# Splash Documentation

***Release 1.0***

**Scrapinghub**

October 14, 2014

# Contents

CHAPTER 1

# Introduction

Splash is a javascript rendering service with a HTTP API. It runs on top of twisted and QT webkit for rendering pages.

The (twisted) QT reactor is used to make the sever fully asynchronous allowing to take advantage of webkit concurrency via QT main loop.

## 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.

## 1.2 Requirements

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

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

## 1.3 Customizing Dockerized Splash

### 1.3.1 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.

### 1.3.2 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 <filters-dir>:/etc/splash/filters scrapinghub/splash
```

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 the same way:

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

### 1.3.3 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.

CHAPTER 2

# Usage

To run the server:

```
python -m splash.server
```

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

CHAPTER 3

# API

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
```

The following endpoints are supported:

## 3.1 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)

**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 viewport=full (see later).

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

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

**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<heigth>", e.g. 800x600. It also accepts 'full' as value; viewport=full means that the whole page (possibly very tall) will be rendered. 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.

**Note:** viewport=full 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 viewport=full.

**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.

### 3.1.1 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.

## 3.2 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.

### 3.2.1 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
```

## 3.3 render.har

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

## 3.4 render.json

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

### 3.4.1 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&showsearc
            "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&showsearc
            "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'
```

CHAPTER 4

# Executing custom Javascript code within page context

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 we 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.

## 4.1 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: *Customizing Dockerized Splash*.

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.

## 4.2 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.

CHAPTER 5

# 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: *Customizing Dockerized Splash*.

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` GET 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` GET 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'* GET 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.

# Proxy Profiles

Splash supports "proxy profiles" that allows to set proxy handling rules per-request using `proxy` GET 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:** See also: *Customizing Dockerized Splash*.

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` GET argument is not specified. If you have `default.ini` profile but don't want to apply it pass `none` as `proxy` value.

# 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-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
```

CHAPTER 8

# Functional Tests

Run with:

```
nosetests
```

CHAPTER 9

# 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
```

CHAPTER 10

# Changes

## 10.1 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.

## 10.2 1.0 (2014-07-28)

Initial release.