Splash Documentation

Release 1.4

Scrapinghub

February 10, 2015

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 sever 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;
- transparently plug into existing software using Proxy interface;
- get detailed rendering info in HAR format.

CHAPTER 1

Documentation

1.1 Installation

1.1.1 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).

1.1.2 OS X + Docker

1. Install Docker (via Boot2Docker).
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 boot2docker:

   ```
   $ boot2docker ip

   The VM's Host only interface IP address is: 192.168.59.103
   ```

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

1.1.3 Ubuntu 12.04 (manual way)

1. Install system dependencies:

```
$ sudo add-apt-repository -y ppa:pi-rho/security
$ sudo apt-get update
$ sudo apt-get install libre2-dev
$ sudo apt-get install netbase ca-certificates python \
                       python-dev build-essential libicu48 \
                       xvfb libqt4-webkit python-twisted python-qt4
```

2. TODO: install Python dependencies using pip, clone repo, chdir to it, start splash.

To run the server execute the following command:

```
python -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:

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

Requirements

```
# install PyQt4 (Splash is tested on PyQT 4.9.x and PyQT 4.11.x)
# and the following packages:
twisted
qt4reactor
psutil
adblockparser >= 0.2
-e git+https://github.com/axiak/pyre2.git#egg=re2
xvfbwrapper
Pillow

# for scripting support
lupa >= 1.1

# the following libraries are only required by tests
pytest
pyOpenSSL
requests >= 1.0
jsonschema >= 2.0
strict-rfc3339
```

1.1.4 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:1.2.1`.

1.1.5 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 *Custom Lua 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.

Splash in Production

In production you may want to daemonize Splash, start it on boot and restart on failures. Since Docker 1.2 an easy way to do this is to use `--restart` and `-d` options together:

```
$ docker run -d -p 8050:8050 --restart=always scrapinghub/splash
```

Another way to do that is to use standard tools like upstart, systemd or supervisor.

1.2 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* (WARNING: it is still experimental). 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.

The following endpoints are supported:

1.2.1 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.

If given, base HTML content will be feched from the URL given in the url argument, and render using this as the base url.

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
```

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 rendering when doing full-page rendering (see *render_all*).

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

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.

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 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 downoad 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.

Examples

Curl example:

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

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.

1.2.2 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&heig
```

1.2.3 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 and response contents; only meta-information like headers and timings is available.

Arguments for this endpoint are the same as for render.html.

1.2.4 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.png 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.

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.

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
            "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
            "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&ifr

# 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&wid

# 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("a
    'http://localhost:8050/render.json?url=http://domain.com&script=1&console=1'
```

1.2.5 execute

Warning: This endpoint is experimental. API could change in future releases.

Execute a custom rendering script and return a result.

render.html, *render.png*, *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.

1.2.6 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.

In *Splash-as-a-proxy* mode use `X-Splash-js-source` header instead of a POST request.

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
    '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.

1.2.7 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,eas
```

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.

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 from github.

1.2.8 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

[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=mywe
```

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.

1.2.9 Splash as a Proxy

Splash supports working as HTTP proxy. In this mode all the HTTP requests received will be proxied and the response will be rendered based in the following HTTP headers:

X-Splash-render [string][required] The render mode to use, valid modes are: html, png and json. These modes have the same behavior as the endpoints: render.html, render.png and render.json respectively.

X-Splash-js-source [string] Allow to execute custom javascript code in page context. See *Executing custom Javascript code within page context*.

X-Splash-js [string] Same as *'js'* argument for render.html. See *Javascript Profiles*.

X-Splash-timeout [string] Same as *'timeout'* argument for render.html.

X-Splash-wait [string] Same as *'wait'* argument for render.html.

X-Splash-proxy [string] Same as *'proxy'* argument for render.html.

X-Splash-filters [string] Same as *'filters'* argument for render.html.

X-Splash-allowed-domains [string] Same as *'allowed_domains'* argument for render.html.

X-Splash-viewport [string] Same as *'viewport'* argument for render.html.

X-Splash-images [string] Same as *'images'* argument for render.html.

X-Splash-width [string] Same as *'width'* argument for render.png.

X-Splash-height [string] Same as *'height'* argument for render.png.

X-Splash-render-all [string] Same as *'render_all'* argument for render.png.

X-Splash-scale-method [string] Same as *'scale_method'* argument for render.png.

X-Splash-html [string] Same as *'html'* argument for render.json.

X-Splash-png [string] Same as *'png'* argument for render.json.

X-Splash-iframes [string] Same as *'iframes'* argument for render.json.

X-Splash-script [string] Same as *'script'* argument for render.json.

X-Splash-console [string] Same as *'console'* argument for render.json.

X-Splash-history [string] Same as *'history'* argument for render.json.

X-Splash-har [string] Same as *'har'* argument for render.json.

Note: Proxying of HTTPS requests is not supported.

Curl examples:

```
# Display json stats
curl -x localhost:8051 -H 'X-Splash-render: json' \
    http://www.domain.com

# Get the html page and screenshot
curl -x localhost:8051 \
    -H "X-Splash-render: json" \
    -H "X-Splash-html: 1" \
    -H "X-Splash-png: 1" \
    http://www.mywebsite.com

# Execute JS and return output
curl -x localhost:8051 \
    -H 'X-Splash-render: json' \
    -H 'X-Splash-script: 1' \
    -H 'X-Splash-js-source: function test(x){ return x; } test("abc");' \
    http://www.domain.com

# Send POST request to site and save screenshot of results
curl -X POST -d '{"key":"val"}' -x localhost:8051 -o screenshot.png \
    -H 'X-Splash-render: png' \
    http://www.domain.com
```

Splash proxy mode is enabled by default; it uses port 8051. To change the port use `--proxy-portnum` option:

```
python -m splash.server --proxy-portnum=8888
```

To disable Splash proxy mode run splash server with `--disable-proxy` option:

```
python -m splash.server --disable-proxy
```

1.3 Splash Scripts Tutorial

Warning: Scripting support is an experimental feature for early adopters; API could change in future releases.

1.3.1 Intro

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

To execute a script and get the result back send it to *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 worths 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 state of the arts Deep Learning framework Torch7. It is easy to get started and there are good online resources available like Learn Lua in 15 minutes tutorial and Programming in Lua book.

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 *execute* endpoint in a `lua_source` argument, Splash will go to example.com website, wait until it loads, then wait 0.5s more, then get 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).

1.3.2 Entry Point: the "main" Function

The script must provide "main" function; this function is called by Splash. The result of this function is returned as an HTTP response. 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. This will return `{"hello":"world!"}` string as an HTTP response:

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

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%
hello
```

"main" function receives an object that allows to control the "browser tab". All Splash features are exposed using this object. By a 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
```

1.3.3 Where Are My Callbacks?

Here is a part of the first example:

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

The code looks like a 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; this enables a more straightforward code.

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:

```
users = {'PhantomJS', 'ariyahidayat'}

function followers(splash, user)
    local ok, msg = splash:go('http://mobile.twitter.com/' .. user)
```

```
    if not ok then
        return "?"
    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] = followers(splash, user)
    end
    return result
end

function main(splash)
    local users = process(splash, users)
    return {users=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 are ten unique PhantomJS features.

1.3.4 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. It is a common criticism of coroutines/greenlets; check e.g. this article for a good description of the problem. However, we feel that in Splash scripts negative effects are not quite there: scripts are meant to be small, shared state is minimized, and an API is designed to execute a single command at time, so in most cases the control flow is linear.

If you want to be safe then think of all `splash` methods as of 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.

Currently async methods are *splash:go*, *splash:wait*, *splash:wait_for_resume*, *splash:http_get* and *splash:set_content*; *splash:autoload* becomes async when an URL is passed. 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.

1.3.5 Calling Splash Methods

Unlike many languages, in Lua methods 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}`:

```
-- 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 the convenience all `splash` methods are designed to support both styles of calling. But note that generally this convention is not followed in Lua. There are no "real" named arguments in Lua, and most Lua functions (including the ones from the standard library) choose to support only one style of calling. Check http://www.lua.org/pil/5.3.html for more info.

1.3.6 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"))
```

1.3.7 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
```

1.3.8 Custom Lua 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-modul
```

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
```

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
```

1.3.9 Timeouts

By default Splash aborts script execution after the timeout (30s by default). To override the timeout value use *'timeout'* argument of the `/execute` endpoint.

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. It is quite typical for scripts to work longer than 60s, so if you use Splash scripts it is recommended to explicitly set the maximum possible timeout by starting Splash with `--max-timeout` command line option:

```
$ python -m splash.server --max-timeout 3600
```

Note: See *Passing Custom Options* if you use Docker to run Splash.

1.4 Splash Scripts Reference

Warning: Scripting support is an experimental feature for early adopters; API could change in future releases.

`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.

1.4.1 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}`

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.

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

Three types of errors are reported (`ok` can be `nil` in 3 cases):

1. There is nothing to render. This can happen if a host doesn't exist, server dropped connection, etc. In this case `reason` is `"error"`.
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"`.

Error handling example:

```
local ok, reason = splash:go("http://example.com")
if not ok:
    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*.

`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*.

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.

1.4.2 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"`.

Returns: `ok, reason` pair. If `ok` is `nil` then the timer was stopped prematurely, and `reason` contains a string with a reason. Possible reasons are `"error"` and `"redirect"`.

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 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
```

1.4.3 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.

Example:

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

    splash:go(splash.args.url)
    return get_div_count()
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 strings, numbers, booleans and tables can be passed as arguments; they are converted to JS strings/numbers/booleans/objects. 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. Lua → JavaScript conversion rules:

Lua	JavaScript
string	string
number	number
boolean	boolean
table	Object
nil	undefined

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. JavaScript → Lua conversion rules:

JavaScript	Lua
string	string
number	number
boolean	boolean
Object	table
Array	table
`undefined`	`nil`
`null`	`""` (an empty string)
Date	string: date's ISO8601 representation, e.g. `1958-05-21T10:12:00Z`
RegExp	table `{_jstype='RegExp', caseSensitive=true/false, pattern='my-regexp'}`
function	an empty table `{}` (don't rely on it)

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.

Note: The rule of thumb: if an argument or a return value can be serialized via JSON, then it is fine.

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*.

1.4.4 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.

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) may fail:

```
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. It is a waste of resources, and it could trigger internal protection measures; *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*.

1.4.5 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.

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*.

1.4.6 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.

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*.

1.4.7 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.

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))
    return splash:evaljs("$.fn.jquery")  -- return jQuery 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.

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

1.4.8 splash:http_get

Send an HTTP 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 Lua table with the response in HAR response format.

Example:

```
local reply = splash:http_get("http://example.com")
-- reply.content.text contains raw HTML data
-- reply.status contains HTTP status code, as a number
-- see HAR docs for more info
```

In addition to all HAR fields the response contains "ok" flag which is true for successful responses and false when error happened:

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

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

1.4.9 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.

Example:

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

1.4.10 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).

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 10 pages on a website, and for each page store initial HTML snapshot and an HTML snapshot after waiting 0.5s:

```
-- Given an url, this function returns a table with
-- two HTML snapshots: HTML right after page is loaded,
-- and HTML after waiting 0.5s.
function page_info(splash, url)
    local ok, msg = splash:go(url)
    if not ok then
        return {ok=false, reason=msg}
    end
    local res = {before=splash:html()}
    assert(splash:wait(0.5))  -- this shouldn't fail, so we wrap it in assert
```

```
    res.after = splash:html() -- the same as res["after"] = splash:html()
    res.ok = true
    return res
end

-- visit first 10 http://example.com/pages/<num> pages,
-- return their html snapshots
function main(splash)
    local result = {}
    for i=1,10 do
       local url = "http://example.com/pages/" .. page_num
       result[i] = page_info(splash, url)
    end
    return result
end
```

1.4.11 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'}`

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'`

Returns: PNG screenshot data.

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.

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.

If the result of `splash:png()` is returned directly as a result of "main" function, the screenshot is returned as binary data:

```
-- A simplistic implementation of render.png endpoint
function main(splash)
    splash:set_result_content_type("image/png")
    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
```

If your script returns the result of `splash:png()` in a top-level `"png"` key (as we've done in a previous example) then Splash UI will display it as an image.

1.4.12 splash:har

Signature: `har = splash:har()`

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

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
```

1.4.13 splash:history

Signature: `entries = splash:history()`

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

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

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
```

1.4.14 splash:url

Signature: `url = splash:url()`

Returns: the current URL.

1.4.15 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.

Example result:

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

1.4.16 splash:add_cookie

Add a cookie.

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

Example:

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

1.4.17 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.

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"))
```

1.4.18 splash:clear_cookies

Clear all cookies.

Signature: `n_removed = splash:clear_cookies()`

Returns a number of cookies deleted.

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

1.4.19 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.

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

1.4.20 splash:lock_navigation

Lock navigation.

Signature: `splash:lock_navigation()`

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

1.4.21 splash:unlock_navigation

Unlock navigation.

Signature: `splash:unlock_navigation()`

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.

1.4.22 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.

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
```

1.4.23 splash:set_images_enabled

Enable/disable images.

Signature: `splash:set_images_enabled(enabled)`

Parameters:

- enabled - `true` to enable images, `false` to disable them.

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:set_images_enabled(false)
    assert(splash:go("http://example.com"))
    return {png=splash:png()}
end
```

1.4.24 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.

1.4.25 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.

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*.

1.4.26 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.

`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:

```
function main(splash)
    assert(splash:go("http://example.com"))
    assert(splash:wait(0.5))
    splash:set_viewport_full()
```

```
    return {png=splash:png()}
end
```

1.4.27 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.

1.4.28 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.

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",
})
```

Named arguments are not supported for this function.

1.4.29 splash:get_perf_stats

Return performance-related statistics.

Signature: `stats = splash:get_perf_stats()`

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

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

1.4.30 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.

1.5 Splash Development

1.5.1 Contributing

Splash is free & open source. Development happens at github: https://github.com/scrapinghub/splash

1.5.2 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
```

1.5.3 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
```

1.6 Changes

1.6.1 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() command 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.

1.6.2 1.3.1 (2014-12-13)

This release fixes packaging issues with Splash 1.3.

1.6.3 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.

1.6.4 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.

1.6.5 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.

1.6.6 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.

1.6.7 1.0 (2014-07-28)

Initial release.