Splash Documentation

Release 2.3.3

Scrapinghub

Jun 07, 2017

Contents

Splash is a javascript rendering service. It's a lightweight web browser with an HTTP API, implemented in Python using Twisted and QT. The (twisted) QT reactor is used to make the service fully asynchronous allowing to take advantage of webkit concurrency via QT main loop. Some of Splash features:

- process multiple webpages in parallel;
- get HTML results and/or take screenshots;
- turn OFF images or use Adblock Plus rules to make rendering faster;
- execute custom JavaScript in page context;
- write Lua browsing *scripts*;
- develop Splash Lua scripts in *Splash-Jupyter* Notebooks.
- get detailed rendering info in HAR format.

CHAPTER 1

Documentation

Installation

Linux + Docker

1. Install Docker.
2. Pull the image:

```
$ sudo docker pull scrapinghub/splash
```

3. Start the container:

```
$ sudo docker run -p 5023:5023 -p 8050:8050 -p 8051:8051 scrapinghub/splash
```

4. Splash is now available at 0.0.0.0 at ports 8050 (http), 8051 (https) and 5023 (telnet).

OS X + Docker

1. Install Docker (via the Toolbox Instructions).
2. Create, run & load the configuration for the docker-machine

   $ docker-machine create default

   $ docker-machine start default

   $ eval “$(docker-machine env default)”

2. Pull the image:

```
$ docker pull scrapinghub/splash
```

3. Start the container:

```
$ docker run -p 5023:5023 -p 8050:8050 -p 8051:8051 scrapinghub/splash
```

4. Figure out the ip address of the docker-machine:

```
$ docker-machine ip default

192.168.59.103
```

5. Splash is available at the returned IP address at ports 8050 (http), 8051 (https) and 5023 (telnet).

Ubuntu 14.04 (manual way)

1. Clone the repo from GitHub:

```
$ git clone https://github.com/scrapinghub/splash/
```

2. Install dependencies:

```
$ cd splash/dockerfiles/splash

$ sudo ./provision.sh prepare_install install_msfonts \
                      install_builddeps install_deps \
                      install_extra_fonts install_pyqt5 \
                      install_python_deps install_flash

Change back to the parent directory of splash, i.e. `cd ~`

$ sudo pip3 install splash/
```

To run the server execute the following command:

```
python3 -m splash.server
```

Run `python -m splash.server --help` to see options available.

By default, Splash API endpoints listen to port 8050 on all available IPv4 addresses. To change the port use `--port` option:

```
python3 -m splash.server --port=5000
```

Required Python packages

```
# install PyQt5 (Splash is tested on PyQT 5.5.1)
# and the following packages:
twisted >= 15.5.0, < 16.3.0
qt5reactor
psutil
adblockparser >= 0.5
https://github.com/sunu/pyre2/archive/c610be52c3b5379b257d56fc0669d022fd70082a.zip
↪#egg=re2
xvfbwrapper
Pillow
six

# for scripting support
```

```
lupa >= 1.3
funcparserlib >= 0.3.6
```

Splash Versions

`docker pull scrapinghub/splash` will give you the latest stable Splash release. To obtain the latest development version use `docker pull scrapinghub/splash:master`. Specific Splash versions are also available, e.g. `docker pull scrapinghub/splash:1.8`.

Customizing Dockerized Splash

Passing Custom Options

To run Splash with custom options pass them to `docker run`. For example, let's increase log verbosity:

```
$ docker run -p 8050:8050 scrapinghub/splash -v3
```

To see all possible options pass `--help`. Not all options will work the same inside Docker: changing ports doesn't make sense (use docker run options instead), and paths are paths in the container.

Folders Sharing

To set custom *Request Filters* use -v Docker option. First, create a folder with request filters on your local filesystem, then make it available to the container:

```
$ docker run -p 8050:8050 -v <my-filters-dir>:/etc/splash/filters scrapinghub/splash
```

Replace `<my-filters-dir>` with a path of your local folder with request filters.

Docker Data Volume Containers can also be used. Check https://docs.docker.com/userguide/dockervolumes/ for more info.

Proxy Profiles and *Javascript Profiles* can be added in a similar way:

```
$ docker run -p 8050:8050 \
      -v <my-proxy-profiles-dir>:/etc/splash/proxy-profiles \
      -v <my-js-profiles-dir>:/etc/splash/js-profiles \
      scrapinghub/splash
```

To setup *Adding Your Own Modules* mount a folder to `/etc/splash/lua_modules`. If you use a *Lua sandbox* (default) don't forget to list safe modules using `--lua-sandbox-allowed-modules` option:

```
$ docker run -p 8050:8050 \
      -v <my-lua-modules-dir>:/etc/splash/lua_modules \
      --lua-sandbox-allowed-modules 'module1;module2' \
      scrapinghub/splash
```

Warning: Folder sharing (`-v` option) doesn't work on OS X and Windows (see https://github.com/docker/docker/issues/4023). It should be fixed in future Docker & Boot2Docker releases. For now use one of the workarounds mentioned in issue comments or clone Splash repo and customize its Dockerfile.

Building Local Docker Images

To build your own Docker image, checkout Splash source code using git, then execute the following command from Splash source root:

```
$ docker build -t my-local-splash .
```

To build *Splash-Jupyter* Docker image use this command:

```
$ docker build -t my-local-splash-jupyter -f  dockerfiles/splash-jupyter/Dockerfile .
```

You may have to change FROM line in `dockerfiles/splash-jupyter/Dockerfile` if you want it to be based on your local Splash Docker container.

Splash HTTP API

Consult with *Installation* to get Splash up and running.

Splash is controlled via HTTP API. For all endpoints below parameters may be sent either as GET arguments or encoded to JSON and POSTed with `Content-Type:  application/json` header.

The most versatile endpoint that provides all Splash features is *execute*; it allows to execute arbitrary Lua rendering scripts.

Other endpoints may be easier to use in specific cases - for example, *render.png* returns a screenshot in PNG format that can be used as *img src* without any further processing, and *render.json* is convenient if you don't need to interact with a page.

render.html

Return the HTML of the javascript-rendered page.

Arguments:

url [string][required] The url to render (required)

baseurl [string][optional] The base url to render the page with.

Base HTML content will be feched from the URL given in the url argument, while relative referenced resources in the HTML-text used to render the page are fetched using the URL given in the baseurl argument as base. See also: *render.html result looks broken in a browser*.

timeout [float][optional] A timeout (in seconds) for the render (defaults to 30).

By default, maximum allowed value for the timeout is 60 seconds. To override it start Splash with `--max-timeout` command line option. For example, here Splash is configured to allow timeouts up to 2 minutes:

```
$ python -m splash.server --max-timeout 120
```

resource_timeout [float][optional] A timeout (in seconds) for individual network requests.

See also: *splash:on_request* and its `request:set_timeout(timeout)` method; *splash.resource_timeout* attribute.

wait [float][optional] Time (in seconds) to wait for updates after page is loaded (defaults to 0). Increase this value if you expect pages to contain setInterval/setTimeout javascript calls, because with wait=0 callbacks of setInterval/setTimeout won't be executed. Non-zero *wait* is also required for PNG and JPEG rendering when doing full-page rendering (see *render_all*). Maximum allowed value for wait is 30 seconds.

proxy [string][optional] Proxy profile name or proxy URL. See *Proxy Profiles*.

A proxy URL should have the following format: `[protocol://][user:password@]proxyhost[:port])`

Where protocol is either `http` or `socks5`. If port is not specified, the port 1080 is used by default.

js [string][optional] Javascript profile name. See *Javascript Profiles*.

js_source [string][optional] JavaScript code to be executed in page context. See *Executing custom Javascript code within page context*.

filters [string][optional] Comma-separated list of request filter names. See *Request Filters*

allowed_domains [string][optional] Comma-separated list of allowed domain names. If present, Splash won't load anything neither from domains not in this list nor from subdomains of domains not in this list.

allowed_content_types [string][optional] Comma-separated list of allowed content types. If present, Splash will abort any request if the response's content type doesn't match any of the content types in this list. Wildcards are supported using the fnmatch syntax.

forbidden_content_types [string][optional] Comma-separated list of forbidden content types. If present, Splash will abort any request if the response's content type matches any of the content types in this list. Wildcards are supported using the fnmatch syntax.

viewport [string][optional] View width and height (in pixels) of the browser viewport to render the web page. Format is "<width>x<height>", e.g. 800x600. Default value is 1024x768.

'viewport' parameter is more important for PNG and JPEG rendering; it is supported for all rendering endpoints because javascript code execution can depend on viewport size.

For backward compatibility reasons, it also accepts 'full' as value; `viewport=full` is semantically equivalent to `render_all=1` (see *render_all*).

images [integer][optional] Whether to download images. Possible values are `1` (download images) and `0` (don't download images). Default is 1.

Note that cached images may be displayed even if this parameter is 0. You can also use *Request Filters* to strip unwanted contents based on URL.

headers [JSON array or object][optional] HTTP headers to set for the first outgoing request.

This option is only supported for `application/json` POST requests. Value could be either a JSON array with `(header_name, header_value)` pairs or a JSON object with header names as keys and header values as values.

"User-Agent" header is special: is is used for all outgoing requests, unlike other headers.

body [string][optional] Body of HTTP POST request to be sent if method is POST. Default `content-type` header for POST requests is `application/x-www-form-urlencoded`.

http_method [string][optional] HTTP method of outgoing Splash request. Default method is GET. Splash also supports POST.

save_args [JSON array or a comma-separated string][optional] A list of argument names to put in cache. Splash will store each argument value in an internal cache and return `X-Splash-Saved-Arguments` HTTP header with a list of SHA1 hashes for each argument (a semicolon-separated list of name=hash pairs):

```
name1=9a6747fc6259aa374ab4e1bb03074b6ec672cf99;
↪name2=ba001160ef96fe2a3f938fea9e6762e204a562b3
```

Client can then use *load_args* parameter to pass these hashes instead of argument values. This is most useful when argument value is large and doesn't change often (*js_source* or *lua_source* are often good candidates).

load_args [JSON object or a string][optional] Parameter values to load from cache. `load_args` should be either `{"name": "<SHA1 hash>", ...}` JSON object or a raw `X-Splash-Saved-Arguments` header value (a semicolon-separated list of name=hash pairs).

For each parameter in `load_args` Splash tries to fetch the value from the internal cache using a provided SHA1 hash as a key. If all values are in cache then Splash uses them as argument values and then handles the request as usual.

If at least on argument can't be found Splash returns **HTTP 498** status code. In this case client should repeat the request, but use *save_args* and send full argument values.

load_args and *save_args* allow to save network traffic by not sending large arguments with each request (*js_source* and *lua_source* are often good candidates).

Splash uses LRU cache to store values; the number of entries is limited, and cache is cleared after each Splash restart. In other words, storage is not persistent; client should be ready to re-send the arguments.

Examples

Curl example:

```
curl 'http://localhost:8050/render.html?url=http://domain.com/page-with-javascript.
↪html&timeout=10&wait=0.5'
```

The result is always encoded to utf-8. Always decode HTML data returned by render.html endpoint from utf-8 even if there are tags like

```
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
```

in the result.

render.png

Return a image (in PNG format) of the javascript-rendered page.

Arguments:

Same as *render.html* plus the following ones:

width [integer][optional] Resize the rendered image to the given width (in pixels) keeping the aspect ratio.

height [integer][optional] Crop the renderd image to the given height (in pixels). Often used in conjunction with the width argument to generate fixed-size thumbnails.

render_all [int][optional] Possible values are `1` and `0`. When `render_all=1`, extend the viewport to include the whole webpage (possibly very tall) before rendering. Default is `render_all=0`.

Note: `render_all=1` requires non-zero *wait* parameter. This is an unfortunate restriction, but it seems that this is the only way to make rendering work reliably with `render_all=1`.

scale_method [string][optional] Possible values are `raster` (default) and `vector`. If `scale_method=raster`, rescaling operation performed via *width* parameter is pixel-wise. If `scale_method=vector`, rescaling is done element-wise during rendering.

> **Note:** Vector-based rescaling is more performant and results in crisper fonts and sharper element boundaries, however there may be rendering issues, so use it with caution.

Examples

Curl examples:

```
# render with timeout
curl 'http://localhost:8050/render.png?url=http://domain.com/page-with-javascript.
↪html&timeout=10'

# 320x240 thumbnail
curl 'http://localhost:8050/render.png?url=http://domain.com/page-with-javascript.
↪html&width=320&height=240'
```

render.jpeg

Return a image (in JPEG format) of the javascript-rendered page.

Arguments:

Same as *render.png* plus the following ones:

quality [integer][optional] JPEG quality parameter in range from `0` to `100`. Default is `quality=75`.

> **Note:** `quality` values above `95` should be avoided; `quality=100` disables portions of the JPEG compression algorithm, and results in large files with hardly any gain in image quality.

Examples

Curl examples:

```
# render with default quality
curl 'http://localhost:8050/render.jpeg?url=http://domain.com/'

# render with low quality
curl 'http://localhost:8050/render.jpeg?url=http://domain.com/&quality=30'
```

render.har

Return information about Splash interaction with a website in HAR format. It includes information about requests made, responses received, timings, headers, etc.

You can use online HAR viewer to visualize information returned from this endpoint; it will be very similar to "Network" tabs in Firefox and Chrome developer tools.

Currently this endpoint doesn't expose raw request contents; only meta-information like headers and timings is available. Response contents is included when *'response_body'* option is set to 1.

Arguments for this endpoint are the same as for *render.html*, plus the following:

response_body [int][optional] Possible values are `1` and `0`. When `response_body=1`, response content is included in HAR records. Default is `response_body=0`.

render.json

Return a json-encoded dictionary with information about javascript-rendered webpage. It can include HTML, PNG and other information, based on arguments passed.

Arguments:

Same as *render.jpeg* plus the following ones:

html [integer][optional] Whether to include HTML in output. Possible values are `1` (include) and `0` (exclude). Default is 0.

png [integer][optional] Whether to include PNG in output. Possible values are `1` (include) and `0` (exclude). Default is 0.

jpeg [integer][optional] Whether to include JPEG in output. Possible values are `1` (include) and `0` (exclude). Default is 0.

iframes [integer][optional] Whether to include information about child frames in output. Possible values are `1` (include) and `0` (exclude). Default is 0.

script [integer][optional] Whether to include the result of the executed javascript final statement in output (see *Executing custom Javascript code within page context*). Possible values are `1` (include) and `0` (exclude). Default is 0.

console [integer][optional] Whether to include the executed javascript console messages in output. Possible values are `1` (include) and `0` (exclude). Default is 0.

history [integer][optional] Whether to include the history of requests/responses for webpage main frame. Possible values are `1` (include) and `0` (exclude). Default is 0.

Use it to get HTTP status codes and headers. Only information about "main" requests/responses is returned (i.e. information about related resources like images and AJAX queries is not returned). To get information about all requests and responses use *'har'* argument.

har [integer][optional] Whether to include HAR in output. Possible values are `1` (include) and `0` (exclude). Default is 0. If this option is ON the result will contain the same data as *render.har* provides under 'har' key.

By default, response content is not included. To enable it use *'response_body'* option.

response_body [int][optional] Possible values are `1` and `0`. When `response_body=1`, response content is included in HAR records. Default is `response_body=0`. This option has no effect when both *'har'* and *'history'* are 0.

Examples

By default, URL, requested URL, page title and frame geometry is returned:

```
{
    "url": "http://crawlera.com/",
    "geometry": [0, 0, 640, 480],
    "requestedUrl": "http://crawlera.com/",
    "title": "Crawlera"
}
```

Add 'html=1' to request to add HTML to the result:

```
{
    "url": "http://crawlera.com/",
    "geometry": [0, 0, 640, 480],
    "requestedUrl": "http://crawlera.com/",
    "html": "<!DOCTYPE html><!--[if IE 8]>....",
    "title": "Crawlera"
}
```

Add 'png=1' to request to add base64-encoded PNG screenshot to the result:

```
{
    "url": "http://crawlera.com/",
    "geometry": [0, 0, 640, 480],
    "requestedUrl": "http://crawlera.com/",
    "png": "iVBORw0KGgoAAAAN...",
    "title": "Crawlera"
}
```

Setting both 'html=1' and 'png=1' allows to get HTML and a screenshot at the same time - this guarantees that the screenshot matches the HTML.

By adding "iframes=1" information about iframes could be obtained:

```
{
    "geometry": [0, 0, 640, 480],
    "frameName": "",
    "title": "Scrapinghub | Autoscraping",
    "url": "http://scrapinghub.com/autoscraping.html",
    "childFrames": [
        {
            "title": "Tutorial: Scrapinghub's autoscraping tool - YouTube",
            "url": "",
            "geometry": [235, 502, 497, 310],
            "frameName": "<!--framePath //<!--frame0-->-->",
            "requestedUrl": "http://www.youtube.com/embed/lSJvVqDLOOs?version=3&rel=1&
↪fs=1&showsearch=0&showinfo=1&iv_load_policy=1&wmode=transparent",
            "childFrames": []
        }
    ],
    "requestedUrl": "http://scrapinghub.com/autoscraping.html"
}
```

Note that iframes can be nested.

Pass both 'html=1' and 'iframes=1' to get HTML for all iframes as well as for the main page:

```
 {
    "geometry": [0, 0, 640, 480],
    "frameName": "",
    "html": "<!DOCTYPE html...",
    "title": "Scrapinghub | Autoscraping",
    "url": "http://scrapinghub.com/autoscraping.html",
    "childFrames": [
        {
            "title": "Tutorial: Scrapinghub's autoscraping tool - YouTube",
            "url": "",
            "html": "<!DOCTYPE html>...",
            "geometry": [235, 502, 497, 310],
            "frameName": "<!--framePath //<!--frame0-->-->",
            "requestedUrl": "http://www.youtube.com/embed/lSJvVqDLOOs?version=3&rel=1&
↪fs=1&showsearch=0&showinfo=1&iv_load_policy=1&wmode=transparent",
            "childFrames": []
        }
    ],
    "requestedUrl": "http://scrapinghub.com/autoscraping.html"
}
```

Unlike 'html=1', 'png=1' does not affect data in childFrames.

When executing JavaScript code (see *Executing custom Javascript code within page context*) add the parameter 'script=1' to the request to include the code output in the result:

```
{
    "url": "http://crawlera.com/",
    "geometry": [0, 0, 640, 480],
    "requestedUrl": "http://crawlera.com/",
    "title": "Crawlera",
    "script": "result of script..."
}
```

The JavaScript code supports the console.log() function to log messages. Add 'console=1' to the request to include the console output in the result:

```
{
    "url": "http://crawlera.com/",
    "geometry": [0, 0, 640, 480],
    "requestedUrl": "http://crawlera.com/",
    "title": "Crawlera",
    "script": "result of script...",
    "console": ["first log message", "second log message", ...]
}
```

Curl examples:

```
# full information
curl 'http://localhost:8050/render.json?url=http://domain.com/page-with-iframes.html&
↪png=1&html=1&iframes=1'

# HTML and meta information of page itself and all its iframes
curl 'http://localhost:8050/render.json?url=http://domain.com/page-with-iframes.html&
↪html=1&iframes=1'

# only meta information (like page/iframes titles and urls)
curl 'http://localhost:8050/render.json?url=http://domain.com/page-with-iframes.html&
↪iframes=1'

# render html and 320x240 thumbnail at once; do not return info about iframes
curl 'http://localhost:8050/render.json?url=http://domain.com/page-with-iframes.html&
↪html=1&png=1&width=320&height=240'

# Render page and execute simple Javascript function, display the js output
```

```
curl -X POST -H 'content-type: application/javascript' \
    -d 'function getAd(x){ return x; } getAd("abc");' \
    'http://localhost:8050/render.json?url=http://domain.com&script=1'

# Render page and execute simple Javascript function, display the js output and the␣
↪console output
curl -X POST -H 'content-type: application/javascript' \
    -d 'function getAd(x){ return x; }; console.log("some log"); console.log("another␣
↪log"); getAd("abc");' \
    'http://localhost:8050/render.json?url=http://domain.com&script=1&console=1'
```

execute

Execute a custom rendering script and return a result.

render.html, *render.png*, *render.jpeg*, *render.har* and *render.json* endpoints cover many common use cases, but sometimes they are not enough. This endpoint allows to write custom *Splash Scripts*.

Arguments:

lua_source [string][required] Browser automation script. See *Splash Scripts Tutorial* for more info.

timeout [float][optional] Same as *'timeout'* argument for *render.html*.

allowed_domains [string][optional] Same as *'allowed_domains'* argument for *render.html*.

proxy [string][optional] Same as *'proxy'* argument for *render.html*.

filters [string][optional] Same as *'filters'* argument for *render.html*.

save_args [JSON array or a comma-separated string][optional] Same as *'save_args'* argument for *render.html*. Note that you can save not only default Splash arguments, but any other parameters as well.

load_args [JSON object or a string][optional] Same as *'load_args'* argument for *render.html*. Note that you can load not only default Splash arguments, but any other parameters as well.

You can pass any other arguments. All arguments passed to *execute* endpoint are available in a script in *splash.args* table.

Executing custom Javascript code within page context

Note: See also: *executing JavaScript in Splash scripts*

Splash supports executing JavaScript code within the context of the page. The JavaScript code is executed after the page finished loading (including any delay defined by 'wait') but before the page is rendered. This allow to use the javascript code to modify the page being rendered.

To execute JavaScript code use *js_source* parameter. It should contain JavaScript code to be executed.

Note that browsers and proxies limit the amount of data can be sent using GET, so it is a good idea to use `content-type: application/json` POST request.

Curl example:

```
# Render page and modify its title dynamically
curl -X POST -H 'content-type: application/json' \
    -d '{"js_source": "document.title=\"My Title\";", "url": "http://example.com"}' \
    'http://localhost:8050/render.html'
```

Another way to do it is to use a POST request with the content-type set to 'application/javascript'. The body of the request should contain the code to be executed.

Curl example:

```
# Render page and modify its title dynamically
curl -X POST -H 'content-type: application/javascript' \
    -d 'document.title="My Title";' \
    'http://localhost:8050/render.html?url=http://domain.com'
```

To get the result of a javascript function executed within page context use *render.json* endpoint with *script* = 1 parameter.

Javascript Profiles

Splash supports "javascript profiles" that allows to preload javascript files. Javascript files defined in a profile are executed after the page is loaded and before any javascript code defined in the request.

The preloaded files can be used in the user's POST'ed code.

To enable javascript profiles support, run splash server with the `--js-profiles-path=<path to a folder with js profiles>` option:

```
python -m splash.server --js-profiles-path=/etc/splash/js-profiles
```

Note: See also: *Splash Versions*.

Then create a directory with the name of the profile and place inside it the javascript files to load (note they must be utf-8 encoded). The files are loaded in the order they appear in the filesystem. Directory example:

```
/etc/splash/js-profiles/
                    mywebsite/
                          lib1.js
```

To apply this javascript profile add the parameter `js=mywebsite` to the request:

```
curl -X POST -H 'content-type: application/javascript' \
    -d 'myfunc("Hello");' \
    'http://localhost:8050/render.html?js=mywebsite&url=http://domain.com'
```

Note that this example assumes that myfunc is a javascript function defined in lib1.js.

Javascript Security

If Splash is started with `--js-cross-domain-access` option

```
python -m splash.server --js-cross-domain-access
```

then javascript code is allowed to access the content of iframes loaded from a security origin diferent to the original page (browsers usually disallow that). This feature is useful for scraping, e.g. to extract the html of a iframe page. An example of its usage:

```
curl -X POST -H 'content-type: application/javascript' \
    -d 'function getContents(){ var f = document.getElementById("external"); return f.
↪contentDocument.getElementsByTagName("body")[0].innerHTML; }; getContents();' \
    'http://localhost:8050/render.html?url=http://domain.com'
```

The javascript function 'getContents' will look for a iframe with the id 'external' and extract its html contents.

Note that allowing cross origin javascript calls is a potential security issue, since it is possible that secret information (i.e cookies) is exposed when this support is enabled; also, some websites don't load when cross-domain security is disabled, so this feature is OFF by default.

Request Filters

Splash supports filtering requests based on Adblock Plus rules. You can use filters from EasyList to remove ads and tracking codes (and thus speedup page loading), and/or write filters manually to block some of the requests (e.g. to prevent rendering of images, mp3 files, custom fonts, etc.)

To activate request filtering support start splash with `--filters-path` option:

```
python -m splash.server --filters-path=/etc/splash/filters
```

Note: See also: *Splash Versions*.

The folder `--filters-path` points to should contain `.txt` files with filter rules in Adblock Plus format. You may download `easylist.txt` from EasyList and put it there, or create `.txt` files with your own rules.

For example, let's create a filter that will prevent custom fonts in `ttf` and `woff` formats from loading (due to qt bugs they may cause splash to segfault on Mac OS X):

```
! put this to a /etc/splash/filters/nofonts.txt file
! comments start with an exclamation mark

.ttf|
.woff|
```

To use this filter in a request add `filters=nofonts` parameter to the query:

```
curl 'http://localhost:8050/render.png?url=http://domain.com/page-with-fonts.html&
↪filters=nofonts'
```

You can apply several filters; separate them by comma:

```
curl 'http://localhost:8050/render.png?url=http://domain.com/page-with-fonts.html&
↪filters=nofonts,easylist'
```

If `default.txt` file is present in `--filters-path` folder it is used by default when `filters` argument is not specified. Pass `filters=none` if you don't want default filters to be applied.

Only related resources are filtered out by request filters; 'main' page loading request can't be blocked this way. If you really want to do that consider checking URL against Adblock Plus filters before sending it to Splash (e.g. for Python there is adblockparser library).

To learn about Adblock Plus filter syntax check these links:

- https://adblockplus.org/en/filter-cheatsheet
- https://adblockplus.org/en/filters

Splash doesn't support full Adblock Plus filters syntax, there are some limitations:

- element hiding rules are not supported; filters can prevent network request from happening, but they can't hide parts of an already loaded page;
- only `domain` option is supported.

Unsupported rules are silently discarded.

Note: If you want to stop downloading images check *'images'* parameter. It doesn't require URL-based filters to work, and it can filter images that are hard to detect using URL-based patterns.

Warning: It is very important to have pyre2 library installed if you are going to use filters with a large number of rules (this is the case for files downloaded from EasyList).

Without pyre2 library splash (via adblockparser) relies on re module from stdlib, and it can be 1000x+ times slower than re2 - it may be faster to download files than to discard them if you have a large number of rules and don't use re2. With re2 matching becomes very fast.

Make sure you are not using re2==0.2.20 installed from PyPI (it is broken); use the latest version.

Proxy Profiles

Splash supports "proxy profiles" that allows to set proxy handling rules per-request using `proxy` parameter.

To enable proxy profiles support, run splash server with `--proxy-profiles-path=<path to a folder with proxy profiles>` option:

```
python -m splash.server --proxy-profiles-path=/etc/splash/proxy-profiles
```

Note: If you run Splash using Docker, check *Folders Sharing*.

Then create an INI file with "proxy profile" config inside the specified folder, e.g. `/etc/splash/proxy-profiles/mywebsite.ini`. Example contents of this file:

```
[proxy]

; required
host=proxy.crawlera.com
port=8010

; optional, default is no auth
username=username
password=password

; optional, default is HTTP. Allowed values are HTTP and SOCKS5
type=HTTP

[rules]
; optional, default ".*"
whitelist=
    .*mywebsite\.com.*
```

```
; optional, default is no blacklist
blacklist=
    .*\.js.*
    .*\.css.*
    .*\.png
```

whitelist and blacklist are newline-separated lists of regexes. If URL matches one of whitelist patterns and matches none of blacklist patterns, proxy specified in `[proxy]` section is used; no proxy is used otherwise.

Then, to apply proxy rules according to this profile, add `proxy=mywebsite` parameter to request:

```
curl 'http://localhost:8050/render.html?url=http://mywebsite.com/page-with-javascript.
↪html&proxy=mywebsite'
```

If `default.ini` profile is present, it will be used when `proxy` argument is not specified. If you have `default.ini` profile but don't want to apply it pass `none` as `proxy` value.

Other Endpoints

_gc

To reclaim some RAM send a POST request to the `/_gc` endpoint:

```
curl -X POST http://localhost:8050/_gc
```

It runs the Python garbage collector and clears internal WebKit caches.

_debug

To get debug information about Splash instance (max RSS used, number of used file descriptors, active requests, request queue length, counts of alive objects) send a GET request to the `/_debug` endpoint:

```
curl http://localhost:8050/_debug
```

_ping

To ping Splash instance send a GET request to the `/_ping` endpoint:

```
curl http://localhost:8050/_ping
```

It returns "ok" status and max RSS used, if instance is alive.

Splash Scripts Tutorial

Intro

Splash can execute custom rendering scripts written in the Lua programming language. This allows us to use Splash as a browser automation tool similar to PhantomJS.

To execute a script and get the result back send it to the *execute* endpoint in a *lua_source* argument.

Note: Most likely you'll be able to follow Splash scripting examples even without knowing Lua; nevertheless, the language is worth learning. With Lua you can, for example, write Redis, Nginx, Apache, World of Warcraft scripts, create mobile apps using Moai or Corona SDK or use the state of the art Deep Learning framework Torch7. It is easy to get started and there are good online resources available like the tutorial Learn Lua in 15 minutes and the book Programming in Lua.

Let's start with a basic example:

```
function main(splash)
    splash:go("http://example.com")
    splash:wait(0.5)
    local title = splash:evaljs("document.title")
    return {title=title}
end
```

If we submit this script to the *execute* endpoint in a `lua_source` argument, Splash will go to the example.com website, wait until it loads, wait aother half-second, then get the page title (by evaluating a JavaScript snippet in page context), and then return the result as a JSON encoded object.

Note: Splash UI provides an easy way to try scripts: there is a code editor for Lua and a button to submit a script to `execute`. Visit http://127.0.0.1:8050/ (or whatever host/port Splash is listening to).

Entry Point: the "main" Function

The script must provide a "main" function which is called by Splash. The result is returned as an HTTP response. The script could contain other helper functions and statements, but 'main' is required.

In the first example 'main' function returned a Lua table (an associative array similar to JavaScript Object or Python dict). Such results are returned as JSON.

The following will return the string `{"hello":"world!"}` as an HTTP response:

```
function main(splash)
    return {hello="world!"}
end
```

The script can also return a string:

```
function main(splash)
    return 'hello'
end
```

Strings are returned as-is (unlike tables they are not encoded to JSON). Let's check it with curl:

```
$ curl 'http://127.0.0.1:8050/execute?lua_source=function+main%28splash%29%0D
↪%0A++return+%27hello%27%0D%0Aend'
hello
```

The "main" function receives an object that allows us to control the "browser tab". All Splash features are exposed using this object. By convention, this argument is called "splash", but you are not required to follow this convention:

```
function main(please)
    please:go("http://example.com")
    please:wait(0.5)
    return "ok"
end
```

Where Are My Callbacks?

Here is a snippet from our first example:

```
splash:go("http://example.com")
splash:wait(0.5)
local title = splash:evaljs("document.title")
```

The code looks like standard procedural code; there are no callbacks or fancy control-flow structures. It doesn't mean Splash works in a synchronous way; under the hood it is still async. When you call `splash.wait(0.5)`, Splash switches from the script to other tasks, and comes back after 0.5s.

It is possible to use loops, conditional statements, functions as usual in Splash scripts which enables more straightforward coding.

Let's check an example PhantomJS script:

```
var users = ["PhantomJS", "ariyahidayat", /*...*/];

function followers(user, callback) {
    var page = require('webpage').create();
    page.open('http://mobile.twitter.com/' + user, function (status) {
        if (status === 'fail') {
            console.log(user + ': ?');
        } else {
            var data = page.evaluate(function () {
                return document.querySelector('div.profile td.stat.stat-last div.
↪statnum').innerText;
            });
            console.log(user + ': ' + data);
        }
        page.close();
        callback.apply();
    });
}
function process() {
    if (users.length > 0) {
        var user = users[0];
        users.splice(0, 1);
        followers(user, process);
    } else {
        phantom.exit();
    }
}
process();
```

The code is (arguably) tricky: `process` function implements a loop by creating a chain of callbacks; `followers` function doesn't return a value (it would be more complex to implement) - the result is logged to the console instead.

A similar Splash script:

```
-- Implementation of "follow.js" example from
-- PhantomJS
```

```lua
-- https://github.com/ariya/phantomjs/blob/master/examples/follow.js

USERS = {
  'PhantomJS',
  'ariyahidayat',
  'detronizator',
  'KDABQt',
  'lfranchi',
  'jonleighton',
  '_jamesmgreene',
  'Vitalliumm',
}

local base = 'http://mobile.twitter.com/'
function follow(splash, user)
  local ok, msg = splash:go(base .. user)
  if not ok then
    return "Can't get followers of " .. user .. ': ' .. msg
  end
  return splash:evaljs([[
    document.querySelector('div.profile td.stat.stat-last div.statnum').innerText;
  ]]);
end

function process(splash, users)
  local result = {}
  for idx, user in ipairs(users) do
    result[user] = follow(splash, user)
  end
  return result
end

function main(splash)
  return {
    users=process(splash, USERS),
  }
end
```

Observations:

- some Lua knowledge is helpful to be productive in Splash Scripts: `ipairs`, `[[multi-line strings]]` or string concatenation via `..` could be unfamiliar;
- in Splash variant `followers` function can return a result (a number of twitter followers); also, it doesn't need a "callback" argument;
- instead of a `page.open` callback which receives "status" argument there is a "blocking" *splash:go* call which returns "ok" flag;
- error handling is different: in case of an HTTP 4xx or 5xx error PhantomJS doesn't return an error code to `page.open` callback - example script will try to get the followers nevertheless because "status" won't be "fail"; in Splash this error will be detected and "?" will be returned;
- `process` function can use a standard Lua `for` loop without a need to create a recursive callback chain;
- instead of console messages we've got a JSON HTTP API;
- apparently, PhantomJS allows to create multiple `page` objects and run several `page.open` requests in parallel (?); Splash only provides a single "browser tab" to a script via its `splash` parameter of `main` function (but you're free to send multiple concurrent requests with Lua scripts to Splash).

There are great PhantomJS wrappers like CasperJS and NightmareJS which (among other things) bring a sync-looking syntax to PhantomJS scripts by providing custom control flow mini-languages. However, they all have their own gotchas and edge cases (loops? moving code to helper functions? error handling?). Splash scripts are standard Lua code.

Note: PhantomJS itself and its wrappers are great, they deserve lots of respect; please don't take this writeup as an attack on them. These tools are much more mature and feature complete than Splash. Splash tries to look at the problem from a different angle, but for each unique Splash feature there is an unique PhantomJS feature.

Living Without Callbacks

Note: For the curious, Splash uses Lua coroutines under the hood.

Internally, "main" function is executed as a coroutine by Splash, and some of the `splash:foo()` methods use `coroutine.yield`. See http://www.lua.org/pil/9.html for Lua coroutines tutorial.

In Splash scripts it is not explicit which calls are async and which calls are blocking; this is a common criticism of coroutines/greenlets. Check this article for a good description of the problem.

However, these negatives have no real impact in Splash scripts which: are meant to be small, where shared state is minimized, and the API is designed to execute a single command at a time, so in most cases the control flow is linear.

If you want to be safe then think of all `splash` methods as async; consider that after you call `splash:foo()` a webpage being rendered can change. Often that's the point of calling a method, e.g. `splash:wait(time)` or `splash:go(url)` only make sense because webpage changes after calling them, but still - keep it in mind.

There are async methods like *splash:go*, *splash:wait*, *splash:wait_for_resume*, etc.; most splash methods are currently **not** async, but thinking of them as of async will allow your scripts to work if we ever change that.

Calling Splash Methods

Unlike in many languages, methods in Lua are usually separated from an object using a colon `:`; to call "foo" method of "splash" object use `splash:foo()` syntax. See http://www.lua.org/pil/16.html for more details.

There are two main ways to call Lua methods in Splash scripts: using positional and named arguments. To call a method using positional arguments use parentheses `splash:foo(val1, val2)`, to call it with named arguments use curly braces: `splash:foo{name1=val1, name2=val2}`:

```lua
-- Examples of positional arguments:
splash:go("http://example.com")
splash:wait(0.5, false)
local title = splash:evaljs("document.title")

-- The same using keyword arguments:
splash:go{url="http://example.com"}
splash:wait{time=0.5, cancel_on_redirect=false}
local title = splash:evaljs{source="document.title"}

-- Mixed arguments example:
splash:wait{0.5, cancel_on_redirect=false}
```

For convenience all `splash` methods are designed to support both styles of calling: positional and named. But since there are no "real" named arguments in Lua most Lua functions (including the ones from the standard library) choose to support just positional arguments.

Error Handling

There are two ways to report errors in Lua: raise an exception and return an error flag. See http://www.lua.org/pil/8.3.html.

Splash uses the following convention:

1. for developer errors (e.g. incorrect function arguments) exception is raised;
2. for errors outside developer control (e.g. a non-responding remote website) status flag is returned: functions that can fail return `ok, reason` pairs which developer can either handle or ignore.

If `main` results in an unhandled exception then Splash returns HTTP 400 response with an error message.

It is possible to raise an exception manually using Lua `error` function:

```
error("A message to be returned in a HTTP 400 response")
```

To handle Lua exceptions (and prevent Splash from returning HTTP 400 response) use Lua `pcall`; see http://www.lua.org/pil/8.4.html.

To convert "status flag" errors to exceptions Lua `assert` function can be used. For example, if you expect a website to work and don't want to handle errors manually, then `assert` allows to stop processing and return HTTP 400 if the assumption is wrong:

```
local ok, msg = splash:go("http://example.com")
if not ok then
    -- handle error somehow, e.g.
    error(msg)
end

-- a shortcut for the code above: use assert
assert(splash:go("http://example.com"))
```

Sandbox

By default Splash scripts are executed in a restricted environment: not all standard Lua modules and functions are available, Lua `require` is restricted, and there are resource limits (quite loose though).

To disable the sandbox start Splash with `--disable-lua-sandbox` option:

```
$ python -m splash.server --disable-lua-sandbox
```

Timeouts

By default Splash aborts script execution after a timeout (30s by default); it is a common problem for long scripts.

For more information see *I'm getting lots of 504 Timeout errors, please help!* and *2. Splash Lua script does too many things*.

Splash Scripts Reference

`splash` object is passed to `main` function; via this object a script can control the browser. Think of it as of an API to a single browser tab.

Attributes

splash.args

`splash.args` is a table with incoming parameters. It contains merged values from the orignal URL string (GET arguments) and values sent using `application/json` POST request.

For example, if you passed 'url' argument to a script using HTTP API, then `splash.args.url` contains this URL.

splash.args is the preferred way to pass parameters to Splash scripts. An alternative way is to use string formatting to build a script with variables embedded. There are two problems which make *splash.args* a better solution:

1. data must be escaped somehow, so that it doesn't break a Lua script;
2. embedding variables makes it impossible to use script cache efficiently (see *save_args* and *load_args* arguments of the HTTP API).

splash.js_enabled

Enable or disable execution of JavaSript code embedded in the page.

Signature: `splash.js_enabled = true/false`

JavaScript execution is enabled by default.

splash.private_mode_enabled

Enable or disable browser's private mode (incognito mode).

Signature: `splash.private_mode_enabled = true/false`

Private mode is enabled by default unless you pass flag `--disable-private-mode` at Splash startup. Note that if you disable private mode browsing data such as cookies or items kept in local storage may persist between requests.

splash.resource_timeout

Set a default timeout for network requests, in seconds.

Signature: `splash.resource_timeout = number`

Example - abort requests to remote resources if they take more than 10 seconds:

```lua
function main(splash)
    splash.resource_timeout = 10.0
    assert(splash:go(splash.args.url))
    return splash:png()
end
```

Zero or nil value means "no timeout".

Request timeouts set in *splash:on_request* using `request:set_timeout` have a priority over *splash.resource_timeout*.

splash.images_enabled

Enable/disable images.

Signature: `splash.images_enabled = true/false`

By default, images are enabled. Disabling of the images can save a lot of network traffic (usually around ~50%) and make rendering faster. Note that this option can affect the JavaScript code inside page: disabling of the images may change sizes and positions of DOM elements, and scripts may read and use them.

Splash uses in-memory cache; cached images will be displayed even when images are disabled. So if you load a page, then disable images, then load a new page, then likely first page will display all images and second page will display some images (the ones common with the first page). Splash cache is shared between scripts executed in the same process, so you can see some images even if they are disabled at the beginning of the script.

Example:

```
function main(splash)
    splash.images_enabled = false
    assert(splash:go(splash.args.url))
    return {png=splash:png()}
end
```

splash.plugins_enabled

Enable or disable browser plugins (e.g. Flash).

Signature: `splash.plugins_enabled = true/false`

Plugins are disabled by default.

splash.response_body_enabled

Enable or disable response content tracking.

Signature: `splash.response_body_enabled = true/false`

By default Splash doesn't keep bodies of each response in memory, for efficiency reasons. It means that in *splash:on_response* callbacks *response.body* attribute is not available, and that response content is not available in HAR exports. To make response content available to a Lua script set `splash.response_body_enabled = true`.

Note that *response.body* is always available in *splash:http_get* and *splash:http_post* results, regardless of *splash.response_body_enabled* option.

To enable response content tracking per-request call *request:enable_response_body* in a *splash:on_request* callback.

Methods

splash:go

Go to an URL. This is similar to entering an URL in a browser address bar, pressing Enter and waiting until page loads.

Signature: `ok, reason = splash:go{url, baseurl=nil, headers=nil, http_method="GET", body=nil, formdata=nil}`

Parameters:

- url - URL to load;
- baseurl - base URL to use, optional. When `baseurl` argument is passed the page is still loaded from `url`, but it is rendered as if it was loaded from `baseurl`: relative resource paths will be relative to `baseurl`, and the browser will think `baseurl` is in address bar;
- headers - a Lua table with HTTP headers to add/replace in the initial request.
- http_method - optional, string with HTTP method to use when visiting url, defaults to GET, Splash also supports POST.
- body - optional, string with body for POST request
- formdata - Lua table that will be converted to urlencoded POST body and sent with header `content-type: application/x-www-form-urlencoded`

Returns: `ok, reason` pair. If `ok` is nil then error happened during page load; `reason` provides an information about error type.

Async: yes, unless the navigation is locked.

Five types of errors are reported (`ok` can be `nil` in 5 cases):

1. There is a network error: a host doesn't exist, server dropped connection, etc. In this case `reason` is `"network<code>"`. A list of possible error codes can be found in Qt docs. For example, `"network3"` means a DNS error (invalid hostname).
2. Server returned a response with 4xx or 5xx HTTP status code. `reason` is `"http<code>"` in this case, i.e. for HTTP 404 Not Found `reason` is `"http404"`.
3. Navigation is locked (see *splash:lock_navigation*); `reason` is `"navigation_locked"`.
4. Splash can't render the main page (e.g. because the first request was aborted) - `reason` is `render_error`.
5. If Splash can't decide what caused the error, just `"error"` is returned.

Error handling example:

```
local ok, reason = splash:go("http://example.com")
if not ok then
    if reason:sub(0,4) == 'http' then
        -- handle HTTP errors
    else
        -- handle other errors
    end
end
-- process the page

-- assert can be used as a shortcut for error handling
assert(splash:go("http://example.com"))
```

Errors (ok==nil) are only reported when "main" webpage request failed. If a request to a related resource failed then no error is reported by `splash:go`. To detect and handle such errors (e.g. broken image/js/css links, ajax requests failed to load) use *splash:har* or *splash:on_response*.

`splash:go` follows all HTTP redirects before returning the result, but it doesn't follow HTML `<meta http-equiv="refresh" ...>` redirects or redirects initiated by JavaScript code. To give the webpage time to follow those redirects use *splash:wait*.

`headers` argument allows to add or replace default HTTP headers for the initial request. To set custom headers for all further requests (including requests to related resources) use *splash:set_custom_headers* or *splash:on_request*.

Custom headers example:

```
local ok, reason = splash:go{"http://example.com", headers={
    ["Custom-Header"] = "Header Value",
}})
```

User-Agent header is special: once used, it is kept for further requests. This is an implementation detail and it could change in future releases; to set User-Agent header it is recommended to use *splash:set_user_agent* method.

splash:wait

Wait for `time` seconds. When script is waiting WebKit continues processing the webpage.

Signature: `ok, reason = splash:wait{time, cancel_on_redirect=false, cancel_on_error=true}`

Parameters:

- time - time to wait, in seconds;
- cancel_on_redirect - if true (not a default) and a redirect happened while waiting, then `splash:wait` stops earlier and returns `nil, "redirect"`. Redirect could be initiated by `<meta http-equiv="refresh" ...>` HTML tags or by JavaScript code.
- cancel_on_error - if true (default) and an error which prevents page from being rendered happened while waiting (e.g. an internal WebKit error or a network error like a redirect to a non-resolvable host) then `splash:wait` stops earlier and returns `nil, "<error string>"`.

Returns: `ok, reason` pair. If `ok` is `nil` then the timer was stopped prematurely, and `reason` contains a string with a reason.

Async: yes.

Usage example:

```
-- go to example.com, wait 0.5s, return rendered html, ignore all errors.
function main(splash)
    splash:go("http://example.com")
    splash:wait(0.5)
    return {html=splash:html()}
end
```

By default wait timer continues to tick when redirect happens. `cancel_on_redirect` option can be used to restart the timer after each redirect. For example, here is a function that waits for a given time after each page load in case of redirects:

```
function wait_restarting_on_redirects(splash, time, max_redirects)
    local redirects_remaining = max_redirects
    while redirects_remaining > 0 do
        local ok, reason = self:wait{time=time, cancel_on_redirect=true}
        if reason ~= 'redirect' then
            return ok, reason
        end
        redirects_remaining = redirects_remaining - 1
    end
    return nil, "too_many_redirects"
end
```

splash:jsfunc

Convert JavaScript function to a Lua callable.

Signature: `lua_func = splash:jsfunc(func)`

Parameters:

- func - a string which defines a JavaScript function.

Returns: a function that can be called from Lua to execute JavaScript code in page context.

Async: no.

Example:

```
function main(splash)
  local get_div_count = splash:jsfunc([[
    function () {
      var body = document.body;
      var divs = body.getElementsByTagName('div');
      return divs.length;
    }
  ]])

  local url = splash.args.url
  splash:go(url)
  return string.format("There are %s DIVs in %s",
      get_div_count(), url)
end
```

Note how Lua `[[ ]]` string syntax is helpful here.

JavaScript functions may accept arguments:

```
local vec_len = splash:jsfunc([[
    function(x, y) {
       return Math.sqrt(x*x + y*y)
    }
]])
return {res=vec_len(5, 4)}
```

Global JavaScript functions can be wrapped directly:

```
local pow = splash:jsfunc("Math.pow")
local twenty_five = pow(5, 2)  -- 5^2 is 25
local thousand = pow(10, 3)    -- 10^3 is 1000
```

Lua → JavaScript conversion rules:

Lua	JavaScript
string	string
number	number
boolean	boolean
table	Object or Array, see below
nil	undefined
Element	DOM node

Lua strings, numbers, booleans and tables can be passed as arguments; they are converted to JS strings/numbers/booleans/objects. *Element* instances are supported, but they can't be inside a Lua table.

Currently it is not possible to pass other Lua objects. For example, it is not possible to pass a wrapped JavaScript function or a regular Lua function as an argument to another wrapped JavaScript function.

By default Lua tables are converted to JavaScript Objects. To convert a table to an Array use *treat.as_array*. JavaScript → Lua conversion rules:

JavaScript	Lua
string	string
number	number
boolean	boolean
Object	table
Array	table, marked as array (see *treat.as_array*)
`undefined`	`nil`
`null`	`""` (an empty string)
Date	string: date's ISO8601 representation, e.g. `1958-05-21T10:12:00.000Z`
Node	*Element* instance
NodeList	a tabl with *Element* instances
function	`nil`
circular object	`nil`
host object	`nil`

Function result is converted from JavaScript to Lua data type. Only simple JS objects are supported. For example, returning a function or a JQuery selector from a wrapped function won't work.

Returning a Node (a reference to a DOM element) or NodeList instance (result of document.querySelectorAll) works though, but only if Node or NodeList is the only result - Nodes and NodeLists can't be inside other objects or arrays.

Note: The rule of thumb: if an argument or a return value can be serialized via JSON, then it is fine. You can also return DOM Element or a NodeList, but they can't be inside other data structures.

Note that currently you can't return JQuery $ results and similar structures from JavaScript to Lua; to pass data you have to extract their attributes of interest as plain strings/numbers/objects/arrays:

```
-- this function assumes jQuery is loaded in page
local get_hrefs = splash:jsfunc([[
    function(sel){
        return $(sel).map(function(){return this.href}).get();
    }
]])
local hrefs = get_hrefs("a.story-title")
```

However, you can also write the code above using *Element* objects and *splash:select_all*:

```
local elems = splash:select_all("a.story-title")
local hrefs = {}
for i, elem in ipairs(elems) do
    hrefs[i] = elem.node:getAttribute("href")
end
```

Function arguments and return values are passed by value. For example, if you modify an argument from inside a JavaScript function then the caller Lua code won't see the changes, and if you return a global JS object and modify it in Lua then object won't be changed in webpage context. The exception is *Element* which has some mutable fields.

If a JavaScript function throws an error, it is re-throwed as a Lua error. To handle errors it is better to use JavaScript try/catch because some of the information about the error can be lost in JavaScript → Lua conversion.

See also: *splash:runjs*, *splash:evaljs*, *splash:wait_for_resume*, *splash:autoload*, *treat.as_array*, *Element Object*, *splash:select*, *splash:select_all*.

splash:evaljs

Execute a JavaScript snippet in page context and return the result of the last statement.

Signature: `result = splash:evaljs(snippet)`

Parameters:

- snippet - a string with JavaScript source code to execute.

Returns: the result of the last statement in `snippet`, converted from JavaScript to Lua data types. In case of syntax errors or JavaScript exceptions an error is raised.

Async: no.

JavaScript → Lua conversion rules are the same as for *splash:jsfunc*.

`splash:evaljs` is useful for evaluation of short JavaScript snippets without defining a wrapper function. Example:

```
local title = splash:evaljs("document.title")
```

Don't use *splash:evaljs* when the result is not needed - it is inefficient and could lead to problems; use *splash:runjs* instead. For example, the following innocent-looking code (using jQuery) will do unnecessary work:

```
splash:evaljs("$(console.log('foo'));")
```

A gotcha is that to allow chaining jQuery `$` function returns a huge object, *splash:evaljs* tries to serialize it and convert to Lua, which is a waste of resources. *splash:runjs* doesn't have this problem.

If the code you're evaluating needs arguments it is better to use *splash:jsfunc* instead of *splash:evaljs* and string formatting. Compare:

```
function main(splash)

    local font_size = splash:jsfunc([[
        function(sel) {
            var el = document.querySelector(sel);
            return getComputedStyle(el)["font-size"];
        }
    ]])

    local font_size2 = function(sel)
        -- FIXME: escaping of `sel` parameter!
        local js = string.format([[
            var el = document.querySelector("%s");
            getComputedStyle(el)["font-size"]
        ]], sel)
        return splash:evaljs(js)
    end

    -- ...
end
```

See also: *splash:runjs*, *splash:jsfunc*, *splash:wait_for_resume*, *splash:autoload*, *Element Object*, *splash:select*, *splash:select_all*.

splash:runjs

Run JavaScript code in page context.

Signature: `ok, error = splash:runjs(snippet)`

Parameters:

- snippet - a string with JavaScript source code to execute.

Returns: `ok, error` pair. When the execution is successful `ok` is True. In case of JavaScript errors `ok` is `nil`, and `error` contains the error string.

Async: no.

Example:

```
assert(splash:runjs("document.title = 'hello';"))
```

Note that JavaScript functions defined using `function foo(){}` syntax **won't** be added to the global scope:

```
assert(splash:runjs("function foo(){return 'bar'}"))
local res = splash:evaljs("foo()")  -- this raises an error
```

It is an implementation detail: the code passed to *splash:runjs* is executed in a closure. To define functions use global variables, e.g.:

```
assert(splash:runjs("foo = function (){return 'bar'}"))
local res = splash:evaljs("foo()")  -- this returns 'bar'
```

If the code needs arguments it is better to use *splash:jsfunc*. Compare:

```
function main(splash)

    -- Lua function to scroll window to (x, y) position.
    function scroll_to(x, y)
        local js = string.format(
            "window.scrollTo(%s, %s);",
            tonumber(x),
            tonumber(y)
        )
        assert(splash:runjs(js))
    end

    -- a simpler version using splash:jsfunc
    local scroll_to2 = splash:jsfunc("window.scrollTo")

    -- ...
end
```

See also: *splash:runjs*, *splash:jsfunc*, *splash:autoload*, *splash:wait_for_resume*.

splash:wait_for_resume

Run asynchronous JavaScript code in page context. The Lua script will yield until the JavaScript code tells it to resume.

Signature: `result, error = splash:wait_for_resume(snippet, timeout)`

Parameters:

- snippet - a string with a JavaScript source code to execute. This code must include a function called `main`. The first argument to `main` is an object that has the properties `resume` and `error`. `resume` is a function which can be used to resume Lua execution. It takes an optional argument which will be returned to Lua in the `result.value` return value. `error` is a function which can be called with a required string value that is returned in the `error` return value.
- timeout - a number which determines (in seconds) how long to allow JavaScript to execute before forceably returning control to Lua. Defaults to zero, which disables the timeout.

Returns: `result, error` pair. When the execution is successful `result` is a table. If the value returned by JavaScript is not `undefined`, then the `result` table will contain a key `value` that has the value passed to `splash.resume(...)`. The `result` table also contains any additional key/value pairs set by `splash.set(...)`. In case of timeout or JavaScript errors `result` is `nil` and `error` contains an error message string.

Async: yes.

Examples:

The first, trivial example shows how to transfer control of execution from Lua to JavaScript and then back to Lua. This command will tell JavaScript to sleep for 3 seconds and then return to Lua. Note that this is an async operation: the Lua event loop and the JavaScript event loop continue to run during this 3 second pause, but Lua will not continue executing the current function until JavaScript calls `splash.resume()`.

```
function main(splash)

    local result, error = splash:wait_for_resume([[
        function main(splash) {
            setTimeout(function () {
                splash.resume();
            }, 3000);
        }
    ]])

    -- result is {}
    -- error is nil

end
```

`result` is set to an empty table to indicate that nothing was returned from `splash.resume`. You can use `assert(splash:wait_for_resume(...))` even when JavaScript does not return a value because the empty table signifies success to `assert()`.

Note: Your JavaScript code must contain a `main()` function. You will get an error if you do not include it. The first argument to this function can have any name you choose, of course. We will call it `splash` by convention in this documentation.

The next example shows how to return a value from JavaScript to Lua. You can return booleans, numbers, strings, arrays, or objects.

```
function main(splash)

    local result, error = splash:wait_for_resume([[
        function main(splash) {
            setTimeout(function () {
                splash.resume([1, 2, 'red', 'blue']);
            }, 3000);
        }
    ]])
    -- result is {value={1, 2, 'red', 'blue'}}
    -- error is nil

end
```

Note: As with *splash:evaljs*, be wary of returning objects that are too large, such as the `$` object in jQuery, which will consume a lot of time and memory to convert to a Lua result.

You can also set additional key/value pairs in JavaScript with the `splash.set(key, value)` function. Key/value pairs will be included in the `result` table returned to Lua. The following example demonstrates this.

```
function main(splash)

    local result, error = splash:wait_for_resume([[
        function main(splash) {
            setTimeout(function () {
                splash.set("foo", "bar");
                splash.resume("ok");
            }, 3000);
        }
    ]])

    -- result is {foo="bar", value="ok"}
    -- error is nil

end
```

The next example shows an incorrect usage of `splash:wait_for_resume()`: the JavaScript code does not contain a `main()` function. `result` is nil because `splash.resume()` is never called, and `error` contains an error message explaining the mistake.

```
function main(splash)

    local result, error = splash:wait_for_resume([[
        console.log('hello!');
    ]])

    -- result is nil
    -- error is "error: wait_for_resume(): no main() function defined"

end
```

The next example shows error handling. If `splash.error(...)` is called instead of `splash.resume()`, then `result` will be `nil` and `error` will contain the string passed to `splash.error(...)`.

```
function main(splash)

    local result, error = splash:wait_for_resume([[
        function main(splash) {
            setTimeout(function () {
                splash.error("Goodbye, cruel world!");
            }, 3000);
        }
    ]])
    -- result is nil
    -- error is "error: Goodbye, cruel world!"

end
```

Your JavaScript code must either call `splash.resume()` or `splash.error()` exactly one time. Subsequent calls to either function have no effect, as shown in the next example.

```
function main(splash)

    local result, error = splash:wait_for_resume([[
        function main(splash) {
            setTimeout(function () {
                splash.resume("ok");
                splash.resume("still ok");
                splash.error("not ok");
            }, 3000);
        }
    ]])

    -- result is {value="ok"}
    -- error is nil

end
```

The next example shows the effect of the `timeout` argument. We have set the `timeout` argument to 1 second, but our JavaScript code will not call `splash.resume()` for 3 seconds, which guarantees that `splash:wait_for_resume()` will time out.

When it times out, `result` will be nil, `error` will contain a string explaining the timeout, and Lua will continue executing. Calling `splash.resume()` or `splash.error()` after a timeout has no effect.

```
function main(splash)

    local result, error = splash:wait_for_resume([[
        function main(splash) {
            setTimeout(function () {
                splash.resume("Hello, world!");
            }, 3000);
        }
    ]], 1)

    -- result is nil
    -- error is "error: One shot callback timed out while waiting for resume() or␣
↪error()."

end
```

Note: The timeout must be >= 0. If the timeout is 0, then `splash:wait_for_resume()` will never timeout (although Splash's HTTP timeout still applies).

Note that your JavaScript code is not forceably canceled by a timeout: it may continue to run until Splash shuts down the entire browser context.

See also: *splash:runjs*, *splash:jsfunc*, *splash:evaljs*.

splash:autoload

Set JavaScript to load automatically on each page load.

Signature: `ok, reason = splash:autoload{source_or_url, source=nil, url=nil}`

Parameters:

- source_or_url - either a string with JavaScript source code or an URL to load the JavaScript code from;
- source - a string with JavaScript source code;
- url - an URL to load JavaScript source code from.

Returns: `ok, reason` pair. If `ok` is nil, error happened and `reason` contains an error description.

Async: yes, but only when an URL of a remote resource is passed.

splash:autoload allows to execute JavaScript code at each page load. *splash:autoload* doesn't doesn't execute the passed JavaScript code itself. To execute some code once, *after* page is loaded use *splash:runjs* or *splash:jsfunc*.

splash:autoload can be used to preload utility JavaScript libraries or replace JavaScript objects before a webpage has a chance to do it.

Example:

```
function main(splash)
    splash:autoload([[
        function get_document_title(){
           return document.title;
        }
    ]])
    assert(splash:go(splash.args.url))
    return splash:evaljs("get_document_title()")
end
```

For the convenience, when a first *splash:autoload* argument starts with "http://" or "https://" a script from the passed URL is loaded. Example 2 - make sure a remote library is available:

```
function main(splash)
    assert(splash:autoload("https://code.jquery.com/jquery-2.1.3.min.js"))
    assert(splash:go(splash.args.url))
    local version = splash:evaljs("$.fn.jquery")
    return 'JQuery version: ' .. version
end
```

To disable URL auto-detection use 'source' and 'url' arguments:

```
splash:autoload{url="https://code.jquery.com/jquery-2.1.3.min.js"}
splash:autoload{source="window.foo = 'bar';"}
```

It is a good practice not to rely on auto-detection when the argument is not a constant.

If *splash:autoload* is called multiple times then all its scripts are executed on page load, in order they were added.

To revert Splash not to execute anything on page load use *splash:autoload_reset*.

See also: *splash:evaljs*, *splash:runjs*, *splash:jsfunc*, *splash:wait_for_resume*, *splash:autoload_reset*.

splash:autoload_reset

Unregister all scripts previously set by *splash:autoload*.

Signature: `splash:autoload_reset()`

Returns: nil

Async: no

After *splash:autoload_reset* call scripts set by *splash:autoload* won't be loaded in future requests; one can use *splash:autoload* again to setup a different set of scripts.

Already loaded scripts are not removed from the current page context.

See also: *splash:autoload*.

splash:call_later

Arrange for the callback to be called after the given delay seconds.

Signature: `timer = splash:call_later(callback, delay)`

Parameters:

- callback - function to run;
- delay - delay, in seconds;

Returns: a handle which allows to cancel pending timer or reraise exceptions happened in a callback.

Async: no.

Example 1 - take two HTML snapshots, at 1.5s and 2.5s after page loading starts:

```
function main(splash)
    local snapshots = {}
    local timer = splash:call_later(function()
        snapshots["a"] = splash:html()
        splash:wait(1.0)
        snapshots["b"] = splash:html()
    end, 1.5)
    assert(splash:go(splash.args.url))
    splash:wait(3.0)
    timer:reraise()
    return snapshots
end
```

splash:call_later returns a handle (a `timer`). To cancel pending task use its `timer:cancel()` method. If a callback is already started `timer:cancel()` has no effect.

By default, exceptions raised in *splash:call_later* callback stop the callback, but don't stop the main script. To reraise these errors use `timer:reraise()`.

splash:call_later arranges callback to be executed in future; it never runs it immediately, even if delay is 0. When delay is 0 callback is executed no earlier than current function yields to event loop, i.e. no earlier than some of the async functions is called.

splash:http_get

Send an HTTP GET request and return a response without loading the result to the browser window.

Signature: `response = splash:http_get{url, headers=nil, follow_redirects=true}`

Parameters:

- url - URL to load;
- headers - a Lua table with HTTP headers to add/replace in the initial request;
- follow_redirects - whether to follow HTTP redirects.

Returns: a *Response Object*.

Async: yes.

Example:

```
local reply = splash:http_get("http://example.com")
```

This method doesn't change the current page contents and URL. To load a webpage to the browser use *splash:go*.

See also: *splash:http_post*, *Response Object*.

splash:http_post

Send an HTTP POST request and return a response without loading the result to the browser window.

Signature: `response = splash:http_post{url, headers=nil, follow_redirects=true, body=nil}`

Parameters:

- url - URL to load;
- headers - a Lua table with HTTP headers to add/replace in the initial request;
- follow_redirects - whether to follow HTTP redirects.
- body - string with body of request, if you intend to send form submission, body should be urlencoded.

Returns: a *Response Object*.

Async: yes.

Example of form submission:

```
local reply = splash:http_post{url="http://example.com", body="user=Frank&
↪password=hunter2"}
-- reply.body contains raw HTML data (as a binary object)
-- reply.status contains HTTP status code, as a number
-- see Response docs for more info
```

Example of JSON POST request:

```
json = require("json")

local reply = splash:http_post{
    url="http://example.com/post",
    body=json.encode({alpha="beta"}),
    headers={["content-type"]="application/json"}
}
```

This method doesn't change the current page contents and URL. To load a webpage to the browser use *splash:go*.

See also: *splash:http_get*, *json*, *Response Object*.

splash:set_content

Set the content of the current page and wait until the page loads.

Signature: `ok, reason = splash:set_content{data, mime_type="text/html; charset=utf-8", baseurl=""}`

Parameters:

- data - new page content;
- mime_type - MIME type of the content;
- baseurl - external objects referenced in the content are located relative to baseurl.

Returns: `ok, reason` pair. If `ok` is nil then error happened during page load; `reason` provides an information about error type.

Async: yes.

Example:

```
function main(splash)
    assert(splash:set_content("<html><body><h1>hello</h1></body></html>"))
    return splash:png()
end
```

splash:html

Return a HTML snapshot of a current page (as a string).

Signature: `html = splash:html()`

Returns: contents of a current page (as a string).

Async: no.

Example:

```
-- A simplistic implementation of render.html endpoint
function main(splash)
    splash:set_result_content_type("text/html; charset=utf-8")
    assert(splash:go(splash.args.url))
    return splash:html()
end
```

Nothing prevents us from taking multiple HTML snapshots. For example, let's visit first 3 pages on a website, and for each page store initial HTML snapshot and an HTML snapshot after waiting 0.5s:

```
treat = require("treat")

-- Given an url, this function returns a table
-- with the page screenshoot, it's HTML contents
-- and it's title.
function page_info(splash, url)
    local ok, msg = splash:go(url)
    if not ok then
        return {ok=false, reason=msg}
    end
    local res = {
        html=splash:html(),
        title=splash:evaljs('document.title'),
        image=splash:png(),
        ok=true,
    }
    return res
end

-- visit first 3 pages of hacker news
local base = "https://news.ycombinator.com/news?p="
function main(splash)
    local result = treat.as_array({})
    for i=1,3 do
        local url =  base .. i
        result[i] = page_info(splash, url)
    end
    return result
end
```

splash:png

Return a *width x height* screenshot of a current page in PNG format.

Signature: `png = splash:png{width=nil, height=nil, render_all=false, scale_method='raster', region=nil}`

Parameters:

- width - optional, width of a screenshot in pixels;
- height - optional, height of a screenshot in pixels;
- render_all - optional, if `true` render the whole webpage;
- scale_method - optional, method to use when resizing the image, `'raster'` or `'vector'`;
- region - optional, `{left, top, right, bottom}` coordinates of a cropping rectangle.

Returns: PNG screenshot data, as a *binary object*. When the result is empty `nil` is returned.

Async: no.

Without arguments `splash:png()` will take a snapshot of the current viewport.

width parameter sets the width of the resulting image. If the viewport has a different width, the image is scaled up or down to match the specified one. For example, if the viewport is 1024px wide then `splash:png{width=100}` will return a screenshot of the whole viewport, but the image will be downscaled to 100px width.

height parameter sets the height of the resulting image. If the viewport has a different height, the image is trimmed or extended vertically to match the specified one without resizing the content. The region created by such extension is transparent.

To set the viewport size use *splash:set_viewport_size*, *splash:set_viewport_full* or *render_all* argument. `render_all=true` is equivalent to running `splash:set_viewport_full()` just before the rendering and restoring the viewport size afterwards.

To render an arbitrary part of a page use *region* parameter. It should be a table with `{left, top, right, bottom}` coordinates. Coordinates are relative to current scroll position. Currently you can't take anything which is not in a viewport; to make sure part of a page can be rendered call *splash:set_viewport_full* before using *splash:png* with *region*. This may be fixed in future Splash versions. With `region` and a bit of JavaScript it is possible to render only a single HTML element. Example:

```
-- this function adds padding around region
function pad(r, pad)
  return {r[1]-pad, r[2]-pad, r[3]+pad, r[4]+pad}
end

-- main script
function main(splash)

  -- this function returns element bounding box
  local get_bbox = splash:jsfunc([[
    function(css) {
      var el = document.querySelector(css);
      var r = el.getBoundingClientRect();
      return [r.left, r.top, r.right, r.bottom];
    }
  ]])

  assert(splash:go(splash.args.url))
  assert(splash:wait(0.5))

  -- don't crop image by a viewport
  splash:set_viewport_full()

  -- let's get a screenshot of a first <a>
  -- element on a page, with extra 32px around it
  local region = pad(get_bbox("a"), 32)
  return splash:png{region=region}
end
```

An easier way is to use *element:png* instead:

```
splash:select('#my-element'):png()
```

scale_method parameter must be either `'raster'` or `'vector'`. When `scale_method='raster'`, the image is resized per-pixel. When `scale_method='vector'`, the image is resized per-element during rendering. Vector scaling is more performant and produces sharper images, however it may cause rendering artifacts, so use it with caution.

The result of `splash:png` is a *binary object*, so you can return it directly from "main" function and it will be sent as a binary image data with a proper Content-Type header:

```
-- A simplistic implementation of render.png endpoint
function main(splash)
   assert(splash:go(splash.args.url))
   return splash:png{
      width=splash.args.width,
      height=splash.args.height
   }
end
```

If the result of `splash:png()` is returned as a table value, it is encoded to base64 to make it possible to embed in JSON and build a data:uri on a client (magic!):

```
function main(splash)
    assert(splash:go(splash.args.url))
    return {png=splash:png()}
end
```

When an image is empty *splash:png* returns `nil`. If you want Splash to raise an error in these cases use `assert`:

```
function main(splash)
    assert(splash:go(splash.args.url))
    local png = assert(splash:png())
    return {png=png}
end
```

See also: *splash:jpeg*, *Binary Objects*, *splash:set_viewport_size*, *splash:set_viewport_full*, *element:jpeg*, *element:png*.

splash:jpeg

Return a *width x height* screenshot of a current page in JPEG format.

Signature: `jpeg = splash:jpeg{width=nil, height=nil, render_all=false, scale_method='raster', quality=75, region=nil}`

Parameters:

- width - optional, width of a screenshot in pixels;
- height - optional, height of a screenshot in pixels;
- render_all - optional, if `true` render the whole webpage;
- scale_method - optional, method to use when resizing the image, `'raster'` or `'vector'`;
- quality - optional, quality of JPEG image, integer in range from `0` to `100`;
- region - optional, `{left, top, right, bottom}` coordinates of a cropping rectangle.

Returns: JPEG screenshot data, as a *binary object*. When the image is empty `nil` is returned.

Async: no.

Without arguments `splash:jpeg()` will take a snapshot of the current viewport.

width parameter sets the width of the resulting image. If the viewport has a different width, the image is scaled up or down to match the specified one. For example, if the viewport is 1024px wide then `splash:jpeg{width=100}` will return a screenshot of the whole viewport, but the image will be downscaled to 100px width.

height parameter sets the height of the resulting image. If the viewport has a different height, the image is trimmed or extended vertically to match the specified one without resizing the content. The region created by such extension is white.

To set the viewport size use *splash:set_viewport_size*, *splash:set_viewport_full* or *render_all* argument. `render_all=true` is equivalent to running `splash:set_viewport_full()` just before the rendering and restoring the viewport size afterwards.

To render an arbitrary part of a page use *region* parameter. It should be a table with `{left, top, right, bottom}` coordinates. Coordinates are relative to current scroll position. Currently you can't take anything which is not in a viewport; to make sure part of a page can be rendered call *splash:set_viewport_full* before using *splash:jpeg* with *region*. This may be fixed in future Splash versions.

With some JavaScript it is possible to render only a single HTML element using `region` parameter. See an *example* in *splash:png* docs. An alternative is to use *element:jpeg*.

scale_method parameter must be either `'raster'` or `'vector'`. When `scale_method='raster'`, the image is resized per-pixel. When `scale_method='vector'`, the image is resized per-element during rendering. Vector scaling is more performant and produces sharper images, however it may cause rendering artifacts, so use it with caution.

quality parameter must be an integer in range from `0` to `100`. Values above `95` should be avoided; `quality=100` disables portions of the JPEG compression algorithm, and results in large files with hardly any gain in image quality.

The result of `splash:jpeg` is a *binary object*, so you can return it directly from "main" function and it will be sent as a binary image data with a proper Content-Type header:

```
-- A simplistic implementation of render.jpeg endpoint
function main(splash)
    assert(splash:go(splash.args.url))
    return splash:jpeg{
       width=splash.args.width,
       height=splash.args.height
    }
end
```

If the result of `splash:jpeg()` is returned as a table value, it is encoded to base64 to make it possible to embed in JSON and build a data:uri on a client:

```
function main(splash)
    assert(splash:go(splash.args.url))
    return {jpeg=splash:jpeg()}
end
```

When an image is empty *splash:jpeg* returns `nil`. If you want Splash to raise an error in these cases use *assert*:

```
function main(splash)
    assert(splash:go(splash.args.url))
    local jpeg = assert(splash:jpeg())
    return {jpeg=jpeg}
end
```

See also: *splash:png*, *Binary Objects*, *splash:set_viewport_size*, *splash:set_viewport_full*, *element:jpeg*, *element:png*.

Note that `splash:jpeg()` is often 1.5..2x faster than `splash:png()`.

splash:har

Signature: `har = splash:har{reset=false}`

Parameters:

- reset - optional; when `true`, reset HAR records after taking a snapshot.

Returns: information about pages loaded, events happened, network requests sent and responses received in HAR format.

Async: no.

Use *splash:har* to get information about network requests and other Splash activity.

If your script returns the result of `splash:har()` in a top-level `"har"` key then Splash UI will give you a nice diagram with network information (similar to "Network" tabs in Firefox or Chrome developer tools):

```
function main(splash)
    assert(splash:go(splash.args.url))
    return {har=splash:har()}
end
```

By default, when several requests are made (e.g. *splash:go* is called multiple times), HAR data is accumulated and combined into a single object (logs are still grouped by page).

If you want only updated information use `reset` parameter: it drops all existing logs and start recording from scratch:

```
function main(splash)
    assert(splash:go(splash.args.url1))
    local har1 = splash:har{reset=true}
    assert(splash:go(splash.args.url2))
    local har2 = splash:har()
    return {har1=har1, har2=har2}
end
```

By default, response content is not returned in HAR data. To enable it, use *splash.response_body_enabled* option or *request:enable_response_body* method.

See also: *splash:har_reset*, *splash:on_response*, *splash.response_body_enabled*, *request:enable_response_body*.

splash:har_reset

Signature: `splash:har_reset()`

Returns: nil.

Async: no.

Drops all internally stored HAR records. It is similar to `splash:har{reset=true}`, but doesn't return anything.

See also: *splash:har*.

splash:history

Signature: `entries = splash:history()`

Returns: information about requests/responses for the pages loaded, in HAR entries format.

Async: no.

`splash:history` doesn't return information about related resources like images, scripts, stylesheets or AJAX requests. If you need this information use *splash:har* or *splash:on_response*.

Let's get a JSON array with HTTP headers of the response we're displaying:

```
function main(splash)
    assert(splash:go(splash.args.url))
    local entries = splash:history()
    -- #entries means "entries length"; arrays in Lua start from 1
    local last_entry = entries[#entries]
    return {
       headers = last_entry.response.headers
    }
end
```

See also: *splash:har*, *splash:on_response*.

splash:url

Signature: `url = splash:url()`

Returns: the current URL.

Async: no.

splash:get_cookies

Signature: `cookies = splash:get_cookies()`

Returns: CookieJar contents - an array with all cookies available for the script. The result is returned in HAR cookies format.

Async: no.

Example result:

```
[
    {
        "name": "TestCookie",
        "value": "Cookie Value",
        "path": "/",
        "domain": "www.example.com",
        "expires": "2016-07-24T19:20:30+02:00",
        "httpOnly": false,
        "secure": false,
    }
]
```

splash:add_cookie

Add a cookie.

Signature: `cookies = splash:add_cookie{name, value, path=nil, domain=nil, expires=nil, httpOnly=nil, secure=nil}`

Async: no.

Example:

```
function main(splash)
    splash:add_cookie{"sessionid", "237465ghgfsd", "/", domain="http://example.com"}
    splash:go("http://example.com/")
    return splash:html()
end
```

splash:init_cookies

Replace all current cookies with the passed `cookies`.

Signature: `splash:init_cookies(cookies)`

Parameters:

- cookies - a Lua table with all cookies to set, in the same format as *splash:get_cookies* returns.

Returns: nil.

Async: no.

Example 1 - save and restore cookies:

```
local cookies = splash:get_cookies()
-- ... do something ...
splash:init_cookies(cookies)  -- restore cookies
```

Example 2 - initialize cookies manually:

```
splash:init_cookies({
    {name="baz", value="egg"},
    {name="spam", value="egg", domain="example.com"},
    {
        name="foo",
        value="bar",
        path="/",
        domain="localhost",
        expires="2016-07-24T19:20:30+02:00",
        secure=true,
        httpOnly=true,
    }
})

-- do something
assert(splash:go("http://example.com"))
```

splash:clear_cookies

Clear all cookies.

Signature: `n_removed = splash:clear_cookies()`

Returns: a number of cookies deleted.

Async: no.

To delete only specific cookies use *splash:delete_cookies*.

splash:delete_cookies

Delete matching cookies.

Signature: `n_removed = splash:delete_cookies{name=nil, url=nil}`

Parameters:

- name - a string, optional. All cookies with this name will be deleted.
- url - a string, optional. Only cookies that should be sent to this url will be deleted.

Returns: a number of cookies deleted.

Async: no.

This function does nothing when both *name* and *url* are nil. To remove all cookies use *splash:clear_cookies* method.

splash:lock_navigation

Lock navigation.

Signature: `splash:lock_navigation()`

Async: no.

After calling this method the navigation away from the current page is no longer permitted - the page is locked to the current URL.

splash:unlock_navigation

Unlock navigation.

Signature: `splash:unlock_navigation()`

Async: no.

After calling this method the navigation away from the page becomes permitted. Note that the pending navigation requests suppressed by *splash:lock_navigation* won't be reissued.

splash:set_result_status_code

Set HTTP status code of a result returned to a client.

Signature: `splash:set_result_status_code(code)`

Parameters:

- code - HTTP status code (a number 200 <= code <= 999).

Returns: nil.

Async: no.

Use this function to signal errors or other conditions to splash client using HTTP status codes.

Example:

```
function main(splash)
    local ok, reason = splash:go("http://www.example.com")
    if reason == "http500" then
        splash:set_result_status_code(503)
        splash:set_result_header("Retry-After", 10)
        return ''
    end
    return splash:png()
end
```

Be careful with this function: some proxies can be configured to process responses differently based on their status codes. See e.g. nginx proxy_next_upstream option.

In case of unhandled Lua errors HTTP status code is set to 400 regardless of the value set with *splash:set_result_status_code*.

See also: *splash:set_result_content_type*, *splash:set_result_header*.

splash:set_result_content_type

Set Content-Type of a result returned to a client.

Signature: `splash:set_result_content_type(content_type)`

Parameters:

- content_type - a string with Content-Type header value.

Returns: nil.

Async: no.

If a table is returned by "main" function then `splash:set_result_content_type` has no effect: Content-Type of the result is set to `application/json`.

This function **does not** set Content-Type header for requests initiated by *splash:go*; this function is for setting Content-Type header of a result.

Example:

```
function main(splash)
    splash:set_result_content_type("text/xml")
    return [[
       <?xml version="1.0" encoding="UTF-8"?>
       <note>
           <to>Tove</to>
           <from>Jani</from>
           <heading>Reminder</heading>
           <body>Don't forget me this weekend!</body>
       </note>
    ]]
end
```

See also:

- *splash:set_result_header* which allows to set any custom response header, not only Content-Type.
- *Binary Objects* which have their own method for setting result Content-Type.

splash:set_result_header

Set header of result response returned to splash client.

Signature: `splash:set_result_header(name, value)`

Parameters:

- name of response header
- value of response header

Returns: nil.

Async: no.

This function **does not** set HTTP headers for responses returned by *splash:go* or requests initiated by *splash:go*; this function is for setting headers of splash response sent to client.

Example 1, set 'foo=bar' header:

```
function main(splash)
    splash:set_result_header("foo", "bar")
    return "hello"
end
```

Example 2, measure the time needed to build PNG screenshot and return it result in an HTTP header:

```
function main(splash)

    -- this function measures the time code takes to execute and returns
    -- it in an HTTP header
    function timeit(header_name, func)
        local start_time = splash:get_perf_stats().walltime
        local result = func()  -- it won't work for multiple returned values!
        local end_time = splash:get_perf_stats().walltime
        splash:set_result_header(header_name, tostring(end_time - start_time))
        return result
    end

    -- rendering script
    assert(splash:go(splash.args.url))
    local screenshot = timeit("X-Render-Time", function()
       return splash:png()
    end)
    splash:set_result_content_type("image/png")
    return screenshot
end
```

See also: *splash:set_result_status_code*, *splash:set_result_content_type*.

splash:get_viewport_size

Get the browser viewport size.

Signature: `width, height = splash:get_viewport_size()`

Returns: two numbers: width and height of the viewport in pixels.

Async: no.

splash:set_viewport_size

Set the browser viewport size.

Signature: `splash:set_viewport_size(width, height)`

Parameters:

- width - integer, requested viewport width in pixels;
- height - integer, requested viewport height in pixels.

Returns: nil.

Async: no.

This will change the size of the visible area and subsequent rendering commands, e.g., *splash:png*, will produce an image with the specified size.

splash:png uses the viewport size.

Example:

```
function main(splash)
    splash:set_viewport_size(1980, 1020)
    assert(splash:go("http://example.com"))
    return {png=splash:png()}
end
```

Note: This will relayout all document elements and affect geometry variables, such as `window.innerWidth` and `window.innerHeight`. However `window.onresize` event callback will only be invoked during the next asynchronous operation and *splash:png* is notably synchronous, so if you have resized a page and want it to react accordingly before taking the screenshot, use *splash:wait*.

splash:set_viewport_full

Resize browser viewport to fit the whole page.

Signature: `width, height = splash:set_viewport_full()`

Returns: two numbers: width and height the viewport is set to, in pixels.

Async: no.

`splash:set_viewport_full` should be called only after page is loaded, and some time passed after that (use *splash:wait*). This is an unfortunate restriction, but it seems that this is the only way to make automatic resizing work reliably.

See *splash:set_viewport_size* for a note about interaction with JS.

splash:png uses the viewport size.

Example:

```lua
function main(splash)
    assert(splash:go("http://example.com"))
    assert(splash:wait(0.5))
    splash:set_viewport_full()
    return {png=splash:png()}
end
```

splash:set_user_agent

Overwrite the User-Agent header for all further requests.

Signature: `splash:set_user_agent(value)`

Parameters:

- value - string, a value of User-Agent HTTP header.

Returns: nil.

Async: no.

splash:set_custom_headers

Set custom HTTP headers to send with each request.

Signature: `splash:set_custom_headers(headers)`

Parameters:

- headers - a Lua table with HTTP headers.

Returns: nil.

Async: no.

Headers are merged with WebKit default headers, overwriting WebKit values in case of conflicts.

When `headers` argument of *splash:go* is used headers set with `splash:set_custom_headers` are not applied to the initial request: values are not merged, `headers` argument of *splash:go* has higher priority.

Example:

```
splash:set_custom_headers({
   ["Header-1"] = "Value 1",
   ["Header-2"] = "Value 2",
})
```

Note: Named arguments are not supported for this function.

See also: *splash:on_request*.

splash:get_perf_stats

Return performance-related statistics.

Signature: `stats = splash:get_perf_stats()`

Returns: a table that can be useful for performance analysis.

Async: no.

As of now, this table contains:

- `walltime` - (float) number of seconds since epoch, analog of `os.clock`
- `cputime` - (float) number of cpu seconds consumed by splash process
- `maxrss` - (int) high water mark number of bytes of RAM consumed by splash process

splash:on_request

Register a function to be called before each HTTP request.

Signature: `splash:on_request(callback)`

Parameters:

- callback - Lua function to call before each HTTP request.

Returns: nil.

Async: no.

splash:on_request callback receives a single `request` argument (a *Request Object*).

To get information about a request use request *attributes*; to change or drop the request before sending use request *methods*;

A callback passed to *splash:on_request* can't call Splash async methods like *splash:wait* or *splash:go*.

Example 1 - log all URLs requested using *request.url* attribute:

```
treat = require("treat")
function main(splash)
    local urls = {}
    splash:on_request(function(request)
        table.insert(urls, request.url)
    end)
    assert(splash:go(splash.args.url))
    return treat.as_array(urls)
end
```

Example 2 - to log full request information use *request.info* attribute; don't store `request` objects directly:

```
treat = require("treat")
function main(splash)
    local entries = treat.as_array({})
    splash:on_request(function(request)
        table.insert(entries, request.info)
    end)
    assert(splash:go(splash.args.url))
    return entries
end
```

Example 3 - drop all requests to resources containing ".css" in their URLs (see *request:abort*):

```
splash:on_request(function(request)
    if string.find(request.url, ".css") ~= nil then
        request.abort()
    end
end)
```

Example 4 - replace a resource (see *request:set_url*):

```
splash:on_request(function(request)
    if request.url == 'http://example.com/script.js' then
        request:set_url('http://mydomain.com/myscript.js')
    end
end)
```

Example 5 - set a custom proxy server, with credentials passed in an HTTP request to Splash (see *request:set_proxy*):

```
splash:on_request(function(request)
    request:set_proxy{
        host = "0.0.0.0",
        port = 8990,
        username = splash.args.username,
        password = splash.args.password,
    }
end)
```

Example 6 - discard requests which take longer than 5 seconds to complete, but allow up to 15 seconds for the first request (see *request:set_timeout*):

```
local first = true
splash.resource_timeout = 5
splash:on_request(function(request)
    if first then
        request:set_timeout(15.0)
        first = false
    end
end)
```

Note: *splash:on_request* doesn't support named arguments.

See also: *splash:on_response*, *splash:on_response_headers*, *splash:on_request_reset*, *treat*, *Request Object*.

splash:on_response_headers

Register a function to be called after response headers are received, before response body is read.

Signature: `splash:on_response_headers(callback)`

Parameters:

- callback - Lua function to call for each response after response headers are received.

Returns: nil.

Async: no.

splash:on_response_headers callback receives a single `response` argument (a *Response Object*).

response.body is not available in a *splash:on_response_headers* callback because response body is not read yet. That's the point of *splash:on_response_headers* method: you can abort reading of the response body using *response:abort* method.

A callback passed to *splash:on_response_headers* can't call Splash async methods like *splash:wait* or *splash:go*. `response` object is deleted after exiting from a callback, so you cannot use it outside a callback.

Example 1 - log content-type headers of all responses received while rendering

```
function main(splash)
    local all_headers = {}
    splash:on_response_headers(function(response)
        local content_type = response.headers["Content-Type"]
        all_headers[response.url] = content_type
    end)
    assert(splash:go(splash.args.url))
    return all_headers
end
```

Example 2 - abort reading body of all responses with content type `text/css`

```
function main(splash)
    splash:on_response_headers(function(response)
        local content_type = response.headers["Content-Type"]
        if content_type == "text/css" then
            response.abort()
        end
    end)
    assert(splash:go(splash.args.url))
    return splash:png()
end
```

Example 3 - extract all cookies set by website without downloading response bodies

```
function main(splash)
    local cookies = ""
    splash:on_response_headers(function(response)
        local response_cookies = response.headers["Set-cookie"]
        cookies = cookies .. ";" .. response_cookies
        response.abort()
    end)
    assert(splash:go(splash.args.url))
    return cookies
end
```

Note: *splash:on_response_headers* doesn't support named arguments.

See also: *splash:on_request*, *splash:on_response*, *splash:on_response_headers_reset*, *Response Object*.

splash:on_response

Register a function to be called after response is downloaded.

Signature: `splash:on_response(callback)`

Parameters:

- callback - Lua function to call for each response after it is downloaded.

Returns: nil.

Async: no.

splash:on_response callback receives a single `response` argument (a *Response Object*).

By default, this `response` object doesn't have *response.body* attribute. To enable it, use *splash.response_body_enabled* option or *request:enable_response_body* method.

Note: *splash:on_response* doesn't support named arguments.

See also: *splash:on_request*, *splash:on_response_headers*, *splash:on_response_reset*, *Response Object*, *splash.response_body_enabled*, *request:enable_response_body*.

splash:on_request_reset

Remove all callbacks registered by *splash:on_request*.

Signature: `splash:on_request_reset()`

Returns: nil

Async: no.

splash:on_response_headers_reset

Remove all callbacks registered by *splash:on_response_headers*.

Signature: `splash:on_response_headers_reset()`

Returns: nil

Async: no.

splash:on_response_reset

Remove all callbacks registered by *splash:on_response*.

Signature: `splash:on_response_reset()`

Returns: nil

Async: no.

splash:get_version

Get Splash major and minor version.

Signature: `version_info = splash:get_version()`

Returns: A table with version information.

Async: no.

As of now, this table contains:

- `splash` - (string) Splash version
- `major` - (int) Splash major version
- `minor` - (int) Splash minor version
- `python` - (string) Python version
- `qt` - (string) Qt version
- `pyqt` - (string) PyQt version
- `webkit` - (string) WebKit version
- `sip` - (string) SIP version
- `twisted` - (string) Twisted version

Example:

```
function main(splash)
     local version = splash:get_version()
     if version.major < 2 and version.minor < 8 then
         error("Splash 1.8 or newer required")
     end
 end
```

splash:mouse_click

Trigger mouse click event in web page.

Signature: `splash:mouse_click(x, y)`

Parameters:

- x - number with x position of element to be clicked (distance from the left, relative to the current viewport)
- y - number with y position of element to be clicked (distance from the top, relative to the current viewport)

Returns: nil

Async: no.

Coordinates for mouse events must be relative to viewport. Element on which action is performed must be inside viewport (must be visible to the user). If element is outside viewport and user needs to scroll to see it, you must either scroll to the element with JavaScript or set viewport to full with *splash:set_viewport_full*.

Mouse events are not propagated immediately, to see consequences of click reflected in page source you must call *splash:wait*.

If you want to click on element an easy way is to use *splash:select* with *element:mouse_click*:

```
local button = splash:select('button')
button:mouse_click()
```

You also can implement it using *splash:mouse_click*; use JavaScript getClientRects to get coordinates of html element:

```
-- Get button element dimensions with javascript and perform mouse click.
function main(splash)
    assert(splash:go(splash.args.url))
    local get_dimensions = splash:jsfunc([[
        function () {
            var rect = document.getElementById('button').getClientRects()[0];
            return {"x": rect.left, "y": rect.top}
        }
    ]])
    splash:set_viewport_full()
    splash:wait(0.1)
    local dimensions = get_dimensions()
    splash:mouse_click(dimensions.x, dimensions.y)

    -- Wait split second to allow event to propagate.
    splash:wait(0.1)
    return splash:html()
end
```

Under the hood *splash:mouse_click* performs *splash:mouse_press* followed by *splash:mouse_release*.

At the moment only left click is supported.

See also: *element:mouse_click*, *splash:mouse_press*, *splash:mouse_release*, *splash:mouse_hover*, .

splash:mouse_hover

Trigger mouse hover (JavaScript mouseover) event in web page.

Signature: `splash:mouse_hover(x, y)`

Parameters:

- x - number with x position of element to be hovered on (distance from the left, relative to the current viewport)
- y - number with y position of element to be hovered on (distance from the top, relative to the current viewport)

Returns: nil

Async: no.

See notes about mouse events in *splash:mouse_click*.

See also: *element:mouse_hover*.

splash:mouse_press

Trigger mouse press event in web page.

Signature: `splash:mouse_press(x, y)`

Parameters:

- x - number with x position of element over which mouse button is pressed (distance from the left, relative to the current viewport)
- y - number with y position of element over which mouse button is pressed (distance from the top, relative to the current viewport)

Returns: nil

Async: no.

See notes about mouse events in *splash:mouse_click*.

splash:mouse_release

Trigger mouse release event in web page.

Signature: `splash:mouse_release(x, y)`

Parameters:

- x - number with x position of element over which mouse button is released (distance from the left, relative to the current viewport)
- y - number with y position of element over which mouse button is released (distance from the top, relative to the current viewport)

Returns: nil

Async: no.

See notes about mouse events in *splash:mouse_click*.

splash:with_timeout

Run the function with the allowed timeout

Signature: `ok, result = splash:with_timeout(func, timeout)`

Parameters:

- func - the function to run
- timeout - timeout, in seconds

Returns: `ok, result` pair. If `ok` is not `true` then error happened during the function call or the timeout expired; `result` provides an information about error type. If `result` is equal to `timeout` then the specified timeout period elapsed. Otherwise, if `ok` is `true` then `result` contains the result of the executed function. If your function returns several values, they will be assigned to the next variables to `result`.

Async: yes.

Example 1:

```
function main(splash)
  local ok, result = splash:with_timeout(function()
    local url = splash.args.url
    splash:wait(3)
    assert(splash:go(url))
  end, 2)

  if not ok then
    if result == "timeout_over" then
      return "Cannot navigate to the url within 2 seconds"
    else
      return result
    end
  end

  return "Navigated to the url within 2 seconds"
end
```

Example 2 - the function returns several values

```
function main(splash)
    local ok, result1, result2, result3 = splash:with_timeout(function()
        splash:wait(0.5)
        return 1, 2, 3
    end, 1)

    return result1, result2, result3
end
```

Note that if the specified timeout period elapsed Splash will try to interrupt the running function. However, Splash scripts are executed in cooperative multitasking manner and because of that sometimes Splash won't be able to stop your running function upon timeout expiration. In two words, cooperative multitasking means that the managing program (in our example, it is Splash scripting engine) won't stop the running function if it doesn't *ask* for that. In Splash scripting the running function can be interrupted only if some *async* operation was called. On the contrary, non of the *sync* operations can be interrupted.

Note: Splash scripts are executing in cooperative multitasking manner. You should be careful while running sync functions.

Let's see the difference in examples.

Example 3:

```
function main(splash)
    local ok, result = splash:with_timeout(function()
        splash:go(splash.args.url) -- during this operation the current function can␣
↪be stopped
        splash:evaljs(long_js_operation) -- during JS function evaluation the␣
↪function cannot be stopped
        local png = splash:png() -- sync operation and during it the function cannot␣
↪be stopped
        return png
    end, 0.1)

    return result
end
```

splash:send_keys

Send keyboard events to page context.

Signature: `splash:send_keys(keys)`

Parameters

- keys - string representing the keys to be sent as keyboard events.

Returns: nil

Async: no.

Key sequences are specified by using a small subset of emacs edmacro syntax:

- whitespace is ignored and only used to separate the different keys
- characters are literally represented
- words within brackets represent function keys, like `<Return>`, `<Left>` or `<Home>`. See Qt docs for a full list of function keys. `<Foo>` will try to match `Qt::Key_Foo`.

Following table shows some examples of macros and what they would generate on an input:

Macro	Result
`Hello World`	`HelloWorld`
`Hello <Space> World`	`Hello World`
`< S p a c e >`	`<Space>`
`Hello <Home> <Delete>`	`ello`
`Hello <Backspace>`	`Hell`

Key events are not propagated immediately until event loop regains control, thus *splash:wait* must be called to reflect the events.

See also: *element:send_keys*, *splash:send_text*.

splash:send_text

Send text as input to page context, literally, character by character.

Signature: `splash:send_text(text)`

Parameters:

- text - string to be sent as input.

Returns: nil

Async: no.

Key events are not propagated immediately until event loop regains control, thus *splash:wait* must be called to reflect the events.

This function in conjuction with *splash:send_keys* covers most needs on keyboard input, such as filling in forms and submitting them.

Example 1: focus first input, fill in a form and submit

```
function main(splash)
    assert(splash:go(splash.args.url))
    assert(splash:wait(0.5))
    splash:send_keys("<Tab>")
    splash:send_text("zero cool")
    splash:send_keys("<Tab>")
    splash:send_text("hunter2")
    splash:send_keys("<Return>")
    -- note how this could be translated to
    -- splash:send_keys("<Tab> zero <Space> cool <Tab> hunter2 <Return>")
    assert(splash:wait(0))
    -- ...
end
```

Example 2: focus inputs with javascript or *splash:mouse_click*

We can't always assume that a <*Tab*> will focus the input we want or an <*Enter*> will submit a form. Selecting an input can either be accomplished by focusing it or by clicking it. Submitting a form can also be done by firing a submit event on the form, or simply by clicking on the submit button.

The following example will focus an input, fill in a form and click on the submit button using *splash:mouse_click*. It assumes there are two arguments passed to splash, *username* and *password*.

```
function main(splash)
    function focus(sel)
        splash:select(sel).node:focus()
    end

    assert(splash:go(splash.args.url))
    assert(splash:wait(0.5))
    focus('input[name=username]')
    splash:send_text(splash.args.username)
    assert(splash:wait(0))
    focus('input[name=password]')
    splash:send_text(splash.args.password)
    splash:select('input[type=submit]'):mouse_click()
    assert(splash:wait(0))
    -- Usually, wait for the submit request to finish
    -- ...
end
```

See also: *element:send_text*, *splash:send_keys*.

splash:select

Select the first HTML element from DOM of the current web page that matches the specified CSS selector.

Signature: `element = splash:select(selector)`

Parameters:

- selector - valid CSS selector

Returns: an *Element* object.

Async: no.

Using *splash:select* you can get the element that matches your specified CSS selector like using document.querySelector in the browser. The returned element is an *Element Object* which has many useful methods and almost all methods and attributes that element has in JavaScript.

If the element cannot be found using the specified selector `nil` will be returned. If your selector is not a valid CSS selector an error will be raised.

Example 1: select an element which has `element` class and return class names off all the siblings of the specified element.

```
local treat = require('treat')

function main(splash)
    assert(splash:go(splash.args.url))
    assert(splash:wait(0.5))

    local el = splash:select('.element')
    local seen = {}
    local classNames = {}

    while el do
      local classList = el.node.classList
      if classList then
        for _, v in ipairs(classList) do
          if (not seen[v]) then
            classNames[#classNames + 1] = v
            seen[v] = true
          end
        end
      end

      el = el.node.nextSibling
    end

    return treat.as_array(classNames)
end
```

Example 2: assert that the returned element exists

```
function main(splash)
    -- ...
    local el = assert(splash:select('.element'))
    -- ...
end
```

splash:select_all

Select the list of HTML elements from DOM of the current web page that match the specified CSS selector.

Signature: `elements = splash:select_all(selector)`

Parameters:

- selector - valid CSS selector

Returns: a list of *Element* objects.

Async: no.

This method differs from *splash:select* by returning the *all* elements in a table that match the specified selector.

If no elements can be found using the specified selector `{}` is returned. If the selector is not a valid CSS selector an error is raised.

Example: select all `<img />` elements and get their `src` attributes

```lua
local treat = require('treat')

function main(splash)
    assert(splash:go(splash.args.url))
    assert(splash:wait(0.5))

    local imgs = splash:select_all('img')
    local srcs = {}

    for _, img in ipairs(imgs) do
      srcs[#srcs+1] = img.node.attributes.src
    end

    return treat.as_array(srcs)
end
```

Response Object

Response objects are returned as a result of several Splash methods (like *splash:http_get* or *splash:http_post*); they are are also passed to some of the callbacks (e.g. *splash:on_response* and *splash:on_response_headers* callbacks). These objects contain information about a response.

response.url

URL of the response. In case of redirects *response.url* is a last URL.

This field is read-only.

response.status

HTTP status code of the response.

This field is read-only.

response.ok

`true` for successful responses and `false` when error happened.

Example:

```lua
local reply = splash:http_get("some-bad-url")
-- reply.ok == false
```

This field is read-only.

response.headers

A Lua table with HTTP headers (header name => header value). Keys are header names (strings), values are header values (strings).

Lookups are case-insensitive, so `response.headers['content-type']` is the same as `response.headers['Content-Type']`.

This field is read-only.

response.info

A Lua table with response data in HAR response format.

This field is read-only.

response.body

Raw response body (a *binary object*).

If you want to process response body from Lua use *treat.as_string* to convert it to a Lua string first.

response.body attribute is not available by default in *splash:on_response* callbacks; use *splash.response_body_enabled* or *request:enable_response_body* to enable it.

response.request

A corresponding *Request Object*.

This field is read-only.

response:abort

Signature: `response:abort()`

Returns: nil.

Async: no.

Abort reading of the response body. This method is only available if a response is not read yet - currently you can use it only in a *splash:on_response_headers* callback.

Request Object

Request objects are received by *splash:on_request* callbacks; they are also available as *response.request*.

Attributes

Request objects has several attributes with information about a HTTP request. These fields are for information only; changing them doesn't change the request to be sent.

request.url

Requested URL.

request.method

HTTP method name in upper case, e.g. "GET".

request.headers

A Lua table with request HTTP headers (header name => header value). Keys are header names (strings), values are header values (strings).

Lookups are case-insensitive, so `request.headers['content-type']` is the same as `request.headers['Content-Type']`.

request.info

A table with request data in HAR request format.

Methods

To change or drop the request before sending use one of the `request` methods. Note that these methods are only available before the request is sent (they has no effect if a request is already sent). Currently it means you can only use them in *splash:on_request* callbacks.

request:abort

Drop the request.

Signature: `request:abort()`

Returns: nil.

Async: no.

request:enable_response_body

Enable tracking of response content (i.e. *response.body* attribute).

Signature: `request:enable_response_body()`

Returns: nil.

Async: no.

This function allows to enable response content tracking per-request when *splash.response_body_enabled* is set to false. Call it in a *splash:on_request* callback.

request:set_url

Change request URL to a specified value.

Signature: `request:set_url(url)`

Parameters:

- url - new request URL

Returns: nil.

Async: no.

request:set_proxy

Set a proxy server to use for this request.

Signature: `request:set_proxy{host, port, username=nil, password=nil, type='HTTP'}`

Parameters:

- host
- port
- username
- password
- type - proxy type; allowed proxy types are 'HTTP' and 'SOCKS5'.

Returns: nil.

Async: no.

Omit `username` and `password` arguments if a proxy doesn't need auth.

When `type` is set to 'HTTP' HTTPS proxying should also work; it is implemented using CONNECT command.

request:set_timeout

Set a timeout for this request.

Signature: `request:set_timeout(timeout)`

Parameters:

- timeout - timeout value, in seconds.

Returns: nil.

Async: no.

If response is not fully received after the timeout, request is aborted. See also: *splash.resource_timeout*.

request:set_header

Set an HTTP header for this request.

Signature: `request:set_header(name, value)`

Parameters:

- name - header name;
- value - header value.

Returns: nil.

Async: no.

See also: *splash:set_custom_headers*

Element Object

Element objects wrap JavaScript DOM nodes. They are created whenever some method returns any type of DOM node (Node, Element, HTMLElement, etc).

splash:select and *splash:select_all* return element objects; *splash:evaljs* may also return element objects, but currently they can't be inside other objects or arrays - only top-level Node and NodeList is supported.

Attributes

element.node

`element.node` is a object that contains almost all DOM element attributes and methods.

The list of supported properties (some of them are mutable, other are read-only):

Properties inherited from HTMLElement:

- accessKey
- accessKeyLabel *(read-only)*
- contentEditable
- isContentEditable *(read-only)*
- dataset *(read-only)*
- dir
- draggable
- hidden
- lang
- offsetHeight *(read-only)*
- offsetLeft *(read-only)*
- offsetParent *(read-only)*
- offsetTop *(read-only)*
- spellcheck
- style - a table with styles which can be modified
- tabIndex
- title
- translate

Properties inherited from Element:

- attributes *(read-only)* - a table with attributes of the element
- classList *(read-only)* - a table with class names of the element
- className
- clientHeight *(read-only)*
- clientLeft *(read-only)*

- clientTop *(read-only)*
- clientWidth *(read-only)*
- id
- innerHTML
- localeName *(read-only)*
- namespaceURI *(read-only)*
- nextElementSibling *(read-only)*
- outerHTML
- prefix *(read-only)*
- previousElementSibling *(read-only)*
- scrollHeight *(read-only)*
- scrollLeft
- scrollTop
- scrollWidth *(read-only)*
- tabStop
- tagName *(read-only)*

Properties inherited from Node:

- baseURI *(read-only)*
- childNodes *(read-only)*
- firstChild *(read-only)*
- lastChild *(read-only)*
- nextSibling *(read-only)*
- nodeName *(read-only)*
- nodeType *(read-only)*
- nodeValue
- ownerDocument *(read-only)*
- parentNode *(read-only)*
- parentElement *(read-only)*
- previousSibling *(read-only)*
- rootNode *(read-only)*
- textContent

The list of supported methods:

Methods inherited from EventTarget:

- addEventListener
- removeEventListener

Methods inherited from HTMLElement:

- blur
- click
- focus

Methods inherited from Element:

- getAttribute
- getAttributeNS
- getBoundingClientRect
- getClientRects
- getElementsByClassName
- getElementsByTagName
- getElementsByTagNameNS
- hasAttribute
- hasAttributeNS
- hasAttributes
- querySelector
- querySelectorAll
- releasePointerCapture
- remove
- removeAttribute
- removeAttributeNS
- requestFullscreen
- requestPointerLock
- scrollIntoView
- setAttribute
- setAttributeNS
- setPointerCapture

Methods inherited from Node:

- appendChild
- cloneNode
- compareDocumentPosition
- contains
- hasChildNodes
- insertBefore
- isDefaultNamespace
- isEqualNode
- isSameNode

- lookupPrefix
- lookupNamespaceURI
- normalize
- removeChild
- replaceChild

Also, you can attach event handlers to the specified event. When the handler is called it will receive `event` table with the almost all available methods and properties.

```
function main(splash)
    -- ...
    local element = splash:select('.element')

    local x, y = 0, 0

    element.onclick = function(event)
        event:preventDefault()
        x = event.clientX
        y = event.clientY
    end

    assert(splash:wait(10))

    return x, y
end
```

The another way to attach event handlers is to use `element.node:addEventListener(event, listener)`. It allows you to add more than a single event handler for an event.

Example of using `element.node:addEventListener(event, listener)`

```
function main(splash)
    -- ...
    local element = splash:select('.element')

    local x, y = 0, 0

    local store_coordinates = function(event)
        x = event.clientX
        y = event.clientY
    end

    element.node:addEventListener('click', store_coordinates)

    assert(splash:wait(10))

    return x, y
end
```

The following fields are read-only.

element.inner_id

Id of the inner representation of the element. It may be useful for comparing the elements for the equality.

Example:

```
function main(splash)
    -- ...

    local same = element2.inner_id == element2.inner_id

    -- ...
end
```

Methods

To modify or retrieve some information about the element you can use the following methods.

element:exists

Check whether the element exists in DOM. If the element doesn't exist some of the methods will fail, returning the error flag.

Signature: `exists = element:exists()`

Returns: `exists` indicated whether the element exists.

Async: no.

There are several reasons why the element can be absent from DOM. One of the reasons is that the element was removed by some JavaScript code.

Example 1: the element was removed by JS code

```
function main(splash)
    -- ...
    local element = splash:select('.element')
    assert(splash:runjs('document.write("<body></body>")'))
    assert(splash:wait(0.1))
    local exists = element:exists() -- exists will be `false`
    -- ...
end
```

Another reason is that the element was created by script and not inserted into DOM.

Example 2: the element is not inserted into DOM

```
function main(splash)
    -- ...
    local element = splash:select('.element')
    local cloned = element.node:cloneNode() -- the cloned element isn't in DOM
    local exists = cloned:exists() -- exists will be `false`
    -- ...
end
```

element:mouse_click

Trigger mouse click event on the element.

Signature: `ok, reason = element:mouse_click{x=0, y=0}`

Parameters:

- x - optional, x coordinate relative to the left corner of the element
- y - optional, y coordinate relative to the top corner of the element

Returns: `ok, reason` pair. If `ok` is nil then error happened during the function call; `reason` provides an information about error type.

Async: no.

If x or y coordinate is not provided they will be set to 0 and the click will be triggered on the left-top corner of the element. The coordinates can have a negative value which means the click will be triggered outside of the element.

Mouse events are not propagated immediately, to see consequences of click reflected in page source you must call *splash:wait*

Example 1: get width and height of the element, calculate its center and click on it

```
function main(splash)
    -- ...
    local element = splash:select('.element')
    local bounds = element:bounds()
    assert(element:mouse_click{x=bounds.width/2, y=bounds.height/2})
    -- ...
end
```

Example 2: click on the area above the element by 10 pixels

```
function main(splash)
    -- ...
    local element = splash:select('.element')
    assert(element:mouse_click{y=-10})
    -- ...
end
```

See more about mouse events in *splash:mouse_click*.

element:mouse_hover

Trigger mouse hover (JavaScript mouseover) event on the element.

Signature: `ok, reason = element:mouse_hover{x=0, y=0}`

Parameters:

- x - optional, x coordinate relative to the left corner of the element
- y - optional, y coordinate relative to the top corner of the element

Returns: `ok, reason` pair. If `ok` is nil then error happened during the function call; `reason` provides an information about error type.

Async: no.

If x or y coordinate is not provided they will be set to 0 and the hover will be triggered on the left-top corner of the element. The coordinates can have a negative value which means the hover will be triggered outside of the element.

Mouse events are not propagated immediately, to see consequences of hover reflected in page source you must call *splash:wait*

Example 1: get width and height of the element, calculate its center and hover over it

```
function main(splash)
    -- ...
    local element = splash:select('.element')
    local bounds = element:bounds()
    assert(element:mouse_hover{x=bounds.width/2, y=bounds.height/2})
    -- ...
end
```

Example 2: hover over the area above the element by 10 pixels

```
function main(splash)
    -- ...
    local element = splash:select('.element')
    assert(element:mouse_hover{y=-10})
    -- ...
end
```

See more about mouse events in *splash:mouse_hover*.

element:styles

Return the computed styles of the element.

Signature: `styles = element:styles()`

Returns: `styles` is a table with computed styles of the element.

Async: no.

This method returns the result of JavaScript window.getComputedStyle() applied on the element.

Example: get all computed styles and return the `font-size` property.

```
function main(splash)
    -- ...
    local element = splash:select('.element')
    return element:styles()['font-size']
end
```

element:bounds

Return the bounding client rectangle of the element

Signature: `bounds = element:bounds()`

Returns: `bounds` is a table with the client bounding rectangle with the `top`, `right`, `bottom` and `left` coordinates and also with `width` and `height` values.

Async: no.

Example: get the bounds of the element.

```
function main(splash)
    -- ..
    local element = splash:select('.element')
    return element:bounds()
    -- e.g. bounds is { top = 10, right = 20, bottom = 20, left = 10, height = 10,␣
↪width = 10 }
end
```

element:png

Return a screenshot of the element in PNG format

Signature: `shot = element:png{width=nil, scale_method='raster', pad=0}`

Parameters:

- width - optional, width of a screenshot in pixels;
- scale_method - optional, method to use when resizing the image, `'raster'` or `'vector'`;
- pad - optional, integer or `{left, top, right, bottom}` values of padding

Returns: `shot` is a PNG screenshot data, as a *binary object*. When the result is empty (e.g. if the element doesn't exist in DOM or it isn't visible) `nil` is returned.

Async: no.

pad parameter sets the padding of the resulting image. If it is a single integer then the padding from all sides will be equal. If the value of the padding is positive the resulting screenshot will be expanded by the specified amount of pixes. And if the value of padding is negative the resulting screenshot will be shrunk by the specified amount of pixels.

Example: return a padded screenshot of the element

```lua
function main(splash)
    -- ..
    local element = splash:select('.element')
    return element:png{pad=10}
end
```

See more in *splash:png*.

element:jpeg

Return a screenshot of the element in JPEG format

Signature: `shot = element:jpeg{width=nil, scale_method='raster', quality=75, region=nil, pad=0}`

Parameters:

- width - optional, width of a screenshot in pixels;
- scale_method - optional, method to use when resizing the image, `'raster'` or `'vector'`;
- quality - optional, quality of JPEG image, integer in range from `0` to `100`;
- pad - optional, integer or `{left, top, right, bottom}` values of padding

Returns: `shot` is a JPEG screenshot data, as a *binary object*. When the result is empty (e.g. if the element doesn't exist in DOM or it isn't visible) `nil` is returned.

Async: no.

pad parameter sets the padding of the resulting image. If it is a single integer then the padding from all sides will be equal. If the value of the padding is positive the resulting screenshot will be expanded by the specified amount of pixes. And if the value of padding is negative the resulting screenshot will be shrunk by the specified amount of pixes.

See more in *splash:jpeg*.

element:visible

Check whether the element is visible.

Signature: `visible = element:visible()`

Returns: `visible` indicates whether the element is visible.

Async: no.

element:focused

Check whether the element has focus.

Signature: `focused = element:focused()`

Returns: `focused` indicates whether the element is focused.

Async: no.

element:text

Fetch a text information from the element

Signature: `text = element:text()`

Returns: `text` is a text content of the element.

Async: no.

It tries to return the trimmed value of the following JavaScript `Node` properties:

- textContent
- innerText
- value

If all of them are empty an empty string is returned.

element:info

Get useful information about the element.

Signature: `info = element:info()`

Returns: `info` is a table with element info.

Async: no.

Info is a table with the following fields:

- nodeName - node name in a lower case (e.g. *h1*)
- attributes - table with attributes names and its values
- tag - html string representation of the element
- html - inner html of the element
- text - inner text of the element
- x - x coordinate of the element

- y - y coordinate of the element
- width - width of the element
- height - height of the element
- visible - flag representing if the element is visible

element:field_value

Get value of the field element (input, select, textarea, button).

Signature: `ok, value = element:field_value()`

Returns: `ok, value` pair. If `ok` is nil then error happened during the function call; `value` provides an information about error type. When there is no error `ok` is true and `value` is a value of the element.

Async: no.

This method works in the following way:

- **if the element type is `select`:**
    - if the `multiple` attribute is `true` it returns a *table* with the selected values;
    - otherwise it returns the value of the select;
- **if the element has attribute `type="radio"`:**
    - if it's checked returns its value;
    - other it returns `nil`
- if the element has attribute `type="checkbox"` it returns *bool* value
- otherwise it returns the value of the `value` attribute or *empty string* if it doesn't exist

element:form_values

Return a table with form values if the element type is *form*

Signature: `form_values, reason = element:form_values{values='auto'}`

Parameters:

- values - type of the return value, can be one of `'auto'`, `'list'` or `'first'`

Returns: `form_values, reason` pair. If `form_values` is nil then error happened during the function call or node type is not *form*; `reason` provides an information about error type; otherwise `form_values` is a table with element names as keys and values as values.

Async: no.

The returned values depend on `values` parameter. It can be in 3 states:

`'auto'` Returned values are tables or singular values depending on the form element type:

- if the element is `<select multiple>` the returned value is a table with the selected option values or text contents if the value attribute is missing;
- if the form has several elements with the same `name` attribute the returned value is a table with all values of that elements;
- otherwise it is a string (for text and radio inputs), bool (for checkbox inputs) or `nil` the value of `value` attribute.

This result type is convenient if you're working with the result in a Lua script.

'list' Returned values always are tables (lists), even if the form element can be a singular value, useful for forms with unknown structure. Few notes:

- if the element is a checkbox input and a `value` attribute then the table will contain that value;
- if the element is `<select multiple>` and they are several of them with the same names then their values will be concatenated with the previous ones

This result type is convenient if you're writing generic form-handling code - unlike `auto` there is no need to support multiple data types.

'first' Returned values always are singular values, even if the form element can multiple value. If the element has multiple values only the *first* one will be selected.

Example 1: return the values of the following login form

```
<form id="login">
    <input type="text" name="username" value="admin" />
    <input type="password" name="password" value="pass" />
    <input type="checkbox" name="remember" value="yes" checked />
</form>
```

```
function main(splash)
    -- ...
    local form = splash:select('#login')
    return assert(form:form_values())
end

-- returned values are
{ username = 'admin', password = 'pass', remember = true }
```

Example 2: when `values` is equal to `'list'`

```
function main(splash)
    -- ...
    local form = splash:select('#login')
    return assert(form:form_values{values='list'}))
end

-- returned values are
{ username = ['admin'], password = ['pass'], remember = ['checked'] }
```

Example 3: return the values of the following form when `values` is equal to `'first'`

```
<form>
    <input type="text" name="foo[]" value="coffee"/>
    <input type="text" name="foo[]" value="milk"/>
    <input type="text" name="foo[]" value="eggs"/>
    <input type="text" name="baz" value="foo"/>
    <input type="radio" name="choice" value="yes"/>
    <input type="radio" name="choice" value="no" checked/>
    <input type="checkbox" name="check" checked/>

    <select multiple name="selection">
        <option value="1" selected>1</option>
        <option value="2">2</option>
        <option value="3" selected>2</option>
    </select>
</form>
```

```
function main(splash)
    -- ...
    local form = splash:select('form')
    return assert(form:form_values(false))
end

-- returned values are
{
    ['foo[]'] = 'coffee',
    baz = 'foo',
    choice = 'no',
    check = false,
    selection = '1'
}
```

element:fill

Fill the form with the provided values

Signature: `ok, reason = element:fill(values)`

Parameters:

- values - table with input names as keys and values as input values

Returns: `ok, reason` pair. If `ok` is nil then error happened during the function call; `reason` provides an information about error type.

Async: no.

In order to fill your form your inputs must have `name` property and this method will select those input using that property.

Example 1: get the current values, change password and fill the form

```
<form id="login">
    <input type="text" name="username" value="admin" />
    <input type="password" name="password" value="pass" />
</form>
```

```
function main(splash)
    -- ...
    local form = splash:select('#login')
    local values = assert(form:form_values())
    values.password = "l33t"
    assert(form:fill(values))
end
```

Example 2: fill more complex form

```
<form id="signup" action="/signup">
    <input type="text" name="name"/>
    <input type="radio" name="gender" value="male"/>
    <input type="radio" name="gender" value="female"/>
    <select multiple name="hobbies">
        <option value="sport">Sport</option>
        <option value="cars">Cars</option>
        <option value="games">Video Games</option>
    </select>

    <button type="submit">Sign Up</button>
</form>
```

```
function main(splash)
  assert(splash:go(splash.args.url))
  assert(splash:wait(0.1))

  local form = splash:select('#signup')
  local values = {
    name = 'user',
    gender = 'female',
    hobbies = {'sport', 'games'},
  }

  assert(form:fill(values))
  assert(form:submit())
  -- ...
end
```

element:send_keys

Send keyboard events to the element.

Signature: `ok, reason = element:send_keys(keys)`

Parameters

- keys - string representing the keys to be sent as keyboard events.

Returns: `ok, reason` pair. If `ok` is nil then error happened during the function call; `reason` provides an information about error type.

Async: no.

This method does the following:

- clicks on the element
- send the specified keyboard events

See more about keyboard events in in *splash:send_keys*.

element:send_text

Send keyboard events to the element.

Signature: `ok, reason = element:send_text(text)`

Parameters

- text - string to be sent as input.

Returns: `ok, reason` pair. If `ok` is nil then error happened during the function call; `reason` provides an information about error type.

Async: no.

This method does the following:

- clicks on the element
- send the specified text to the element

See more about it in *splash:send_text*.

element:submit

Submit the form element.

Signature: `ok, reason = element:submit()`

Returns: `ok, reason` pair. If `ok` is nil then error happened during the function call (e.g. you are trying to submit on element which is not a form); `reason` provides an information about error type.

Async: no.

Example: get the form, fill with values and submit it

```
<form id="login" action="/login">
    <input type="text" name="username" />
    <input type="password" name="password" />
    <input type="checkbox" name="remember" />
    <button type="submit">Submit</button>
</form>
```

```
function main(splash)
    -- ...
    local form = splash:select('#login')
    assert(form:fill({ username='admin', password='pass', remember=true }))
    assert(form:submit())
    -- ...
end
```

Working with Binary Data

Motivation

Splash assumes that most strings in a script are encoded to UTF-8. This is true for HTML content - even if the original response was not UTF-8, internally browser works with UTF-8, so *splash:html* result is always UTF-8.

When you return a Lua table from the `main` function Splash encodes it to JSON; JSON is a text protocol which can't handle arbitrary binary data, so Splash assumes all strings are UTF-8 when returning a JSON result.

But sometimes it is necessary to work with binary data: for example, it could be raw image data returned by *splash:png* or a response body of a non-UTF-8 page returned by *splash:http_get*.

Binary Objects

To pass non-UTF8 data to Splash (returning it as a result of `main` or passing as arguments to `splash` methods) a script may mark it as a binary object using *treat.as_binary* function.

Some of the Splash functions already return binary objects: *splash:png*, *splash:jpeg*; *response.body* attribute is also a binary object.

A binary object can be returned as a `main` result directly. It is the reason the following example works (a basic *render.png* implementation in Lua):

```
-- basic render.png emulation
function main(splash)
    assert(splash:go(splash.args.url))
    return splash:png()
end
```

All binary objects have content-type attached. For example, *splash:png* result will have content-type `image/png`.

When returned directly, a binary object data is used as-is for the response body, and Content-Type HTTP header is set to the content-type of a binary object. So in the previous example the result will be a PNG image with a proper Content-Type header.

To construct your own binary objects use *treat.as_binary* function. For example, let's return a 1x1px black GIF image as a response:

```
treat = require("treat")
base64 = require("base64")

function main(splash)
    local gif_b64 = "AQABAIAAAAAAAAAAACH5BAAAAAAALAAAAAABAAEAAAICTAEAOw=="
    local gif_bytes = base64.decode(gif_b64)
    return treat.as_binary(gif_bytes, "image/gif")
end
```

When `main` result is returned, binary object content-type takes a priority over a value set by *splash:set_result_content_type*. To override content-type of a binary object create another binary object with a required content-type:

```
lcoal treat = require("treat")
function main(splash)
    -- ...
    local img = splash:png()
    return treat.as_binary(img, "image/x-png") -- default was "image/png"
end
```

When a binary object is serialized to JSON it is auto-encoded to base64 before serializing. For example, it may happen when a table is returned as a `main` function result:

```
function main(splash)
    assert(splash:go(splash.args.url))

    -- result is a JSON object {"png": "...base64-encoded image data"}
    return {png=splash:png()}
end
```

Available Lua Libraries

When *Sandbox* is disabled all standard Lua modules are available; with a Sandbox ON (default) only some of them can be used. See *Standard Library* for more.

Splash ships several non-standard modules by default:

- *json* - encoded/decode JSON data
- *base64* - encode/decode Base64 data
- *treat* - fine-tune the way Splash works with your Lua varaibles and returns the result.

Unlike standard modules, custom modules should to be imported before use, for example:

```
base64 = require("base64")
function main(splash)
    return base64.encode('hello')
end
```

It is possible to add more Lua libraries to Splash using *Custom Lua Modules* feature.

Standard Library

The following standard Lua 5.2 libraries are available to Splash scripts when Sandbox is enabled (default):

- string
- table
- math
- os

Aforementioned libraries are pre-imported; there is no need to `require` them.

Note: Not all functions from these libraries are currently exposed when *Sandbox* is enabled.

json

A library to encode data to JSON and decode it from JSON to Lua data structure. It provides 2 functions: *json.encode* and *json.decode*.

json.encode

Encode data to JSON.

Signature: `result = json.encode(obj)`

Parameters:

- obj - an object to encode.

Returns: a string with JSON representation of `obj`.

JSON format doesn't support binary data; json.encode handles *Binary Objects* by automatically encoding them to Base64 before putting to JSON.

json.decode

Decode JSON string to a Lua object.

Signature: `decoded = json.decode(s)`

Parameters:

- s - a string with JSON.

Returns: decoded Lua object.

Example:

```
json = require("json")

function main(splash)
    local resp = splash:http_get("http:/myapi.example.com/resource.json")
    local decoded = json.decode(resp.content.text)
    return {myfield=decoded.myfield}
end
```

Note that unlike *json.encode* function, *json.decode* doesn't have any special features to support *binary data*. It means that if you want to get a binary object encoded by *json.encode* back, you need to decode data from base64 yourselves. This can be done in a Lua script using *base64* module.

base64

A library to encode/decode strings to/from Base64. It provides 2 functions: *base64.encode* and *base64.decode*. These functions are handy if you need to pass some binary data in a JSON request or response.

base64.encode

Encode a string or a *binary object* to Base64.

Signature: `encoded = base64.encode(s)`

Parameters:

- s - a string or a *binary object* to encode.

Returns: a string with Base64 representation of `s`.

base64.decode

Decode a string from base64.

Signature: `data = base64.decode(s)`

Parameters:

- s - a string to decode.

Returns: a Lua string with decoded data.

Note that base64.decode may return a non-UTF-8 Lua string, so the result may be unsafe to pass back to Splash (as a part of `main` function result or as an argument to `splash` methods). It is fine if you know the original data was ASCII or UTF8, but if you work with unknown data, "real" binary data or just non-UTF-8 content then call *treat.as_binary* on the result of *base64.decode*.

Example - return 1x1px black gif:

```
treat = require("treat")
base64 = require("base64")

function main(splash)
    local gif_b64 = "AQABAIAAAAAAAAAAACH5BAAAAAAALAAAAAABAAEAAAICTAEAOw=="
    local gif_bytes = base64.decode(gif_b64)
    return treat.as_binary(gif_bytes, "image/gif")
end
```

treat

treat.as_binary

Get a *binary object* for a string.

Signature: `bytes = treat.as_binary(s, content_type="application/octet-stream")`

Parameters:

- s - a string.
- content-type - Content-Type of `s`.

Returns: a *binary object*.

treat.as_binary returns a binary object for a string. This binary object no longer can be processed from Lua, but it can be returned as a main() result as-is.

treat.as_string

Get a Lua string with a raw data from a *binary object*.

Signature: `s, content_type = treat.as_string(bytes)`

Parameters:

- bytes - a *binary object*.

Returns: `(s, content_type)` pair: a Lua string with raw data and its Content-Type.

treat.as_string "unwraps" a *binary object* and returns a plain Lua string which can be processed from Lua. If the resulting string is not encoded to UTF-8 then it is still possible to process it in Lua, but it is not safe to return it as a `main` result or pass to Splash functions. Use *treat.as_binary* to convert processed string to a binary object if you need to pass it back to Splash.

treat.as_array

Mark a Lua table as an array (for JSON encoding and Lua -> JS conversions).

Signature: `tbl = treat.as_array(tbl)`

Parameters:

- tbl - a Lua table.

Returns: the same table.

JSON can represent arrays and objects, but in Lua there is no distinction between them; both key-value mappings and arrays are stored in Lua tables.

By default, Lua tables are converted to JSON objects when returning a result from Splash `main` function and when using *json.encode* or ref:*splash-jsfunc*:

```
function main(splash)
    -- client gets {"foo": "bar"} JSON object
    return {foo="bar"}
end
```

It can lead to unexpected results with array-like Lua tables:

```
function main(splash)
    -- client gets {"1": "foo", "2": "bar"} JSON object
    return {"foo", "bar"}
end
```

treat.as_array allows to mark tables as JSON arrays:

```
treat = require("treat")

function main(splash)
    local tbl = {"foo", "bar"}
    treat.as_array(tbl)

    -- client gets ["foo", "bar"] JSON object
    return tbl
end
```

This function modifies its argument inplace, but as a shortcut it returns the same table; it allows to simplify the code:

```
treat = require("treat")
function main(splash)
    -- client gets ["foo", "bar"] JSON object
    return treat.as_array({"foo", "bar"})
end
```

Note: There is no autodetection of table type because `{}` Lua table is ambiguous: it can be either a JSON array or as a JSON object. With table type autodetection it is easy to get a wrong output: even if some data is always an array, it can be suddenly exported as an object when an array is empty. To avoid surprises Splash requires an explicit *treat.as_array* call.

Adding Your Own Modules

Splash provides a way to use custom Lua modules (stored on server) from scripts passed via HTTP API. This allows to

1. reuse code without sending it over network again and again;
2. use third-party Lua modules;
3. implement features which need unsafe code and expose them safely in a sandbox.

Note: To learn about Lua modules check e.g. http://lua-users.org/wiki/ModulesTutorial. Please prefer "the new way" of writing modules because it plays better with a sandbox. A good Lua modules style guide can be found here: http://hisham.hm/2014/01/02/how-to-write-lua-modules-in-a-post-module-world/

Setting Up

To use custom Lua modules, do the following steps:

1. setup the path for Lua modules and add your modules there;
2. tell Splash which modules are enabled in a sandbox;
3. use Lua `require` function from a script to load a module.

To setup the path for Lua modules start Splash with `--lua-package-path` option. `--lua-package-path` value should be a semicolon-separated list of places where Lua looks for modules. Each entry should have a ? in it that's replaced with the module name.

Example:

```
$ python -m splash.server --lua-package-path "/etc/splash/lua_modules/?.lua;/home/
↪myuser/splash-modules/?.lua"
```

Note: If you use Splash installed using Docker see *Folders Sharing* for more info on how to setup paths.

Note: For the curious: `--lua-package-path` value is added to Lua `package.path`.

When you use a *Lua sandbox* (default) Lua `require` function is restricted when used in scripts: it only allows to load modules from a whitelist. This whitelist is empty by default, i.e. by default you can require nothing. To make your modules available for scripts start Splash with `--lua-sandbox-allowed-modules` option. It should contain a semicolon-separated list of Lua module names allowed in a sandbox:

```
$ python -m splash.server --lua-sandbox-allowed-modules "foo;bar" --lua-package-path
↪"/etc/splash/lua_modules/?.lua"
```

After that it becomes possible to load these modules from Lua scripts using `require`:

```
local foo = require("foo")
function main(splash)
    return {result=foo.myfunc()}
end
```

Writing Modules

A basic module could look like the following:

```
-- mymodule.lua
local mymodule = {}

function mymodule.hello(name)
    return "Hello, " .. name
```

```
end

return mymodule
```

Usage in a script:

```
local mymodule = require("mymodule")

function main(splash)
    return mymodule.hello("world!")
end
```

Many real-world modules will likely want to use `splash` object. There are several ways to write such modules. The simplest way is to use functions that accept `splash` as an argument:

```
-- utils.lua
local utils = {}

-- wait until `condition` function returns true
function utils.wait_for(splash, condition)
    while not condition() do
        splash:wait(0.05)
    end
end

return utils
```

Usage:

```
local utils = require("utils")

function main(splash)
    splash:go(splash.args.url)

    -- wait until <h1> element is loaded
    utils.wait_for(splash, function()
       return splash:evaljs("document.querySelector('h1') != null")
    end)

    return splash:html()
end
```

Another way to write such module is to add a method to `splash` object. This can be done by adding a method to its `Splash` class - the approach is called "open classes" in Ruby or "monkey-patching" in Python.

```
-- wait_for.lua

-- Sandbox is not enforced in custom modules, so we can import
-- internal Splash class and change it - add a method.
local Splash = require("splash")

function Splash:wait_for(condition)
    while not condition() do
        self:wait(0.05)
    end
end

-- no need to return anything
```

Usage:

```
require("wait_for")

function main(splash)
    splash:go(splash.args.url)

    -- wait until <h1> element is loaded
    splash:wait_for(function()
       return splash:evaljs("document.querySelector('h1') != null")
    end)

    return splash:html()
end
```

Which style to prefer is up to the developer. Functions are more explicit and composable, monkey patching enables a more compact code. Either way, `require` is explicit.

As seen in a previous example, sandbox restrictions for standard Lua modules and functions **are not applied** in custom Lua modules, i.e. you can use all the Lua powers. This makes it possible to import third-party Lua modules and implement advanced features, but requires developer to be careful. For example, let's use os module:

```
-- evil.lua
local os = require("os")
local evil = {}

function evil.sleep()
    -- Don't do this! It blocks the event loop and has a startup cost.
    -- splash:wait is there for a reason.
    os.execute("sleep 2")
end

function evil.touch(filename)
    -- another bad idea
    os.execute("touch " .. filename)
end

-- todo: rm -rf /

return evil
```

Splash and Jupyter

Splash provides a custom Jupyter (previously known as IPython) kernel for Lua. Together with Jupyter notebook frontend it forms an interactive web-based development environment for Splash Scripts with syntax highlighting, smart code completion, context-aware help, inline images support and a real live WebKit browser window with Web Inspector enabled, controllable from a notebook.

Installation

To install Splash-Jupyter using Docker, run:

```
$ docker pull scrapinghub/splash-jupyter
```

Then start the container:

```
$ docker run -p 8888:8888 -it scrapinghub/splash-jupyter
```

Note: Without `-it` flags you won't be able to stop the container using Ctrl-C.

If you're on Linux, Jupyter server with Splash kernel enabled will be available at http://0.0.0.0:8888.

If you use boot2docker, run `$ boot2docker ip` to get the ip address, the visit http://<ip-returned-by-boot2docker>:8888. If you use docker-machine, run `$ docker-machine ip <your machine>` to get the ip.

By default, notebooks are stored in a Docker container; they are destroyed when you restart an image. To persist notebooks you can mount a local folder to `/notebooks`. For example, let's use current folder to store the notebooks:

```
$ docker run -v `/bin/pwd`/notebooks:/notebooks -p 8888:8888 -it splash-jupyter
```

To view Live Webkit window with web inspector when Splash-Jupyter is executed from Docker, you will need to pass additional docker parameters to share the host system's X server with the docker container, and use the `--disable-xvfb` command line flag:

```
$ docker run -e DISPLAY=unix$DISPLAY \
             -v /tmp/.X11-unix:/tmp/.X11-unix \
             -v $XAUTHORITY:$XAUTHORITY \
             -e XAUTHORITY=$XAUTHORITY \
             -p 8888:8888 \
             -it scrapinghub/splash-jupyter --disable-xvfb
```

Alternatively, to enable live Webkit window you can install Splash in a "manual way" - see *Ubuntu 14.04 (manual way)*.

1. Install IPython/Jupyter with notebook feature. Splash kernel requires IPython 4.x:

   ```
   $ pip3 install 'ipython[notebook] >= 4.1.2, < 5.0'
   ```

2. Make sure Splash is installed: run `pip3 install -U splash`. If you use Splash master branch run `pip3 install -U .` from source checkout instead.

3. Let IPython know about Splash kernel by running the following command:

   ```
   $ python3 -m splash.kernel install
   ```

To run IPython with Splash notebook, first start IPython notebook and then create a new Splash notebook using "New" button.

From Notebook to HTTP API

After you finished developing the script using a Jupyter Notebook, you may want to convert it to a form suitable for submitting to Splash HTTP API (see *execute*).

To do that, copy-paste (or download using "File -> Download as -> .lua") all relevant code, then put it inside `function main(splash)`:

```
function main(splash)
    -- Script code goes here,
    -- including all helper functions.
    return {...}  -- return the result
end
```

To make the script more generic you can use *splash.args* instead of hardcoded constants (e.g. for page urls). Also, consider submitting several requests with different arguments instead of running a loop in a script if you need to visit and process several pages - it is an easy way to parallelize the work.

There are some gotchas:

1. When you run a notebook cell and then run another notebook cell there is a delay between runs; the effect is similar to inserting *splash:wait* calls at the beginning of each cell.
2. Regardless of *sandbox* settings, scripts in Jupyter notebook are **not** sandboxed. Usually it is not a problem, but some functions may be unavailable in HTTP API if sandbox is enabled.

FAQ

I'm getting lots of 504 Timeout errors, please help!

HTTP 504 error means a request to Splash took more than *timeout* seconds to complete (30s by default) - Splash aborts script execution after the timeout. To override the timeout value pass *'timeout'* argument to the Splash endpoint you're using.

Note that the maximum allowed `timeout` value is limited by the maximum timeout setting, which is by default 60 seconds. In other words, by default you can't pass `?timeout=300` to run a long script - an error will be returned.

Maximum allowed timeout can be increased by passing `--max-timeout` option to Splash server on startup:

```
$ python -m splash.server --max-timeout 3600
```

For Docker the command would be something like this (see *Passing Custom Options*):

```
$ docker run -it -p 8050:8050 scrapinghub/splash --max-timeout 3600
```

The next question is why a request can need 10 minutes to render. There are 3 common reasons:

1. Slow website

A website can be really slow, or it can try to get some remote resources which are really slow.

There is no way around increasing timeouts and reducing request rate if the website itself is slow. However, often the problem lays in unreliable remote resources like third-party trackers or advertisments. By default Splash waits for all remote resources to load, but in most cases it is better not to wait for them forever.

To abort resource loading after a timeout and give the whole page a chance to render use resource timeouts. For render.*** endpoints use *'resource_timeout'* argument; for *execute* use either *splash.resource_timeout* or `request:set_timeout` (see *splash:on_request*).

It is a good practive to always set resource_timeout; something similar to `resource_timeout=20` often works well.

2. Splash Lua script does too many things

When a script fetches many pages or uses large delays then timeouts are inevitable. Sometimes you have to run such scripts; in this case increase `--max-timeout` Splash option and use larger *timeout* values.

But before increasing the timeouts consider splitting your script into smaller steps and sending them to Splash individually. For example, if you need to fetch 100 websites, don't write a Splash Lua script which takes a list of 100 URLs and fetches them - write a Splash Lua script that takes 1 URL and fetches it, and send 100 requests to Splash. This approach has a number of benefits: it makes scripts more simple and robust and enables parallel processing.

3. Splash instance is overloaded

When Splash is overloaded it may start producing 504 errors.

Splash renders requests in parallel, but it doesn't render them *all* at the same time - concurrency is limited to a value set at startup using `--slots` option. When all slots are used a request is put into a queue. The thing is that a timeout starts to tick once Splash receives a request, not when Splash starts to render it. If a request stays in an internal queue for a long time it can timeout even if a website is fast and splash is capable of rendering the website.

To increase rendering speed and fix an issue with a queue it is recommended to start several Splash instances and use a load balancer capable of maintaining its own request queue. HAProxy has all necessary features; check an example config here. A shared request queue in a load balancer also helps with reliability: you won't be loosing requests if a Splash instance needs to be restarted.

Note: Nginx (which is another popular load balancer) provides an internal queue only in its commercial version, Nginx Plus.

How to run Splash in production?

Easy Way

If you want to get started quickly take a look at Aquarium (which is a Splash setup without many of the pitfalls) or use a hosted solution like ScrapingHub's.

Don't forget to use resource timeous in your client code (see *1. Slow website*). It also makes sense to retry a couple of times if Splash returns 5xx error response.

Hard Way

If you want to create your own production setup, here is a small non-exhaustive checklist:

- Splash should be daemonized and started on boot;
- in case of failures or segfaults Splash must be restarted;
- memory usage should be limited;
- several Splash instances should be started to use all CPU cores and/or multiple servers;
- requests queue should be moved to the load balancer to make rendering more robust (see *3. Splash instance is overloaded*).

Of course, it is also good to setup monitoring, configuration management, etc. - all the usual stuff.

To daemonize Splash, start it on boot and restart on failures one can use Docker: since Docker 1.2 there are `--restart` and `-d` options which can be used together. Another way to do that is to use standard tools like upstart, systemd or supervisor.

Note: Docker `--restart` option won't work without `-d`.

Splash uses an unbound in-memory cache and so it will eventually consume all RAM. A workaround is to restart the process when it uses too much memory; there is Splash `--maxrss` option for that. You can also add Docker `--memory` option to the mix.

In production it is a good idea to pin Splash version - instead of `scrapinghub/splash` it is usually better to use something like `scrapinghub/splash:2.0`.

A command for starting a long-running Splash server which uses up to 4GB RAM and daemonizes & restarts itself could look like this:

```
$ docker run -d -p 8050:8050 --memory=4.5G --restart=always scrapinghub/splash:2.0 --
↪maxrss 4000
```

You also need a load balancer; for example configs check Aquarium or an HAProxy config in Splash repository.

Ansible Way

Ansible role for Splash is available via third-party project: https://github.com/nabilm/ansible-splash.

How do I disable Private mode?

With Splash>=2.0, you can disable Private mode (which is "on" by default). There are two ways to go about it:

- at startup, with the `--disable-private-mode` argument, e.g., if you're using Docker:

  ```
  $ sudo docker run -it -p 5023:5023 -p 8050:8050 -p 8051:8051 scrapinghub/splash --
  ↪disable-private-mode
  ```

- at runtime when using the `/execute` endpoint and setting *splash.private_mode_enabled* attribute to `false`

Note that if you disable private mode then browsing data such as cookies or items kept in localStorage may persist between requests. If you're using Splash in a shared environment it could mean your cookies or local storage items can be accessed by other clients, or that you can occasionally access other client's cookies.

You may still want to turn Private mode off because in WebKit localStorage doesn't work when Private mode is enabled, and it is not possible to provide a JavaScript shim for localStorage. So for some websites you may have to turn Private model off.

Why was Splash created in the first place?

Please refer to this great answer from kmike on reddit.

Why does Splash use Lua for scripting, not Python or JavaScript?

Check this GitHub Issue for the motivation.

render.html result looks broken in a browser

When you check `http://<splash-server>:8050/render.html?url=<url>` in a browser it is likely stylesheets & other resources won't load properly. It happens when resource URLs are relative - the browser will resolve them as relative to `http://<splash-server>:8050/render.html?url=<url>`, not to `url`. This is not a Splash bug, it is a standard browser behaviour.

If you just want to check how the page looks like after rendering use *render.png* or *render.jpeg* endpoints. If screenshot is not an option and you want to display html with images, etc. using a browser then you may post-process the HTML and add an appropriate <base> HTML tag to the page.

baseurl Splash argument can't help here. It allows to render a page located at one URL as if it is located at another URL. For example, you can host a copy of page HTML on your server, but use baseurl of the original page. This way Splash will resolve relative URLs as relative to original page URL, so that you can get e.g. a proper screenshot or execute proper JavaScript code.

But by passing baseurl you're instructing **Splash** to use it, not **your browser**. It doesn't change relative links to absolute in DOM, it makes Splash to treat them as relative to baseurl when rendering.

Changing links to absolute in DOM tree is not what browsers do when base url is applied - e.g. if you check href attribute using JS code it will still contain relative value even if `<base>` tag is used. *render.html* returns DOM snapshot, so the links are not changed.

When you load *render.html* result in a browser it is **your browser** who resolves relative links, not Splash, so they are resolved incorrectly.

Contributing to Splash

Splash is free & open source. Development happens at GitHub: https://github.com/scrapinghub/splash

Development Setup

Consult with *Installation* to get Splash up and running.

Install development specific dependencies with:

```
$ sudo apt-get install libffi-dev libssl-dev

pip install -r requirements-dev.txt
```

Functional Tests

Run with:

```
py.test --doctest-modules splash
```

To speedup test running install `pytest-xdist` Python package and run Splash tests in parallel:

```
py.test --doctest-modules -n4 splash
```

Stress tests

There are some stress tests that spawn its own splash server and a mock server to run tests against.

To run the stress tests:

```
python -m splash.tests.stress
```

Typical output:

```
$ python -m splash.tests.stress
Total requests: 1000
Concurrency   : 50
Log file      : /tmp/splash-stress-48H91h.log
......................................................................................
↪....................................................................................
↪....................................................................................
↪....................................................................................
↪....................................................................................
↪....................................................................................
↪....................................................................................
↪....................................................................................
↪....................................................................................
↪....................................................................................
↪....................................................................................
↪..........................................................................
Received/Expected (per status code or error):
  200: 500/500
  504: 200/200
  502: 300/300
```

Implementation Details

This section contains information useful if you want to understand Splash codebase.

JavaScript <-> Python <-> Lua intergation

Lua and JavaScript are not connected directly; they communicate through Python.

Python <-> Lua is handled using lupa library. `splash.qtrender_lua.command()` decorator handles most of Python <-> Lua integration.

Python <-> JavaScript is handled using custom serialization code. QT host objects are not used (with a few exceptions). Instead of this JavaScript results are sanitized and processed in Python; Python results are encoded to JSON and decoded/processed in JavaScript.

Python -> Lua

Data is converted from Python to Lua in two cases:

1. method of an exposed Python object returns a result (most common example is a method of `splash` Lua object);
2. Python code calls Lua function with arguments - it could be e.g. an on_request callback.

Conversion rules:

- Basic Python types are converted to Lua: strings -> Lua strings, lists and dicts -> Lua tables, numbers -> Lua numbers, None -> nil(?).

  This is handled using `splash.lua_runtime.SplashLuaRuntime.python2lua()` method. For attributes exposed to Lua this method is called manually; for return results of Python functions / methods it is handled by `splash.qtrender_lua.emits_lua_objects()` decorator. Methods decorated with `@command` use `splash.qtrender_lua.emits_lua_objects` internally, so a Python method decorated with `@command` decorator may return Python result in its body, and the final result would be a Lua object.

- If there is a need to expose a custom Python object to Lua then a subclass of `splash.qtrender_lua.BaseExposedObject` is used; it is wrapped to a Lua table using utilities from wraputils.lua. Lua table exposes whitelisted attributes and methods of the object using metatable, and disallows access to all other attributes.

- Other than that, there is no automatic conversion. If something is not converted then it is available for Lua as an opaque userdata object; access to methods and attributes is disabled by a sandbox.

- To prevent wrapping method may return `splash.lua.PyResult` instance.

Lua -> Python

Lua -> Python conversion is needed in two cases:

1. Lua code calls Python code, passing some arguments;
2. Python code calls Lua code and wants a result back.

- Basic Lua types are converted to Python using `splash.lua_runtime.SplashLuaRuntime.lua2python()`. For method arguments lua2python is called by `splash.qtrender_lua.decodes_lua_arguments()` decorator; `@command` decorator uses `decodes_lua_arguments` internally.

- Python objects which were exposed to Lua (BaseExposedObject subclasses) are **not** converted back. By default they raise an error; with decode_arguments=False they are available as opaque Lua (lupa) table objects.

  `splash.qtrender_lua.is_wrapped_exposed_object()` can be used to check if a lupa object is a wrapped BaseExposedObject instance; obj.unwrapped() method can be used to access the underlying Python object.

JavaScript -> Python

To get results from JavaScript to Python they are converted to primitive JSON-serializable types first. QtWebKit host objects are not used. Objects of unknown JavaScript types are discared, max depth of result is limited.

JavaScript -> Python conversion utilities reside in

- `splash.jsutils` module - JavaScript side, i.e. sanitizing and encoding; two main functions are `SANITIZE_FUNC_JS` and `STORE_DOM_ELEMENTS_JS`;
- `splash.browser_tab.BrowserTab.evaljs()` method - Python side, i.e. decoding of the result.

For most types (objects, arrays, numbers, strings) conversion method is straightforward; the most tricky case is a reference to DOM nodes.

For top-level DOM nodes (i.e. a result is a DOM node or a NodeList) a node is stored in a special window attribute, and generated id is returned to Python instead. All other DOM nodes are discarded - returning a Node or a NodeList as a part of data structure is not supported at the moment. `STORE_DOM_ELEMENTS_JS` processes Node and NodeList objects; `SANITIZE_FUNC_JS` sanitizes the result (handles all other data types, drops unsupported data).

In Python HTMLElement objects are created for DOM nodes; they contain node_id attribute with id returned by JavaScript; it allows to fetch the real Node object in JavaScript. This is handled by `splash.browser_tab.BrowserTab.evaljs()`.

Python -> JavaScript

There are two cases Python objects are converted to JavaScript objects:

1. functions created with splash:jsfunc() are called with arguments;
2. methods of HtmlElement which wrap JS functions are called with arguments.

The conversion is handled either by `splash.html_element.escape_js_args()` or by `splash.jsutils.escape_js()`.

- `escape_js` just encodes Python data to JSON and removes quotes; the result can be used as literal representation of argument values, i.e. added to a JS function call using string formatting.
- `escape_js_args` is similar to `escape_js`, but it handles `splash.html_element.HTMLElement` instances by replacing them with JS code to access stored nodes.

Changes

2.3.3 (2017-06-07)

- WebGL support in default Docker image;
- Maximum value for `wait` argument in render.??? endpoints is increased from 10 seconds to 30 seconds;
- Lua sandbox limits (RAM and CPU) are raised;
- documentation and testing improvements.

2.3.2 (2017-03-03)

- security fix: Xvfb shouldn't listen to tcp.

2.3.1 (2017-01-24)

- Fixed proxy authentication for proxies set using *'proxy'* HTTP argument;
- minor documentation fixes.

2.3 (2016-12-01)

This release adds lots of scraping helpers to Splash: CSS selectors, form filling, easy access to HTML node attributes. Scraping helpers were implemented by Michael Manukyan as a Google Summer of Code 2016 project.

New features:

- *splash:select* and *splash:select_all* methods which allow to execute CSS selectors;
- new *Element* object which wraps JavaScript DOM node and allows to interact with it.

2.2.2 (2016-11-10)

This is a bug fix release:

- Splash-Jupyter is fixed;
- fix an issue with non-ascii HTTP status messages;
- upgrade Pillow to 3.4.2.

2.2.1 (2016-10-17)

This is a bug fix release:

- fix Splash UI in Chrome when serving from localhost;
- upgrade adblockparser to 0.7 to support recent easylist filters;
- upgrade Pillow to 3.3.3.

2.2 (2016-09-10)

New features:

- new *splash:send_keys* and *splash:send_text* methods allow to send native keyboard events to browser;
- new *splash:with_timeout* method allows to limit execution time of blocks of code;
- new *splash.plugins_enabled* attribute which allows to enable Flash; Flash is now available in Docker image, but it is still disabled by default.
- new *splash.response_body_enabled* attribute, *request:enable_response_body* method and *response_body* argument allows to access and export response bodies.

Bug fixes:

- fixed handling of *splash:call_later*, *splash:on_request*, *splash:on_response* and *splash:on_response_headers* callback arguments;
- fixed cleanup of various callbacks;
- fixed *save_args* in Python 2.x;

Other changes:

- internal cleanup of Lua <-> Python interaction;
- Pillow library is updated in Docker image;
- HarViewer is upgraded to a recent version.

2.1 (2016-04-20)

New features:

- 'region' argument for *splash:png* and *splash:jpeg* methods allow to take screenshots of parts of pages;
- *save_args* and *load_args* parameters allow to save network traffic by caching large request arguments inside Splash server;
- new *splash:mouse_click*, *splash:mouse_press*, *splash:mouse_release* and *splash:mouse_hover* methods for sending mouse events to web pages.

Bug fixes:

- User-Agent is set correctly for requests with baseurl;
- "download" links in Splash UI are fixed;
- an issue with ad blockers preventing Splash UI to work is fixed.

2.0.3 (2016-03-04)

This is a bugfix release:

- Splash Notebook is fixed to work with recent ipykernel versions;
- segfaults in adblock middleware are fixed;
- adblock parsing issues are fixed by upgrading adblockparser to v0.5;
- fixed handling of adblock rules with 'domain' option: domain is now extracted from the page URL, not necessarily from 'url' Splash argument.

2.0.2 (2016-02-26)

This is a bugfix release:

- an issue which may cause segfaults is fixed.

2.0.1 (2016-02-25)

This is a bugfix release:

- XSS in HTTP UI is fixed;
- Splash-Jupyter docker image is fixed.

2.0 (2016-02-21)

Splash 2.0 uses Qt 5.5.1 instead of Qt 4; it means the rendering engine now supports more HTML5 features and is more modern overall. Also, the official Docker image now uses Python 3 instead of Python 2. This work is largely done by Tarashish Mishra as a Google Summer of Code 2015 project.

Splash 2.0 release introduces other cool new features:

- many Splash HTTP UI improvements;
- better support for *binary data*;
- built-in *json* and *base64* libraries;
- more *control* for result serialization (support for JSON arrays and raw bytes);
- it is now possible to turn Private mode OFF at startup using command-line option or at runtime using *splash.private_mode_enabled* attribute;
- *_ping* endpoint is added;
- cookie handling is fixed;
- downloader efficiency is improved;

- request processing is stopped when client disconnects;
- logging inside callbacks now uses proper verbosity;
- sandbox memory limit for user objects is increased to 50MB;
- some sandboxing issues are fixed;
- *splash:evaljs* and *splash:jsfunc* results are sanitized better;
- it is possible to pass arguments when starting Splash-Jupyter - it means now you can get a browser window for splash-jupyter when it is executed from docker;
- proxy authentication is fixed;
- logging improvements: logs now contain request arguments in JSON format; errors are logged;

There are **backwards-incompatible** changes to *Splash Scripting*: previously, different Splash methods were returning/receiving inconsistent response and request objects. For example, *splash:http_get* response was not in the same format as `response` received by *splash:on_response* callbacks. Splash 2.0 uses *Request* and *Response* objects consistently. Unfortunately this requires changes to existing user scripts:

- replace `resp = splash:http_get(...)` and `resp = splash:http_post(...)` with `resp = splash:http_get(...).info` and `resp = splash:http_post(...).info`. Client code also may need to be changed: the default encoding of `info['content']['text']` is now base64. If you used `resp.content.text` consider switching to *response.body*.
- `response` object received by *splash:on_response_headers* and *splash:on_response* callbacks is changed: instead of `response.request` write `response.request.info`.

Serialization of JS objects in *splash:jsfunc*, *splash:evaljs* and *splash:wait_for_resume* **is changed**: circular objects are no longer returned, Splash doesn't try to serialize DOM elements, and error messages are changed.

Splash **no longer supports** QT-based disk cache; it was disable by default and it usage was discouraged since Splash 1.0, in Splash 2.0 `--cache` command-line option is removed. For HTTP cache there are better options like Squid.

Another **backwards-incompatible** change is that Splash-as-a-proxy feature is removed. Please use regular HTTP API instead of this proxy interface. Of course, Splash will still support using proxies to make requests, these are two different features.

1.8 (2015-09-29)

New features:

- POST requests support: *http_method* and *body* arguments for render endpoints; new *splash:go* arguments: `body`, `http_method` and `formdata`; new *splash:http_post* method.
- Errors are now returned in JSON format; error mesages became more detailed; Splash UI now displays detailed error information.
- new *splash:call_later* method which allows to schedule tasks in future;
- new *splash:on_response* method allows to register a callback to be executed after each response;
- proxy can now be set directly, without using proxy profiles - there is a new *proxy* argument for render endpoints;
- more detailed *splash:go* errors: a new "render_error" error type can be reported;
- new *splash:set_result_status_code* method;
- new *splash.resource_timeout* attribute as a shortcut for `request:set_timeout` in *splash:on_request*;
- new *splash:get_version* method;

- new *splash:autoload_reset*, *splash:on_response_reset*, *splash:on_request_reset*, *splash:on_response_headers_reset*, *splash:har_reset* methods and a new `reset=true` argument for *splash:har*. They are most useful with Splash-Jupyter.

Bug fixes and improvements:

- fixed an issue: proxies were not applied for POST requests;
- improved argument validation for various methods;
- more detailed logs;
- it is now possible to load a combatibility shim for window.localStorage;
- code coverage integration;
- improved Splash-Jupyter tests;
- Splash-Jupyter is upgraded to Jupyter 4.0.

1.7 (2015-08-06)

New features:

- *render.jpeg* endpoint and *splash:jpeg* function allow to take screenshots in JPEG format;
- *splash:on_response_headers* Lua function and *allowed_content_types* / *forbidden_content_types* HTTP arguments allow to discard responses earlier based on their headers;
- *splash.images_enabled* attribute to enable/disable images from Lua scripts;
- *splash.js_enabled* attribute to enable/disable JS processing from Lua scripts;
- *splash:set_result_header* method for setting custom HTTP headers returned to Splash clients;
- *resource_timeout* argument for setting network request timeouts in render endpoints;
- `request:set_timeout(timeout)` method (ses *splash:on_request*) for setting request timeouts from Lua scripts;
- SOCKS5 proxy support: new 'type' argument in *proxy profile* config files and `request:set_proxy` method (ses *splash:on_request*)
- enabled HTTPS proxying;

Other changes:

- HTTP error detection is improved;
- MS fonts are added to the Docker image for better rendering quality;
- Chinese fonts are added to the Docker image to enable rendering of Chinese websites;
- validation of `timeout` and `wait` arguments is improved;
- documentation: grammar is fixed in the tutorial;
- assorted documentation improvements and code cleanups;
- `splash:set_images_enabled` method is deprecated.

1.6 (2015-05-15)

The main new feature in Splash 1.6 is *splash:on_request* function which allows to process individual outgoing requests: log, abort, change them.

Other improvements:

- a new *_gc* endpoint which allows to clear QWebKit caches;
- Docker images are updated with more recent package versions;
- HTTP arguments validation is improved;
- serving Splash UI under HTTPS is fixed.
- documentation improvements and typo fixes.

1.5 (2015-03-03)

In this release we introduce *Splash-Jupyter* - a web-based IDE for Splash Lua scripts with syntax highlighting, auto-completion and a connected live browser window. It is implemented as a kernel for Jupyter (IPython).

Docker images for Splash 1.5 are optimized - download size is much smaller than in previous releases.

Other changes:

- *splash:go()* returned incorrect result after an unsuccessful splash:go() call - this is fixed;
- Lua `main` function can now return multiple results;
- there are testing improvements and internal cleanups.

1.4 (2015-02-10)

This release provides faster and more robust screenshot rendering, many improvements in Splash scripting engine and other improvements like better cookie handling.

From version 1.4 Splash requires Pillow (built with PNG support) to work.

There are backwards-incompatible changes in Splash scripts:

- splash:set_viewport() is split into *splash:set_viewport_size()* and *splash:set_viewport_full()*;
- old splash:runjs() method is renamed to *splash:evaljs()*;
- new *splash:runjs* method just runs JavaScript code without returning the result of the last JS statement.

To upgrade check all splash:runjs() usages: if the returned result is used then replace splash:runjs() with splash:evaljs().

`viewport=full` argument is deprecated; use `render_all=1`.

New scripting features:

- it is now possible to write custom Lua plugins stored server-side;
- a restricted version of Lua `require` is enabled in sandbox;
- *splash:autoload()* method for setting JS to load on each request;
- *splash:wait_for_resume()* method for interacting with async JS code;
- *splash:lock_navigation()* and *splash:unlock_navigation()* methods;
- splash:set_viewport() is split into *splash:set_viewport_size()* and *splash:set_viewport_full()*;

- *splash:get_viewport_size()* method;
- *splash:http_get()* method for sending HTTP GET requests without loading result to the browser;
- *splash:set_content()* method for setting page content from a string;
- *splash:get_cookies()*, *splash:add_cookie()*, *splash:clear_cookies()*, *splash:delete_cookies()* and *splash:init_cookies()* methods for working with cookies;
- *splash:set_user_agent()* method for setting User-Agent header;
- *splash:set_custom_headers()* method for setting other HTTP headers;
- *splash:url()* method for getting current URL;
- *splash:go()* now accepts `headers` argument;
- *splash:evaljs()* method, which is a splash:runjs() from Splash v1.3.1 with improved error handling (it raises an error in case of JavaScript exceptions);
- *splash:runjs()* method no longer returns the result of last computation;
- *splash:runjs()* method handles JavaScript errors by returning `ok, error` pair;
- *splash:get_perf_stats()* method for getting Splash resource usage.

Other improvements:

- –max-timeout option can be passed to Splash at startup to increase or decrease maximum allowed timeout value;
- cookies are no longer shared between requests;
- PNG rendering becomes more efficient: less CPU is spent on compression. The downside is that the returned PNG images become 10-15% larger;
- there is an option (`scale_method=vector`) to resize images while painting to avoid pixel-based resize step - it can make taking a screenshot much faster on image-light webpages (up to several times faster);
- when 'height' is set and image is downscaled the rendering is more efficient because Splash now avoids rendering unnecessary parts;
- /debug endpoint tracks more objects;
- testing setup improvements;
- application/json POST requests handle invalid JSON better;
- undocumented splash:go_and_wait() and splash:_wait_restart_on_redirects() methods are removed (they are moved to tests);
- Lua sandbox is cleaned up;
- long log messages from Lua are truncated in logs;
- more detailed error info is logged;
- example script in Splash UI is simplified;
- stress tests now include PNG rendering benchmark.

Bug fixes:

- default viewport size and window geometry are now set to 1024x768; this fixes PNG screenshots with viewport=full;
- PNG rendering is fixed for huge viewports;
- splash:go() argument validation is improved;

- timer is properly deleted when an exception is raised in an errback;
- redirects handling for baseurl requests is fixed;
- reply is deleted only once when baseurl is used.

1.3.1 (2014-12-13)

This release fixes packaging issues with Splash 1.3.

1.3 (2014-12-04)

This release introduces an experimental *scripting support*.

Other changes:

- manhole is disabled by default in Debian package;
- more objects are tracked in /debug endpoint;
- "history" in render.json now includes "queryString" keys; it makes the output compatible with HAR entry format;
- logging improvements;
- improved timer cancellation.

1.2.1 (2014-10-16)

- Dockerfile base image is downgraded to Ubuntu 12.04 to fix random crashes;
- Debian/buildbot config is fixed to make Splash UI available when deployed from deb;
- Qt / PyQt / sip / WebKit / Twisted version numbers are logged at startup.

1.2 (2014-10-14)

- All Splash rendering endpoints now accept `Content-Type: application/json` POST requests with JSON-encoded rendering options as an alternative to using GET parameters;
- `headers` parameter allows to set HTTP headers (including user-agent) for all endpoints - previously it was possible only in proxy mode;
- `js_source` parameter allows to execute JS in page context without `application/javascript` POST requests;
- testing suite is switched to pytest, test running can now be parallelized;
- viewport size changes are logged;
- `/debug` endpoint provides leak info for more classes;
- Content-Type header parsing is less strict;
- documentation improvements;
- various internal code cleanups.

1.1 (2014-10-10)

- An UI is added - it allows to quickly check Splash features.
- Splash can now return requests/responses information in HAR format. See *render.har* endpoint and *har* argument of render.json endpoint. A simpler *history* argument is also available. With HAR support it is possible to get timings for various events, HTTP status code of the responses, HTTP headers, redirect chains, etc.
- Processing of related resources is stopped earlier and more robustly in case of timeouts.
- *wait* parameter changed its meaning: waiting now restarts after each redirect.
- Dockerfile is improved: image is updated to Ubuntu 14.04; logs are shown immediately; it becomes possible to pass additional options to Splash and customize proxy/js/filter profiles; adblock filters are supported in Docker; versions of Python dependencies are pinned; Splash is started directly (without supervisord).
- Splash now tries to start Xvfb automatically - no need for xvfb-run. This feature requires `xvfbwrapper` Python package to be installed.
- Debian package improvements: Xvfb viewport matches default Splash viewport, it is possible to change Splash option using SPLASH_OPTS environment variable.
- Documentation is improved: finally, there are some install instructions.
- Logging: verbosity level of several logging events are changed; data-uris are truncated in logs.
- Various cleanups and testing improvements.

1.0 (2014-07-28)

Initial release.