collection-predicate(used by [predicate-func](#))
  isCollection

date-predicate(used by [predicate-func](#))
  isDate

iso-date-predicate(used by [predicate-func](#))
  isIsoDate

is-ipv4-predicate(used by [predicate-func](#))
  isIpv4

is-ipv6-predicate(used by [predicate-func](#))
  isIpv6

predicate-value(used by [equal-predicate](#), [not-equal-predicate](#), [include-predicate](#))
   [boolean](#)
  |[multiline-string](#)
  |[null](#)
  |[number](#)
  |[oneline-string](#)
  |[oneline-base64](#)
  |[oneline-file](#)
  |[oneline-hex](#)
  |[quoted-string](#)
  |[placeholder](#)

Bytes

bytes(used by [body](#))
   [json-value](#)
  |[xml](#)
  |[multiline-string](#)
  |[oneline-string](#)
  |[oneline-base64](#)
  |[oneline-file](#)
  |[oneline-hex](#)

xml(used by [bytes](#))

< To Be Defined >

oneline-base64(used by <u>start-with-predicate</u>, <u>end-with-predicate</u>, <u>predicate-value</u>, <u>bytes</u>)
  base64, [A-Z0-9+-= \n]+ ;

oneline-file(used by <u>predicate-value</u>, <u>bytes</u>)
  file, <u>filename</u> ;

oneline-hex(used by <u>start-with-predicate</u>, <u>end-with-predicate</u>, <u>predicate-value</u>, <u>bytes</u>)
  hex, <u>hexdigit</u>* ;

Strings

quoted-string(used by <u>variable-value</u>, <u>header-query</u>, <u>cookie-query</u>, <u>xpath-query</u>, <u>jsonpath-query</u>, <u>regex-query</u>, <u>variable-query</u>, <u>greater-predicate</u>, <u>greater-or-equal-predicate</u>, <u>less-predicate</u>, <u>less-or-equal-predicate</u>, <u>start-with-predicate</u>, <u>end-with-predicate</u>, <u>contain-predicate</u>, <u>match-predicate</u>, <u>predicate-value</u>, <u>jsonpath-filter</u>, <u>regex-filter</u>, <u>replace-filter</u>, <u>split-filter</u>, <u>xpath-filter</u>)
  " (<u>quoted-string-content</u>|<u>placeholder</u>)* **"**

quoted-string-content(used by <u>quoted-string</u>)
  (<u>quoted-string-text</u>|<u>quoted-string-escaped-char</u>)*

quoted-string-text(used by <u>quoted-string-content</u>)
  ~["\\]+

quoted-string-escaped-char(used by <u>quoted-string-content</u>)
  \ ("|\|\b|\f|\n|\r|\t|\u <u>unicode-char</u>)

key-string(used by <u>key-value</u>, <u>file-param</u>, <u>capture</u>, <u>variable-value</u>)
  (<u>key-string-content</u>|<u>placeholder</u>)+

key-string-content(used by <u>key-string</u>)
  (<u>key-string-text</u>|<u>key-string-escaped-char</u>)*

key-string-text(used by <u>key-string-content</u>)
  (<u>alphanum</u>|_|-|.|[|]|@|$)+

key-string-escaped-char(used by <u>key-string-content</u>)
  \ (#|:|\|\b|\f|\n|\r|\t|\u <u>unicode-char</u>)

value-string(used by <u>request</u>, <u>key-value</u>, <u>file-contenttype</u>, <u>aws-sigv4-option</u>, <u>client-key-option</u>, <u>connect-to-option</u>, <u>header-option</u>, <u>netrc-file-option</u>, <u>output-option</u>, <u>proxy-option</u>, <u>resolve-option</u>, <u>unix-socket-option</u>, <u>user-option</u>)
  (<u>value-string-content</u>|<u>placeholder</u>)*

value-string-content(used by <u>value-string</u>)
  (<u>value-string-text</u>|<u>value-string-escaped-char</u>)*

value-string-text(used by <u>value-string-content</u>)
  ~[#\n\\]+

value-string-escaped-char(used by <u>value-string-content</u>)
  \ (#|\|\b|\f|\n|\r|\t|\u <u>unicode-char</u>)

oneline-string(used by <u>predicate-value</u>, <u>bytes</u>)
  ` (<u>oneline-string-content</u>|<u>placeholder</u>)* `

oneline-string-content(used by <u>oneline-string</u>)
  (<u>oneline-string-text</u>|<u>oneline-string-escaped-char</u>)*

oneline-string-text(used by <u>oneline-string-content</u>)
  ~[#\n\\] ~`

oneline-string-escaped-char(used by oneline-string-content)
  \ (`|#|\|b|f|u unicode-char)

multiline-string(used by predicate-value, bytes)
  ``` multiline-string-type? (, multiline-string-attribute)* lt
  (multiline-string-content|placeholder)* lt
  ```

multiline-string-type(used by multiline-string)
   base64
   |hex
   |json
   |xml
   |graphql

multiline-string-attribute(used by multiline-string)
   escape
   |novariable

multiline-string-content(used by multiline-string)
  (multiline-string-text|multiline-string-escaped-char)*

multiline-string-text(used by multiline-string-content)
  ~[\\]+ ~```

multiline-string-escaped-char(used by multiline-string-content)
  \ (\|b|f|n|r|t|`|u unicode-char)

filename(used by file-value, ca-certificate-option, oneline-file)
  (filename-content|placeholder)*

filename-content(used by filename)
  (filename-text|filename-escaped-char)*

filename-text(used by filename-content)
  ~[#;{} \n\\]+

filename-escaped-char(used by filename-content)
  \ (\|b|f|n|r|t|#|;| |{|}|u unicode-char)

filename-password(used by client-certificate-option)
  (filename-password-content|placeholder)*

filename-password-content(used by filename-password)
  (filename-password-text|filename-password-escaped-char)*

filename-password-text(used by filename-password-content)
  ~[#;{} \n\\]+

filename-password-escaped-char(used by filename-password-content)
  \ (\|b|f|n|r|t|#|;| |{|}|:|u unicode-char)

unicode-char(used by quoted-string-escaped-char, key-string-escaped-char, value-string-escaped-char, oneline-string-escaped-char, multiline-string-escaped-char, filename-escaped-char, filename-password-escaped-char)
  { hexdigit+ }

JSON

json-value(used by bytes, json-key-value, json-array)
   placeholder
   |json-object
   |json-array
   |json-string
   |json-number
   |boolean

```
                    |null

json-object(used by json-value)
  { json-key-value (, json-key-value)* }

json-key-value(used by json-object)
  json-string : json-value

json-array(used by json-value)
  [ json-value (, json-value)* ]

json-string(used by json-value, json-key-value)
  " (json-string-content|placeholder)* "

json-string-content(used by json-string)
  json-string-text|json-string-escaped-char

json-string-text(used by json-string-content)
  ~["\\]

json-string-escaped-char(used by json-string-content)
  \ ("|\|b|f|n|r|t|u hexdigit hexdigit hexdigit hexdigit)

json-number(used by json-value)
  [-]? json-integer fraction? exponent?

json-integer(used by json-number)
  0|[1-9] digit*
```

Expression

```
placeholder(used by boolean-option, integer-option, duration-option, predicate-
value, quoted-string, key-string, value-string, oneline-string, multiline-
string, filename, filename-password, json-value, json-string)
  {{ expr }}

expr(used by placeholder)
  (variable-name|function) (sp filter)*

variable-name(used by variable-definition, expr)
  [A-Za-z] [A-Za-z_-0-9]*
```

Function

```
function(used by expr)
   env-function
  |now-function
  |uuid-function

env-function(used by function)
  getEnv

now-function(used by function)
  newDate

uuid-function(used by function)
  newUuid
```

Filter

```
filter(used by capture, assert, expr)
   base64-decode-filter
```

|base64-encode-filter
         |count-filter
         |days-after-now-filter
         |days-before-now-filter
         |decode-filter
         |format-filter
         |html-escape-filter
         |html-unescape-filter
         |jsonpath-filter
         |nth-filter
         |regex-filter
         |replace-filter
         |split-filter
         |to-date-filter
         |to-float-filter
         |to-int-filter
         |to-string-filter
         |url-decode-filter
         |url-encode-filter
         |xpath-filter

base64-decode-filter(used by filter)
  base64Decode

base64-encode-filter(used by filter)
  base64Encode

count-filter(used by filter)
  count

days-after-now-filter(used by filter)
  daysAfterNow

days-before-now-filter(used by filter)
  daysBeforeNow

decode-filter(used by filter)
  decode

format-filter(used by filter)
  format

html-escape-filter(used by filter)
  htmlEscape

html-unescape-filter(used by filter)
  htmlUnescape

jsonpath-filter(used by filter)
  jsonpath sp quoted-string

nth-filter(used by filter)
  nth sp integer

regex-filter(used by filter)
  regex sp (quoted-string|regex)

replace-filter(used by filter)
  replace sp (quoted-string|regex) sp quoted-string

split-filter(used by filter)
  split sp quoted-string

to-date-filter(used by filter)
  toDate

to-float-filter(used by filter)
  toFloat

to-int-filter(used by [filter])
  toInt

to-string-filter(used by [filter])
  toString

url-decode-filter(used by [filter])
  urlDecode

url-encode-filter(used by [filter])
  urlEncode

xpath-filter(used by [filter])
  xpath [sp] [quoted-string]

Lexical Grammar

boolean(used by [boolean-option], [variable-value], [predicate-value], [json-value])
  true|false

null(used by [variable-value], [predicate-value], [json-value])
  null

alphanum(used by [key-string-text])
  [A-Za-z0-9]

integer(used by [integer-option], [duration-option], [variable-value], [nth-filter], [float], [number])
  [digit]+

float(used by [variable-value], [number])
  [integer] [fraction]

number(used by [greater-predicate], [greater-or-equal-predicate], [less-predicate], [less-or-equal-predicate], [predicate-value])
  [integer]|[float]

digit(used by [json-integer], [integer], [fraction], [exponent])
  [0-9]

hexdigit(used by [oneline-hex], [unicode-char], [json-string-escaped-char])
  [0-9A-Fa-f]

fraction(used by [json-number], [float])
  . [digit]+

exponent(used by [json-number])
  (e|E) (+|-)? [digit]+

sp(used by [request], [response], [capture], [assert], [header-query], [certificate-query], [cookie-query], [xpath-query], [jsonpath-query], [regex-query], [variable-query], [predicate], [equal-predicate], [not-equal-predicate], [greater-predicate], [greater-or-equal-predicate], [less-predicate], [less-or-equal-predicate], [start-with-predicate], [end-with-predicate], [contain-predicate], [match-predicate], [include-predicate], [expr], [jsonpath-filter], [nth-filter], [regex-filter], [replace-filter], [split-filter], [xpath-filter], [lt])
  [\t]

lt(used by [hurl-file], [request], [response], [header], [body], [query-string-params-section], [form-params-section], [multipart-form-data-section], [cookies-section], [captures-section], [asserts-section], [basic-auth-section], [options-section], [file-param], [capture], [assert], [option], [aws-sigv4-option], [ca-certificate-option], [client-certificate-option], [client-key-option], [compressed-option], [connect-to-option], [connect-timeout-option], [delay-option], [follow-redirect-option], [follow-redirect-trusted-option], [header-option], [http10-option], [http11-option], [http2-option], [http3-option], [insecure-option], [ipv4-option], [ipv6-option], [limit-rate-

option, max-redirs-option, netrc-option, netrc-file-option, netrc-optional-option, output-option, path-as-is-option, proxy-option, resolve-option, repeat-option, retry-option, retry-interval-option, skip-option, unix-socket-option, user-option, variable-option, verbose-option, very-verbose-option, multiline-string)
  sp* comment? [\n]?

comment(used by lt)
  # ~[\n]*

regex(used by regex-query, match-predicate, regex-filter, replace-filter)
  / regex-content /

regex-content(used by regex)
  (regex-text|regex-escaped-char)*

regex-text(used by regex-content)
  ~[\n\/]+

regex-escaped-char(used by regex-content)
  \ ~[\n]

Resources

License

```
                         Apache License
                   Version 2.0, January 2004
                http://www.apache.org/licenses/

   TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

   1. Definitions.

      "License" shall mean the terms and conditions for use, reproduction,
      and distribution as defined by Sections 1 through 9 of this document.

      "Licensor" shall mean the copyright owner or entity authorized by
      the copyright owner that is granting the License.

      "Legal Entity" shall mean the union of the acting entity and all
      other entities that control, are controlled by, or are under common
      control with that entity. For the purposes of this definition,
      "control" means (i) the power, direct or indirect, to cause the
      direction or management of such entity, whether by contract or
      otherwise, or (ii) ownership of fifty percent (50%) or more of the
      outstanding shares, or (iii) beneficial ownership of such entity.

      "You" (or "Your") shall mean an individual or Legal Entity
      exercising permissions granted by this License.

      "Source" form shall mean the preferred form for making modifications,
      including but not limited to software source code, documentation
      source, and configuration files.

      "Object" form shall mean any form resulting from mechanical
      transformation or translation of a Source form, including but
      not limited to compiled object code, generated documentation,
      and conversions to other media types.

      "Work" shall mean the work of authorship, whether in Source or
      Object form, made available under the License, as indicated by a
      copyright notice that is included in or attached to the work
      (an example is provided in the Appendix below).

      "Derivative Works" shall mean any work, whether in Source or Object
      form, that is based on (or derived from) the Work and for which the
      editorial revisions, annotations, elaborations, or other modifications
      represent, as a whole, an original work of authorship. For the purposes
      of this License, Derivative Works shall not include works that remain
      separable from, or merely link (or bind by name) to the interfaces of,
      the Work and Derivative Works thereof.

      "Contribution" shall mean any work of authorship, including
      the original version of the Work and any modifications or additions
      to that Work or Derivative Works thereof, that is intentionally
      submitted to Licensor for inclusion in the Work by the copyright owner
      or by an individual or Legal Entity authorized to submit on behalf of
      the copyright owner. For the purposes of this definition, "submitted"
      means any form of electronic, verbal, or written communication sent
      to the Licensor or its representatives, including but not limited to
      communication on electronic mailing lists, source code control systems,
      and issue tracking systems that are managed by, or on behalf of, the
      Licensor for the purpose of discussing and improving the Work, but
      excluding communication that is conspicuously marked or otherwise
      designated in writing by the copyright owner as "Not a Contribution."

      "Contributor" shall mean Licensor and any individual or Legal Entity
      on behalf of whom a Contribution has been received by Licensor and
      subsequently incorporated within the Work.
```

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
   Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
   stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
   that You distribute, all copyright, patent, trademark, and
   attribution notices from the Source form of the Work,
   excluding those notices that do not pertain to any part of
   the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
   distribution, then any Derivative Works that You distribute must
   include a readable copy of the attribution notices contained
   within such NOTICE file, excluding those notices that do not
   pertain to any part of the Derivative Works, in at least one
   of the following places: within a NOTICE text file distributed
   as part of the Derivative Works; within the Source form or
   documentation, if provided along with the Derivative Works; or,
   within a display generated by the Derivative Works, if and
   wherever such third-party notices normally appear. The contents
   of the NOTICE file are for informational purposes only and
   do not modify the License. You may add Your own attribution
   notices within Derivative Works that You distribute, alongside
   or as an addendum to the NOTICE text from the Work, provided
   that such additional attribution notices cannot be construed
   as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,

except as required for reasonable and customary use in describing the
origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

    To apply the Apache License to your work, attach the following
    boilerplate notice, with the fields enclosed by brackets "[]"
    replaced with your own identifying information. (Don't include
    the brackets!) The text should be enclosed in the appropriate
    comment syntax for the file format. We also recommend that a
    file or class name and description of purpose be included on the
    same "printed page" as the copyright notice for easier
    identification within third-party archives.

Copyright 2021 Hurl

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

     http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.