Phantombuster Documentation

Release 1.0

Phantombuster

Nov 13, 2017

Contents

CHAPTER 1

Writing scripts

At its core, Phantombuster allows you to script “web robots” in two languages: JavaScript and CoffeeScript.

To create a new bot script, log in, go to your scripts page and simply enter a name. Click *Advanced* to select what kind of script you want to create.

Each script can be launched on our platform by one of the following commands (in fact, binaries):

- Node — Execute your scripts in V8, Chrome’s JavaScript runtime
- PhantomJS — Headless, scriptable WebKit browser (where Phantombuster got its name from)
- CasperJS — Framework built on top of PhantomJS to easily write complex navigation scenarios

You can also write your own modules to better control and optimize how your robots will navigate the web. Or use the one made by the Phantombuster team: *Nick*.

1.1 Quick start (read this!)

If you are in a hurry, please read at least the four following sections.

The best way to understand and get started quickly with Phantombuster is to try some sample scripts.

Once logged in (log in here if you haven’t already), check out some of these scripts (choose your preferred language and framework or the first one if you don’t know):

- **Nick — Phantombuster’s custom navigation module, easiest to understand and get started with:**
  - CoffeeScript sample: sample-Nick.coffee
  - JavaScript sample: sample-Nick.js
  - *API documentation*
- **CasperJS — popular choice but not everyone agrees with its step-based navigation system:**
  - CoffeeScript sample: sample-CasperJS.coffee
  - JavaScript sample: sample-CasperJS.js

  - API documentation: http://docs.casperjs.org/
- **PhantomJS — provides the "low-level" functionality on which the two previous frameworks are based:**
  - CoffeeScript sample: sample-PhantomJS.coffee
  - JavaScript sample: sample-PhantomJS.js
  - API documentation: http://phantomjs.org/api/
- **Node — by far the fastest of all but uses a completely different scripting API (and it's not a headless web browser):**
  - CoffeeScript sample: sample-Node.coffee
  - JavaScript sample: sample-Node.js
  - API documentation: https://nodejs.org/api/

When viewing a script, click *Quick Launch* in the top right corner to run it. You'll see your script execution in real-time. Just below the console output, your persistent storage is displayed (it's where your saved files will show up).

Do not hesitate to copy-paste these scripts to test more features.

1.2 Agents and scripts

Now that you launched your first few scripts, you probably noticed that they run within an **agent** (if you used the *Quick Launch* feature, the agent you used was named *Quick Launch Agent*).

Agents are configuration settings that describe how to run a certain script. They allow you to control how and when a script is launched. The combination of a script and an agent gives you a full featured "web robot" that can scrape and automate stuff on the web.

To create an agent, go to your agents page and enter a name of your choice. The most important settings of an agent are which script to launch and when to launch it. But you'll see there are a lot of other options...

1.3 In what environment do my scripts run?

Your scripts are executed in Linux containers (they are similar to very light virtual machines).

Available to you are a few gigabytes of RAM, a few gigabytes of hard disk space and a fast internet connection. These are temporary resources that are freed right after your agent finishes its job.

What's important to know is that files written on your agent's disk **will be lost when it exits**. To keep files, save them to your persistent storage using our *agent module*.

More technical details (for the nerds):

- The container engine is Docker
- Containers are running Debian
- Agents always start in `/home/phantom/agent` which is empty
- Agents run under the user `phantom`

1.4 Phantombuster's SDK

All your scripts can easily be written right on our website, in the provided CoffeeScript/JavaScript web editor.

However, you might prefer using your own editor, locally on your machine. We made Phantombuster's SDK specifically for this.

The SDK will **monitor a directory** on your disk for changes in your scripts. As soon as a change is detected, the script will be uploaded in your Phantombuster account.

First, you need to have `npm` installed. Then do this:

```
# npm install -g phantombuster-sdk
```

It will globally install the `phantombuster` command. *Discover how to use it →*

1.5 Requiring other scripts

All your scripts (and samples/libraries) can be required. The requiring script must have a `phantombuster dependencies` directive (similar to `"use strict";`) listing its dependencies.

```
"use strict";
"phantombuster command: casperjs";
"phantombuster package: 2";
// Comma separated list of dependencies
// Specify the full name (with extension)
"phantombuster dependencies: lib-Foo.js, lib-Nick-beta.coffee";

// The rest of your script...
MyLib = require("lib-Foo");
Nick = require("lib-Nick-beta");
```

1.6 Writing your own modules

When the name of a script starts with `lib`, its launch will be disabled. This allows you to safely write **reusable modules** that can later be required using `phantombuster dependencies` and then `require()`.

To create a new module, log in, go to your scripts page, select the *reusable module* tab and enter your module name.

```
// In script "lib-Foo.js"
"use strict";

module.exports = {
    foo: function() {
        console.log("bar");
    }
}
```

```
// In script "my-script.js"

"use strict";
"phantombuster command: casperjs";
"phantombuster package: 2";
"phantombuster dependencies: lib-Foo.js";
```

```
require("lib-Foo").foo(); // outputs "bar"
```

There are a few more subtleties to consider when writing your own modules →

1.7 Locking a script's launch command

If you want to make sure a script is always launched with the same command, add a `phantombuster command` directive (similar to `"use strict";`).

```
// Possible values are: casperjs, phantomjs and node
"phantombuster command: node";
"phantombuster package: 2";
"use strict";

// The rest of your script...
needle = require("needle");
```

CHAPTER 2

Modules

Below are listed all the installed modules that you can `require()` within your scripts. All modules are installed directly from NPM.

If you need us to install a specific module, contact us at contact@phantombuster.com and we'll do what we can.

To know the exact installed version of each module and the different versions of PhantomJS available, please see *Packages*.

2.1 For CasperJS, PhantomJS or Node

These modules are compatible with all the Phantombuster commands (CasperJS, PhantomJS and Node). Require them at will.

- phantombuster — Access Phantombuster's services (download and save files, send emails, ...)
- async — Higher-order functions and common patterns for asynchronous code
- bluebird — Full featured Promises/A+ implementation
- deep-diff — Calculate object differences
- jstoxml, jsontoxml, easyxml, data2xml — Convert objects or JSON to XML
- linq — Microsoft's Language Integrated Query (LINQ) for JavaScript
- mime — Mime-type mapping
- pretty-data2, pretty-data — Pretty-print or minify XML, JSON, CSS and SQL files
- resemblejs — Image analysis and comparison with HTML5
- underscore — Functional programming helper library
- xmltojson — Convert XML to objects or JSON

2.2 For CasperJS and PhantomJS only

These modules are only compatible with CasperJS or PhantomJS.

- papaparse — Fast, in-browser CSV parser
- whatwg-fetch — HTTP client (`window.fetch` polyfill)

2.3 For Node only

These modules are only compatible with Node.

- aws-sdk — The official AWS SDK for JavaScript
- fast-csv — CSV parser and writer
- mkdirp, fs-extra — File system utilities
- mongodb, mongoose — MongoDB
- needle, request — HTTP client
- node-fetch — `window.fetch` ported to node.js
- through, through2 — Simplified stream construction
- xml2js, xml2json — XML to JavaScript object converter

2.4 Other modules

These modules are also available and compatible with Node. However, we have not yet tested them for CasperJS or PhantomJS compatibility.

- axios — Promise based HTTP client
- babel-polyfill — ES2015 environment emulation
- cheerio — Implementation of core jQuery for the server
- es6-promise — Tools for organizing asynchronous code (ES6 Promise polyfill)
- is-my-json-valid, jsen — JSON Schema validation
- jsdom — Implementation of the DOM and HTML standards
- lodash — JavaScript utility library
- moment — Parse, validate, manipulate, and display dates
- once — Only call a function once
- pluralize — Pluralize and singularize any word
- q — Library for promises (CommonJS/Promises/A,B,D)
- qs — Querystring parser
- when — Lightweight Promises/A+ and when() implementation

2.5 Injectable modules

These modules come preloaded on all agent disks. They can be injected in visited pages for easier DOM manipulation (mainly for scraping and automation purposes).

Use *nick.inject()* or page.injectJs() from PhantomJS to add them in web pages.

- `../injectables/jquery-2.2.3.min.js` - jQuery 2 (84kB)
- `../injectables/jquery-3.0.0.min.js` - jQuery 3 (85kB)
- `../injectables/underscore-1.8.3.min.js` - Underscore (17kB)
- `../injectables/lodash-core-4.13.1.min.js` - Lodash (core build, 12kB)
- `../injectables/lodash-full-4.13.1.min.js` - Lodash (full build, 67kB)

For example, to use jQuery in a page that doesn't include it already, you could do this:

```
nick.inject('../injectables/jquery-3.0.0.min.js', function(err) {
    if (!err)
        console.log('jQuery injected in current page')
});
```

2.6 Writing your own modules

When the name of a script starts with `lib`, its launch will be disabled. This allows you to safely write **reusable modules** that can later be required using `phantombuster dependencies` and then `require()`.

To create a new module, log in, go to your scripts page, select the *reusable module* tab and enter your module name.

```
// In script "lib-Foo.js"
"use strict";

module.exports = {
    foo: function() {
        console.log("bar");
    }
}
```

```
// In script "my-script.js"

"use strict";
"phantombuster command: casperjs";
"phantombuster package: 2";
"phantombuster dependencies: lib-Foo.js";

require("lib-Foo").foo(); // outputs "bar"
```

Please note that there is no need to specify your script extension when you require it. Both JavaScript and CoffeeScript modules can be required without their extensions.

There are a few more subtleties to consider when writing your own modules:

- If you write a CoffeeScript module and want to require it from a JavaScript script using Phantombuster package 1, you will need to add this `require('coffee-script/register')` to the requiring script. Package 2+ scripts do not need this because in this case CoffeeScript is transpiled to JavaScript before the bot is started.

- If you want to use CasperJS internal modules in your own module, you will need to add this `require = patchRequire(require);`. Obviously the requiring script must be started with the `casperjs` command in this case. More info here

CHAPTER 3

Packages

Phantombuster allows you to select between multiple versions of PhantomJS and CasperJS. Available modules are bundled together in what we call "packages". Each package is assigned a number that you can specify within your scripts.

Very often a script will start like this:

```
'use strict';
'phantombuster command: casperjs';
'phantombuster package: 2'; // use the 2nd module package (PhantomJS 2.1.1)
'phantombuster dependencies: lib-Nick-beta.coffee';

buster = require('phantombuster').create();
Nick = require('lib-Nick-beta')
nick = new Nick({
    printNavigation: true,
    userAgent: 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,␣
↪like Gecko) Chrome/42.0.2311.135 Safari/537.36 Edge/12.246'
});
```

The important line here is the third one. It sets up the script's environment with a specific package.

3.1 Package 2: PhantomJS 2.1.1

This is the recommended package. Use it! To do so, add

```
'phantombuster package: 2';
```

at the beginning of your script.

This package was added July 18th 2016. It includes:

- **PhantomJS 2.1.1** (installed from phantomjs-prebuilt@2.1.7)
- **CasperJS 1.1.2** (installed from casperjs@1.1.2)

- **Node 6.x** (installed from NodeSource setup_6.x)

Exact versions of all bundled modules:

```
async@2.0.0-rc.6
aws-sdk@2.4.6
axios@0.13.1
bluebird@3.4.1
babel-polyfill@6.9.1
cheerio@0.20.0
coffee-script@1.10.0
data2xml@1.2.5
deep-diff@0.3.4
easyxml@2.0.1
es6-promise@3.2.1
fast-csv@2.0.1
fs-extra@0.30.0
is-my-json-valid@2.13.1
jsdom@9.4.1
jsen@0.6.1
jsontoxml@0.0.11
jstoxml@0.2.3
linq@3.0.5
lodash@4.13.1
mime@1.3.4
mkdirp@0.5.1
moment@2.14.1
mongodb-extended-json@1.6.3
mongodb@2.2.1
mongoose@4.5.3
needle@1.0.0
node-fetch@1.5.3
once@1.3.3
papaparse@4.1.2
pluralize@3.0.0
pretty-data2@0.40.1
pretty-data@0.40.0
q@1.4.1
qs@6.2.0
request@2.73.0
resemblejs@2.2.1
through2@2.0.1
through@2.3.8
underscore@1.8.3
whatwg-fetch@1.0.0
when@3.7.7
xml2js@0.4.17
xml2json@0.9.1
xmltojson@1.1.0
```

3.2 Package 1: PhantomJS 1.9.8

This is the default package. Remove any `phantombuster package` directive from your script to use this package.

Alternatively, your could force the package to `1` by putting this at the beginning of your script:

```
'phantombuster package: 1';
```

but it’s not necessary.

It’s not recommended to use this package because it comes with old versions of PhantomJS and CasperJS.

This is the original Phantombuster package. It includes:

- **PhantomJS 1.9.8** (installed from the **deprecated** phantomjs@1.9.20)
- **CasperJS 1.1.0-beta3** (installed from the **deprecated** casperjs@1.1.0-beta3)
- **Node 6.x** (installed from NodeSource setup_6.x)

Exact versions of all bundled modules:

```
async@2.0.0-rc.4
aws-sdk@2.3.9
axios@0.13.1
bluebird@3.4.1
babel-polyfill@6.9.1
cheerio@0.20.0
coffee-script@1.10.0
data2xml@1.2.5
deep-diff@0.3.4
easyxml@2.0.1
es6-promise@3.2.1
fast-csv@2.0.0
fs-extra@0.30.0
is-my-json-valid@2.13.1
jsdom@9.3.0
jsen@0.6.1
jsontoxml@0.0.11
jstoxml@0.2.3
linq@3.0.5
lodash@4.13.1
mime@1.3.4
mkdirp@0.5.1
moment@2.14.1
mongodb-extended-json@1.6.3
mongodb@2.1.18
mongoose@4.5.3
needle@1.0.0
node-fetch@1.5.3
once@1.3.3
papaparse@4.1.2
pluralize@3.0.0
pretty-data2@0.40.1
pretty-data@0.40.0
q@1.4.1
qs@6.2.0
request@2.72.0
resemblejs@2.2.0
through2@2.0.1
through@2.3.8
underscore@1.8.3
whatwg-fetch@1.0.0
when@3.7.7
xml2js@0.4.16
```

```
xml2json@0.9.1
xmltojson@1.1.0
```

CHAPTER 4

Agent Module

The Agent Module (also called `buster`) is a special module made by the Phantombuster team. It provides some cool features and is linked to your Phantombuster account.

Here's a short list of what you can do with our module:

- Save files to your persistent storage
- Download files to your agent's disk
- Indicate the progress of your agent
- Set or reset the time limit of your agent
- Send emails
- Interact with MongoDB databases
- Solve CAPTCHAS
- ...

This module is compatible with all the Phantombuster commands (CapserJS, PhantomJS and Node).

4.1 Initialization

The agent module is named `phantombuster`. Use `require('phantombuster')` and call `create()` to instantiate it.

- When using Nick, Node or PhantomJS:

  ```
  buster = require('phantombuster').create();
  ```

- When using the agent module in conjunction with CasperJS, you **must** pass the CasperJS instance to `create()`. A typical CasperJS script starts like this:

```
casper = require('casper').create();
buster = require('phantombuster').create(casper);
```

4.2 Asynchronous methods

Following the philosophy of Node, most methods of the agent module are asynchronous. You have to use the callback function to know when (and if) a call finished successfully.

For example, this is bad:

```
buster.saveFolder(function() {
    console.log('Folder saved');
});
phantom.exit(); // BAD! saveFolder() will not have finished here, in fact it will not␣
↪even have started!
```

This is better:

```
buster.saveFolder(function(err) {
    if (err) {
        console.log('Error when saving folder: ' + err);
        phantom.exit(1);
    } else {
        phantom.exit(0);
    }
});
```

4.3 CasperJS step control

If the agent module is instantiated with a CasperJS instance passed in `create()`, its methods will block the current navigation step for your convenience. For example:

```
casper = require('casper').create();
buster = require('phantombuster').create(casper); // pass the CasperJS instance

casper.start('https://example.com', function() {

    // methods ask CasperJS to not go to the next step until they are finished
    buster.saveText('foo bar baz', 'foo.txt', function() {
        console.log('Text saved!');
    });

    console.log('Navigation step 1');
});

// steps will wait for the end of all the previous agent module calls
casper.then(function() {
    console.log('This is executed a few ticks after saveText()');
});

casper.run(function() {
    console.log('And this is executed last');
    casper.exit()
});
```

This script will output:

```
Navigation step 1
Text saved!
This is executed a few ticks after saveText()
And this is executed last
```

Internally, the agent module makes calls to *blockSteps()* and *unblockSteps()*. You can also use these methods for stopping the execution of steps when calling your own asynchronous functions or when using `async` for example.

4.4 agentId

```
buster.agentId
```

Contains the ID of the currently running agent as a `Number`. This is useful for making requests to the Phantombuster API from within the agent.

4.5 apiKey

```
buster.apiKey
```

Contains your Phantombuster API key as a `String`. This is useful for making requests to the Phantombuster API from within the agent.

4.6 argument

```
buster.argument
```

Contains the agent's argument as a `PlainObject`. On Phantombuster, each agent receives a JSON object as argument, which can be set each time they are launched.

4.7 blockSteps()

```
buster.blockSteps()
```

CasperJS only.

Stops the execution of CasperJS steps until *unblockSteps()* is called (behind the scenes, it uses the same system as `casper.wait()`).

This is very useful when calling asynchronous functions if you want to wait for the callback before continuing your CasperJS navigation. Simply call `blockSteps()` before the asynchronous call, and `unblockSteps()` in the callback.

After, unblockSteps() must be called the same number of times, otherwise navigation will be blocked.

4.8 containerId

```
buster.containerId
```

Contains the ID of the currently running container as a `Number`. This is useful for making requests to the Phantombuster API from within the agent.

4.9 download()

```
buster.download(url [, saveAs, headers, callback])
```

Downloads a distant file to your agent's disk (not to your persistent storage). If you do not save the file to your persistent storage (see *Agent Module: Store files*), **it will be lost when your agent exits**.

This method is asynchronous and returns nothing. Use the callback to know when it has finished.

`url (String)` URL of the file to be downloaded.

- `https://www.google.com/images/srpr/logo11w.png`
- `http://soundcloud.com/` (you'll get the HTML content of their homepage)

`saveAs (String)` Where to put the file on your agent's disk (optional). By default, the name will be taken from `url` and the file will be saved in the current working directory on your agent's disk. If a file with the same name already exists, it is overwritten.

- `foo/` (saves `http://example.com/baz/bar.png` as `foo/bar.png`)
- *null* (saves `http://example.com/foo/bar.png` as `bar.png`)
- `foo/` (fails on `http://example.com/` with `could not determine filename`)
- `foo/a` (saves `http://example.com/bar.png` as `foo/a`)

Intermediate directories are not created automatically on your agent's disk.

`headers (PlainObject)` HTTP headers to use when requesting the file (optional). Cookies are automatically set when using CasperJS or PhantomJS.

`callback (Function(String err, String path))` Function to call when finished (optional). When there is no error, `err` is *null* and `path` contains the path to the file on your agent's disk.

4.10 getAgentObject()

```
buster.getAgentObject([agentId,] callback)
```

Gets the object of an agent.

This method is asynchronous and returns nothing. Use the callback to know when it has finished.

`agentId (String)` ID of the agent from which to get the object (optional). By default, this is the ID of the currently running agent.

`callback (Function(String err, PlainObject object))` Function to call when finished. When there is no error, `err` is *null* and `object` is a valid object (which may be empty but never *null*).

4.11 getGlobalObject()

```
buster.getGlobalObject(callback)
```

Gets the global object of your account.

This method is asynchronous and returns nothing. Use the callback to know when it has finished.

callback (Function(String err, PlainObject object)) Function to call when finished. When there is no error, `err` is *null* and `object` is a valid object (which may be empty but never *null*).

4.12 overrideTimeLimit()

```
buster.overrideTimeLimit(seconds [, callback])
```

Overrides the execution time limit of the agent. When the execution time reaches the specified number of seconds, the agent is stopped.

This method is asynchronous and returns nothing. Use the callback to know when it has finished.

seconds (Number) New time limit of the agent in seconds (integer), or `0` to disable the time limit. If the specified number of seconds is already lower than the current execution time, the agent is stopped right away.

callback (Function(String err)) Function to call when finished (optional). When there is no error, `err` is *null*.

4.13 proxyAddress

```
buster.proxyAddress
```

Contains the proxy address currently being used by your agent as a `String` (PhantomJS/CasperJs only), or an empty string if there is no proxy in use. This is useful to know which proxy was selected from a pool.

4.14 setAgentObject()

```
buster.setAgentObject([agentId,] object [, callback])
```

Sets (in fact, **replaces**) the object of an agent. It's recommended to first fetch the object with *getAgentObject()* (to update it) because **this method overwrites the whole object**.

This method is asynchronous and returns nothing. Use the callback to know when it has finished.

agentId (Number) ID of the agent which will get its object set (optional). By default, this is the ID of the currently running agent.

object (PlainObject) Object to save.

callback (Function(String err)) Function to call when finished (optional). When there is no error, `err` is *null*.

4.15 setGlobalObject()

```
buster.setAgentObject(object [, callback])
```

Sets (in fact, **replaces**) the global object of your account. It's recommended to first fetch the global object with *getGlobalObject()* (to update it) because **this method overwrites the whole object**.

This method is asynchronous and returns nothing. Use the callback to know when it has finished.

object (PlainObject) Object to save.

callback (Function(String err)) Function to call when finished (optional). When there is no error, `err` is *null*.

4.16 solveCaptcha()

```
buster.solveCaptcha(selector [, casperInstance], callback)
```

This method is only available with Nick or CasperJS.

Tries to solve a CAPTCHA image. This method takes a screenshot of the area indicated by `selector` and sends it to one of our partners for solving.

If your CAPTCHA image is trivial, an OCR algorithm will quickly return the text, otherwise a human will solve it. This process generally takes less than 30 seconds and accuracy is >90%.

When a result string is returned, 1 is substracted from your daily CAPTCHA counter. In approximately 10% of the cases the result will be incorrect — retry at will.

This method is asynchronous and returns nothing. Use the callback to know when it has finished.

selector (String) CSS3 selector pointing to the CAPTCHA image.

casperInstance (CasperJS) CasperJS instance that will be used for capturing the image (optional). When using Nick, simply put `nick.casper` here. Ignore this parameter if you called `create()` with a CasperJS instance already.

callback (Function(String err, String result)) Function to call when finished. When there is no error, `err` is *null* and `result` contains the solved CAPTCHA text.

4.17 solveCaptchaBase64()

```
buster.solveCaptchaBase64(base64String, callback)
```

Tries to solve a CAPTCHA image. This method takes Base64 encoded image and sends it to one of our partners for solving.

If your CAPTCHA image is trivial, an OCR algorithm will quickly return the text, otherwise a human will solve it. This process generally takes less than 30 seconds and accuracy is >90%.

When a result string is returned, 1 is substracted from your daily CAPTCHA counter. In approximately 10% of the cases the result will be incorrect — retry at will.

This method is asynchronous and returns nothing. Use the callback to know when it has finished.

base64String (String) CAPTCHA image to solve. Can be pure Base64 or a Data URI Scheme string starting with `data:`.

callback (Function(String err, String result)) Function to call when finished. When there is no error, `err` is *null* and `result` contains the solved CAPTCHA text.

4.18 solveCaptchaImage()

```
buster.solveCaptchaImage(urlOrPath [, headers], callback)
```

Tries to solve a CAPTCHA image. This method takes an URL or a path of an image and sends it to one of our partners for solving.

If your CAPTCHA image is trivial, an OCR algorithm will quickly return the text, otherwise a human will solve it. This process generally takes less than 30 seconds and accuracy is >90%.

When a result string is returned, 1 is substracted from your daily CAPTCHA counter. In approximately 10% of the cases the result will be incorrect — retry at will.

This method is asynchronous and returns nothing. Use the callback to know when it has finished.

urlOrPath (String) URL or path of the CAPTCHA image to be solved.

headers (PlainObject) HTTP headers to use when requesting the image (optional). Cookies are automatically set when using CasperJS or PhantomJS.

callback (Function(String err, String result)) Function to call when finished. When there is no error, `err` is *null* and `result` contains the solved CAPTCHA text.

4.19 unblockSteps()

```
buster.unblockSteps()
```

CasperJS only.

Signals CasperJS to continue the execution of its steps. Goes in pair with *blockSteps()*.

This method must be called the same number of times blockSteps() was called, otherwise navigation will be blocked.

CHAPTER 5

Agent Module: Return data

These four methods allow you to return data from your agents in different ways.

Before these methods can be used, you need to *require() and create() the agent module*.

5.1 mail()

```
buster.mail(subject, text [, to, callback])
```

Sends an email from your agent and substracts 1 to your daily email counter.

This method is asynchronous and returns nothing. Use the callback to know when it has finished.

subject (String) Subject of the email.

text (String) Plain text contents of the email.

to (String) Where to send the email (optional). When omitted, the email will be sent to the address associated with your Phantombuster account.

callback (Function(String err)) Function to call when finished (optional). When there is no error, `err` is *null*.

5.2 notify()

```
buster.notify(message [, options , callback])
```

Sends a push notification to your device(s) using Pushover. For this call to work, you must have set a Pushover user key in your settings and have installed a Pushover client on at least one of your devices.

This method is asynchronous and returns nothing. Use the callback to know when it has finished.

message (String) Text contents of the notification.

options (PlainObject) Additionnal parameters to send to Pushover. Get the full details at the Pushover API documentation.

- `device` - your device name to send the message directly to that device, rather than all of your devices (multiple devices may be separated by a comma)
- `title` - your message's title, otherwise *Phantombuster* is used
- `url` - a supplementary URL to show with your message
- `url_title` - a title for your supplementary URL, otherwise just the URL is shown
- `priority` - send as `-2` to generate no notification/alert, `-1` to always send as a quiet notification or `1` to display as high-priority and bypass your quiet hours
- `timestamp` - a Unix timestamp of your message's date and time to display, rather than the time the message is received by Pushover
- `sound` - the name of one of the sounds supported by device clients to override your default sound choice

Note: at the moment this method does not support the receipt system of Pushover (`priority` set to `2`).

callback (Function(String err)) Function to call when finished (optional). When there is no error, `err` is *null*.

5.3 progressHint()

```
buster.progressHint(progress [, label])
```

Reports the progress state of the agent. This affects the width and content of the progress bar displayed in the agent console on Phantombuster.

This is useful for debugging purposes and is not required for the agent to function properly. Sometimes it's just nice to see the progress of your agent in real-time.

This method returns nothing and has no callback.

progress (Number) Progress float value between `0` and `1`. `1` means 100% of the work was completed, and `0` means 0%.

label (String) Optional textual description of the state of your agent (clipped to 50 characters). This shows up as a text inside the progress bar displayed in the agent console.

5.4 setResultObject()

```
buster.setResultObject(object [, callback])
```

Sets (in fact, **replaces**) the result object of your agent. Think of the result object as the output value of your agent. This is useful for returning a small set of JSON fields containing whatever your agent might want to return when it exits.

Use this method in conjunction with the *launch* API endpoint (with `output` set to `result-object`) to launch **and** get the result of your agent in one single HTTP request.

This method is asynchronous and returns nothing. Use the callback to know when it has finished.

object (PlainObject) Object to set as result object.

callback (Function(String err)) Function to call when finished (optional). When there is no error, `err` is *null*.

CHAPTER 6

Agent Module: Store files

These four methods allow you to store files in your persitent storage.

Files that are on your agent's disk but not saved to your persistent storage **will be lost** when your agent exits.

Before these methods can be used, you need to *require() and create() the agent module*.

6.1 save()

```
buster.save(urlOrPath [, saveAs, headers, callback])
```

Saves a distant or local file to your persistent storage.

This method is asynchronous and returns nothing. Use the callback to know when it has finished.

`urlOrPath` (`String`) URL or path of the file to be saved.

- `https://www.google.com/images/srpr/logo11w.png` (from the web)
- `foo/my_screenshot.jpg` (from your agent's disk)
- `http://soundcloud.com/` (you'll get the HTML content of their homepage)

When saving a distant file, the MIME type is taken from the `Content-Type` HTTP header (if present). When saving a local file, the MIME type is guessed from the file extension (if this fails, no MIME type is set).

`saveAs` (`String`) Where to put the file on your persistent storage (optional). By default, the name will be taken from `urlOrPath` and the file will be saved at the root of your agent's folder in your persistent storage. If a file with the same name already exists, it is overwritten.

- `foo/` (saves `http://example.com/baz/bar.png` as `foo/bar.png`)
- *null* (saves `http://example.com/foo/bar.png` as `bar.png`)
- `foo/` (fails on `http://example.com/` with `could not determine filename`)
- `foo/a` (saves `http://example.com/bar.png` as `foo/a`)

You do not need to create any intermediate directory (`a/b/c/d/e.jpg` will work).

`headers` (**`PlainObject`**) HTTP headers to use when requesting the file (optional). Cookies are automatically set when using CasperJS or PhantomJS.

`callback` (**`Function(String err, String url)`**) Function to call when finished. When there is no error, `err` is *null* and `url` contains the full URL to the file on your persistent storage.

6.2 saveBase64()

```
buster.saveBase64(base64String, saveAs [, mime, callback])
```

Saves a Base64 encoded file to your persistent storage.

This method is asynchronous and returns nothing. Use the callback to know when it has finished.

`base64String` (**`String`**) Contents of the file to save. Can be pure Base64 or a Data URI Scheme string starting with `data:`.

`saveAs` (**`String`**) Where to put the file on your persistent storage. If a file with the same name already exists, it is overwritten.

- `file.jpg`
- `any/sub/directory/file.png`
- `dir/` (fails because no file name was given)

You do not need to create any intermediate directory (`a/b/c/d` will work).

`mime` (**`String`**) MIME type of the file being saved (optional). By default it is guessed either from the Data URI Scheme string or from the file extension of the `saveAs` parameter (if this fails, no MIME type is set).

- `image/jpeg`
- `image/png`
- `image/svg+xml`

`callback` (**`Function(String err, String url)`**) Function to call when finished (optional). When there is no error, `err` is *null* and `url` contains the full URL to the file in your persistent storage.

6.3 saveFolder()

```
buster.saveFolder([path, saveAs, callback])
```

Saves a folder from your agent's disk to your persistent storage.

This method is asynchronous and returns nothing. Use the callback to know when it has finished.

`path` (**`String`**) Path of the folder to save (optional, defaults to `.`).

- `.` (everything from your current working directory)
- `any/sub/../sub/directory`

Each file has its MIME type guessed from its extension (if this fails, no MIME type is set).

saveAs (String) Where to put the folder on your persistent storage (optional). By default, the folder will be saved at the root of your agent's folder in your persistent storage. If files with the same name already exist, they are overwritten.

- `/` or empty string (root of your agent's folder in your persistent storage)
- `any/sub/directory`
- `dir/foo.txt` (this will create a directory named `foo.txt`, obviously not recommended)

You do not need to create any intermediate directory (`a/b/c/d` will work).

callback (Function(String err, String url)) Function to call when finished (optional). When there is no error, `err` is *null* and `url` contains the full URL to the folder in your persistent storage.

6.4 saveText()

```
buster.saveText(text, saveAs [, mime, callback])
```

Saves a string to a file in your persistent storage.

This method is asynchronous and returns nothing. Use the callback to know when it has finished.

text (String) Contents of the file to save. Can be anything, really.

saveAs (String) Where to put the file on your persistent storage. If a file with the same name already exists, it is overwritten.

- `file.txt`
- `any/sub/directory/file.json`
- `dir/` (fails because no file name was given)

You do not need to create any intermediate directory (`a/b/c/d` will work).

mime (String) MIME type of the file being saved (optional). By default it is guessed from the file extension of the `saveAs` parameter (if this fails, no MIME type is set).

- `application/json`
- `text/csv`
- `text/html`

callback (Function(String err, String url)) Function to call when finished (optional). When there is no error, `err` is *null* and `url` contains the full URL to the file in your persistent storage.

CHAPTER 7

Agent Module: Database

These methods allow you to read and write collections of JSON documents to a MongoDB database.

Before these methods can be used, you need to *require() and create() the agent module*.

This module works with PhantomJS, CasperJS and Node. However, when using Node, we recommend using the `mongodb` module instead, which is faster and way more complete.

If your Phantombuster account has a database, you can call these methods without any configuration. If that's not the case (or if you want to use your own database), you must specifiy your MongoDB connection string with *db.setConnectionUrl()* before any other call to database methods.

7.1 Extended JSON mode

Requests can be made in either normal JSON mode (which is the default) or extended JSON mode by setting the `extendedJson` field to `true` in query options.

Internally, MongoDB supports many different types such as dates, regular expressions and IDs. These cannot be represented in plain JSON and this is where extended JSON becomes useful.

- **In normal JSON mode**, data is transferred to and from MongoDB in plain JSON. JavaScript native types `string`, `number` and `boolean` are automatically converted to their equivalent for MongoDB, but no other processing is done.

  For example, a call to *db.find()* could return this:

  ```
  [
    {
      "_id": "56211c5e7ab57e010aa49280",
      "date": "2015-10-16T15:48:46.134Z",
      "aNumberField": 42
    }
  ]
  ```

- **In extended JSON mode**, JSON data is processed before being written and after being read from the MongoDB database.

  For example, a call to *db.find()* could return this:

  ```
  [
    {
      "_id": {
        "$oid": "56211d1bf90c1e0100371c63"
      },
      "date": {
        "$date": "2015-10-16T15:51:55.304Z"
      },
      "aNumberField": 42
    }
  ]
  ```

  Please see the official MongoDB documentation on extended JSON.

Extended JSON mode can be enabled globally by instantiating the agent module like this:

```
// casperInstance is optional (applies only when using CasperJS)
buster = require('phantombuster').create(casperInstance, { extendedJson: true });
```

When extended JSON mode is enabled globally, you can still disable it for individual requests by explicitly setting the `extendedJson` field to `false` in query options.

Note: when enabled, extended JSON mode always applies for incoming **and** outgoing data (such as inserted documents and update queries).

7.2 db.count()

```
buster.db.count(collection [, options], callback)
```

Returns the number of documents in a collection.

This method is asynchronous and returns nothing. Use the callback to get the result.

`collection (String)` Name of the collection on which to execute the count operation.

`options (PlainObject)` Additionnal parameters for the request (optional, by default all documents are counted).

- `limit` (`Number`) - limit of documents to count
- `skip` (`Boolean`) - number of documents to skip for the count

`callback (Function(String err, Number count))` Function to call when finished. When there is no error, `err` is *null*.

`count` contains the result.

7.3 db.delete()

```
buster.db.delete(collection [, options, callback])
```

Deletes documents from a collection.

This method is asynchronous and returns nothing. Use the callback to know when it has finished.

collection (String) Name of the collection from which documents will be deleted.

options (PlainObject) Additionnal parameters for the request (optional, by default all documents are deleted).

- `query` (`PlainObject`) - MongoDB query for choosing which documents to delete
- `extendedJson` (`Boolean`) - enable *Extended JSON mode* for this query

callback (Function (String err, PlainObject res)) Function to call when finished (optional). When there is no error, `err` is *null*.

`res.deletedCount` contains the number of documents deleted by the query.

7.4 db.find()

```
buster.db.find(collection [, options], callback)
```

Returns documents from a collection.

This method is asynchronous and returns nothing. Use the callback to get the result.

collection (String) Name of the collection in which documents will be searched.

options (PlainObject) Additionnal parameters for the request (optional, by default all documents are returned).

- `query` (`PlainObject`) - MongoDB query for selecting documents
- `limit` (`Number`) - limits the number of documents returned
- `sort` (`PlainObject`) - how to sort the documents (field names associated with `1` or `-1`, for example `{ "name": -1 }`)
- `fields` (`PlainObject`) - fields to include (`1`) or exclude (`0`), for example: `{ "name": 1 }`
- `skip` (`Number`) - number of documents to skip (useful for pagination)
- `extendedJson` (`Boolean`) - enable *Extended JSON mode* for this query (applies for the returned array **and** for the `query` option field)

callback (Function (String err, Array documents)) Function to call when finished. When there is no error, `err` is *null*.

`documents` is an array containing all the documents found.

7.5 db.insert()

```
buster.db.insert(collection, documents [, options, callback])
```

Inserts documents into a collection.

This method is asynchronous and returns nothing. Use the callback to know when it has finished.

collection (String) Name of the collection in which documents will be inserted.

documents (PlainObject or Array) Document or array of documents to insert into the collection.

options (PlainObject) Additionnal parameters for the request (optional).

- `extendedJson` (`Boolean`) - enable *Extended JSON mode* for this query (applies for the returned object **and** for the `documents` array)

callback (Function(String err, PlainObject res)) Function to call when finished (optional). When there is no error, `err` is *null*.

- `res.insertedCount` contains the number of documents inserted by the query
- `res.insertedIds` is an array containing all the newly created MongoDB IDs

7.6 db.setConnectionUrl()

```
buster.db.setConnectionUrl(url [, callback])
```

Sets the connection string of the MongoDB server you wish to use. If your Phantombuster account has a database, there is no need to call this method.

Note: this method must be called prior to any other database methods.

This method is asynchronous and returns nothing. Use the callback to know when it has finished.

url (String) Full, canonical MongoDB connection string, for example `mongodb://user:password@host.com:27017/database`.

callback (Function(String err)) Function to call when finished (optional). When there is no error, `err` is *null*.

7.7 db.update()

```
buster.db.update(collection, update [, options, callback])
```

Updates documents stored in a collection.

This method is asynchronous and returns nothing. Use the callback to know when it has finished.

collection (String) Name of the collection in which documents will be updated.

update (PlainObject) MongoDB update parameter describing what modifications to apply.

options (PlainObject) Additionnal parameters for the request (optional, by default all documents are updated).

- `query (PlainObject)` - MongoDB filter query for selecting which documents to update
- `upsert (Boolean)` - Do an upsert instead of an update (creates a new document when no document matches the query criteria)
- `extendedJson (Boolean)` - enable *Extended JSON mode* for this query (applies for the returned object **and** for the `update` object **and** for the `query` option field)

callback (Function(String err, PlainObject res)) Function to call when finished (optional). When there is no error, `err` is *null*.

- `res.matchedCount` contains the number of documents that matched the filter query
- `res.modifiedCount` contains the number of documents that were updated by the query
- `res.upsertedCount` contains the number of inserted documents in case of an upsert
- `res.upsertedId` contains the newly created MongoDB ID in case of an upsert

CHAPTER 8

Nick

Nick is a special module made by the Phantombuster team. It provides an easy navigation/web automation/scraping system. It's based on CasperJS and is written in CoffeeScript.

CasperJS is a powerful library with many methods. But only a few are essential. Nick limits the number of methods and replaces the step-by-step CasperJS paradigm with the async/callback paradigm of Node.

Before `Nick` methods can be used, you need to `require()` a `Nick` class and instantiate it.

Your agents must be launched with the CasperJS command when using this module. Nick is not compatible with Node or PhantomJS.

8.1 Initialization

The module is named `Nick`. Use `require('lib-Nick-beta')` and create an instance to use it. A typical Nick-based script starts like this:

```
'use strict';
'phantombuster command: casperjs';
'phantombuster package: 2';
'phantombuster dependencies: lib-Nick-beta.coffee'

var Nick = require('lib-Nick-beta');
var nick = new Nick({
    printNavigation: true
});
```

An instance of Nick is similar to a browser tab. If you want to open multiple pages/tabs simultaneously, instantiate multiple Nicks.

8.2 Asynchronous methods

Following the philosophy of Node, most of Nick's methods are asynchronous. You have to use the callback function to know when (and if) a call finished successfully.

For example, this is bad:

```
nick.screenshot('image.jpg', function() {
    console.log('Screenshot taken');
});
nick.exit(); // BAD! nick.screenshot() will not have finished here!
```

This is better:

```
nick.screenshot('image.jpg', function(err) {
    if (err) {
        console.log('Error when capturing the screenshot: ' + err);
        nick.exit(1);
    } else {
        nick.exit(0);
    }
});
```

8.3 Warning: no parallelism

8.4 Options

8.5 open()

```
nick.open(url [, options], callback);
```

Opens the webpage at `url`. You can forge GET, POST, PUT, DELETE and HEAD requests.

This method is asynchronous and returns nothing. Use the `callback` to know when it has finished.

More info: http://docs.casperjs.org/en/latest/modules/casper.html#open

`url` **(`String`)** URL of the page to visit.

`options` **(`PlainObject`)** Optional request headers.

```
{
    method: 'post',
    data:   {
        'title': 'Plop',
        'body':  'Wow.'
    },
    headers: {
        'Accept-Language': 'fr,fr-fr;q=0.8,en-us;q=0.5,en;q=0.3'
    }
}
```

`callback` **(`Function()`)** Function called when finished. No arguments are returned. To know if the page opened successfully, use *waitUntilVisible()* or similar.

Example:

```
nick.open("https://phantombuster.com/cloud-services", function() {
    console.log("The page is loaded (or not!) -- use waitUntilVisible() or␣
↪similar to be sure");
    nick.exit();
});
```

8.6 wait()

```
nick.wait(duration, callback);
```

Waits for a `duration` time before calling `callback`.

This method is asynchronous and returns nothing. Use the `callback` to know when it has finished.

More info: http://docs.casperjs.org/en/latest/modules/casper.html#wait

`duration` (`Number`) Milliseconds to wait before calling `callback` function.

`callback` (`Function()`) Function called when finished. This function never fails, no arguments will be passed.

Example:

```
nick.open("https://phantombuster.com/cloud-services", function() {
    console.log('Hello')
    nick.wait(1000, function() {
        console.log('world!');
        nick.exit();
    })
});
```

8.7 waitUntilPresent()

```
nick.waitUntilPresent(selectors, timeout [, condition = "and"], callback);
```

Waits until a DOM element, matching the provided selector, is present. If the method has to wait more than `timeout` milliseconds, `callback` is called with a `"timeout"` error. By default, `condition` is set to `"and"`.

It is considered good practice to always use a `wait*()` method after a page load and before any action on selectors.

This method is asynchronous and returns nothing. Use the `callback` to know when it has finished.

More info: http://docs.casperjs.org/en/latest/modules/casper.html#waitforselector

`selectors` (`Array or String`) An array of CSS3 selectors describing the path to DOM elements.

`timeout` (`Number`) Milliseconds to wait before calling `callback` function with an error.

`condition` (`String`) When `selectors` is an array, this argument lets you choose how to wait for the elements. If `condition` is `"and"`, the method will wait for the presence of all `selectors` in the DOM. Otherwise if `condition` is `"or"`, the method will wait until any `selector` of the array is present in the DOM.

`callback` (`Function(String err, String sel)`) Function called when finished. When there is no error, `err` is null.

- **In case of success (`err` is *null*):**

 - if `condition` is `"and"` then `sel` is *null* because all selectors are present
 - if `condition` is `"or"` then `sel` is one of the present selectors of the given array

- **In case of failure (`err` is `"timeout"`)**
  - if `condition` is `"and"` then `sel` is one of the absent selectors of the given array
  - if `condition` is `"or"` then `sel` is *null* because no selectors were found

Example with selector argument as a string:

```
nick.open("https://phantombuster.com/cloud-services", function() {
    nick.waitUntilPresent('html', 2000, function(err) {
        if (err) {
            console.log(err);
            nick.exit(1);
        }
        console.log("'html' selector is present");
        nick.exit(0);
    });
});
```

This example succeeds if all selectors are present in the DOM:

```
nick.open("https://phantombuster.com/cloud-services", function() {
    nick.waitUntilPresent(['p', 'span', 'h2.title'], 2000, 'and',␣
↪function(err, selector) {
        if (err) {
            console.log(err);
            console.log("One of the missing selectors is:", selector);
            nick.exit(1);
        }
        console.log("'html', 'foo', 'bar' selectors are present");
        nick.exit(0);
    });
});
```

This example succeeds if one or more selector is present in the DOM:

```
nick.open("https://phantombuster.com/cloud-services", function() {
    nick.waitUntilPresent(['p', 'span', 'h2.title'], 2000, 'or',␣
↪function(err, selector) {
        if (err) {
            console.log(err);
            console.log("'p', 'span', 'h2.title' selectors are missing");
            nick.exit(1);
        }
        console.log("First matching selector:", selector);
        nick.exit(0);
    });
});
```

8.8 waitWhilePresent()

```
nick.waitWhilePresent(selectors, timeout [, condition = "and"], callback);
```

Waits while a DOM element, matching the provided selector, is present. If the method has to wait more than `timeout` milliseconds, `callback` is called with a `"timeout"` error. By default, `condition` is set to `"and"`.

It is considered good practice to always use a `wait*()` method after a page load and before any action on selectors.

This method is asynchronous and returns nothing. Use the `callback` to know when it has finished.

More info: http://docs.casperjs.org/en/latest/modules/casper.html#waitwhileselector

selectors (Array or String) An array of CSS3 selectors describing the path to DOM elements.

timeout (Number) Milliseconds to wait before calling `callback` function with an error.

condition (String) When `selectors` is an array, this argument lets you choose how to wait for the elements. If `condition` is `"and"`, the method will wait for the presence of all `selectors` in the DOM. Otherwise if `condition` is `"or"`, the method will wait until any `selector` of the array is present in the DOM.

callback (Function(String err, String sel)) Function called when finished. When there is no error, `err` is null.

- **In case of success (`err` is *null*):**
  - if `condition` is `"and"` then `sel` is *null* because all selectors are present
  - if `condition` is `"or"` then `sel` is one of the present selectors of the given array
- **In case of failure (`err` is `"timeout"`)**
  - if `condition` is `"and"` then `sel` is one of the absent selectors of the given array
  - if `condition` is `"or"` then `sel` is *null* because no selectors were found

Example with selector argument as a string:

```
nick.open("https://phantombuster.com/cloud-services", function() {
    nick.waitWhilePresent('html', 2000, function(err) {
        if (err) {
            console.log(err);
            nick.exit(1);
        }
        console.log("'html' selector is not present anymore");
        nick.exit(0);
    });
});
```

This example succeeds if all selectors is not present in the DOM:

```
nick.open("https://phantombuster.com/cloud-services", function() {
    nick.waitWhilePresent(['p', 'span', 'h2.title'], 2000, 'and',␣
↪function(err, selector) {
        if (err) {
            console.log(err);
            console.log("One of the missing selectors is:", selector);
            nick.exit(1);
        }
        console.log("'html', 'foo', 'bar' selectors are present");
        nick.exit(0);
    });
});
```

This example succeeds if one or more selector is not present in the DOM:

```
nick.open("https://phantombuster.com/cloud-services", function() {
    nick.waitWhilePresent(['p', 'span', 'h2.title'], 2000, 'or',␣
↪function(err, selector) {
        if (err) {
            console.log(err);
            console.log("'p', 'span', 'h2.title' selectors are missing");
            nick.exit(1);
        }
        console.log("First matching selector:", selector);
        nick.exit(0);
    });
});
```

8.9 waitUntilVisible()

```
nick.waitUntilVisible(selectors, timeout [, condition = "and"], callback);
```

Waits until a DOM element, matching the provided selector, is visible. If the method has to wait more than `timeout` milliseconds, `callback` is called with a `"timeout"` error. By default, `condition` is set to `"and"`.

It is considered good practice to always use a `wait*()` method after a page load and before any action on selectors.

This method is asynchronous and returns nothing. Use the `callback` to know when it has finished.

More info: http://docs.casperjs.org/en/latest/modules/casper.html#waituntilvisible

selectors (Array or String) An array of CSS3 selectors describing the path to DOM elements.

timeout (Number) Milliseconds to wait before calling `callback` function with an error.

condition (String) When `selectors` is an array, this argument lets you choose how to wait for the elements. If `condition` is `"and"`, the method will wait for the presence of all `selectors` in the DOM. Otherwise if `condition` is `"or"`, the method will wait until any `selector` of the array is present in the DOM.

callback (Function(String err, String sel)) Function called when finished. When there is no error, `err` is null.

- **In case of success (`err` is *null*):**
  - if `condition` is `"and"` then `sel` is *null* because all selectors are present
  - if `condition` is `"or"` then `sel` is one of the present selectors of the given array
- **In case of failure (`err` is `"timeout"`)**
  - if `condition` is `"and"` then `sel` is one of the absent selectors of the given array
  - if `condition` is `"or"` then `sel` is *null* because no selectors were found

Example with selector argument as a string:

```
nick.open("https://phantombuster.com/cloud-services", function() {
    nick.waitUntilVisible('html', 2000, function(err) {
        if (err) {
            console.log(err);
            nick.exit(1);
        }
        console.log("'html' selector is not present anymore");
        nick.exit(0);
    });
});
```

This example succeeds if all selectors is visible in the DOM:

```
nick.open("https://phantombuster.com/cloud-services", function() {
    nick.waitUntilVisible(['p', 'span', 'h2.title'], 2000, 'and',␣
↪function(err, selector) {
        if (err) {
            console.log(err);
            console.log("One of the missing selectors is:", selector);
            nick.exit(1);
        }
        console.log("'html', 'foo', 'bar' selectors are present");
        nick.exit(0);
    });
});
```

This example succeeds if one or more selector is visible in the DOM:

```
nick.open("https://phantombuster.com/cloud-services", function() {
    nick.waitUntilVisible(['p', 'span', 'h2.title'], 2000, 'or',␣
↪function(err, selector) {
        if (err) {
            console.log(err);
            console.log("'p', 'span', 'h2.title' selectors are missing");
            nick.exit(1);
        }
        console.log("First matching selector:", selector);
        nick.exit(0);
    });
});
```

8.10 waitWhileVisible()

```
nick.waitWhileVisible(selectors, timeout [, condition = "and"], callback);
```

Waits while a DOM element, matching the provided selector, is visible. If the method has to wait more than `timeout` milliseconds, `callback` is called with a `"timeout"` error. By default, `condition` is set to `"and"`.

It is considered good practice to always use a `wait*()` method after a page load and before any action on selectors.

This method is asynchronous and returns nothing. Use the `callback` to know when it has finished.

More info: http://docs.casperjs.org/en/latest/modules/casper.html#waitwhilevisible

selectors (Array or String) An array of CSS3 selectors describing the path to DOM elements.

timeout (Number) Milliseconds to wait before calling `callback` function with an error.

condition (String) When `selectors` is an array, this argument lets you choose how to wait for the elements. If `condition` is `"and"`, the method will wait for the presence of all `selectors` in the DOM. Otherwise if `condition` is `"or"`, the method will wait until any `selector` of the array is present in the DOM.

callback (Function(String err, String sel)) Function called when finished. When there is no error, `err` is null.

- **In case of success (`err` is *null*):**

  - if `condition` is `"and"` then `sel` is *null* because all selectors are present
  - if `condition` is `"or"` then `sel` is one of the present selectors of the given array
- **In case of failure (`err` is `"timeout"`)**
  - if `condition` is `"and"` then `sel` is one of the absent selectors of the given array
  - if `condition` is `"or"` then `sel` is *null* because no selectors were found

Example with selector argument as a string:

```
nick.open("https://phantombuster.com/cloud-services", function() {
    nick.waitWhileVisible('html', 2000, function(err) {
        if (err) {
            console.log(err);
            nick.exit(1);
        }
        console.log("'html' selector is not present anymore");
        nick.exit(0);
    });
});
```

This example succeeds if all selectors are not visible:

```
nick.open("https://phantombuster.com/cloud-services", function() {
    nick.waitWhileVisible(['p', 'span', 'h2.title'], 2000, 'and',␣
↪function(err, selector) {
        if (err) {
            console.log(err);
            console.log("One of the missing selectors is:", selector);
            nick.exit(1);
        }
        console.log("'html', 'foo', 'bar' selectors are present");
        nick.exit(0);
    });
});
```

This example succeeds if one or more selector is not visible:

```
nick.open("https://phantombuster.com/cloud-services", function() {
    nick.waitWhileVisible(['p', 'span', 'h2.title'], 2000, 'or',␣
↪function(err, selector) {
        if (err) {
            console.log(err);
            console.log("'p', 'span', 'h2.title' selectors are missing");
            nick.exit(1);
        }
        console.log("First matching selector:", selector);
        nick.exit(0);
    });
});
```

8.11 end()

```
Exit the process.
```

8.12 exit()

```
Exit the process.
```

8.13 evaluate()

```
nick.evaluate(sandboxedFunction [, argumentObject], callback);
```

Evaluates the function in the current page DOM context. The execution is sandboxed, the web page has no access to the Nick context. Data can be given through `argumentObject`.

This method is asynchronous and returns nothing. Use the `callback` to know when it has finished.

More info: http://docs.casperjs.org/en/latest/modules/casper.html#evaluate

sandboxedFunction (Function([Object argumentObject])) The function evaluated in the DOM context. `argumentObject` is a copy of the object given in the second optional argument.

argumentObject (PlainObject) Object to copy to the DOM context and given to the `sandboxedFunction` optional argument.

callback (Function(String err[, Object ret])) Function called when finished. When there is no error, `err` is null and `ret` is a copy of the object returned by sandboxedFunction call in DOM context.

Example:

```
var num = 21;

nick.evaluate(function(arg) {
    return arg.n * 2;
}, {
    'n': num
}, function(err, ret) {
    if (err) {
        console.log(err);
        nick.exit(1);
    }
    console.log("Evaluation succeeded. Return value is", ret); //
↪"Evaluation succeeded. Return value is 42"
    nick.exit(0);
});
```

8.14 evaluateAsync()

```
nick.evaluateAsync(sandboxedFunction [, argumentObject], callback);
```

Evaluates the function in the current page DOM context. The execution is sandboxed and asynchronous, the web page has no access to the Nick context. Data can be given through `argumentObject`. Because `sandboxedFunction` is asynchronous the function `done` must be called.

This method is asynchronous and returns nothing. Use the `callback` to know when it has finished.

More info: http://docs.casperjs.org/en/latest/modules/casper.html#evaluate

sandboxedFunction (Function([Object argumentObject], done)) The function evaluated in the DOM context. `argumentObject` is a copy of the object given in the second optional argument. `done` must be called before the function ends with the same arguments as `callback`.

argumentObject (PlainObject) Object to copy to the DOM context and given to the `sandboxedFunction` optional argument.

callback (Function(String err[, Object ret])) Function called when finished. When there is no error, `err` is null and `ret` is a copy of the object returned by sandboxedFunction call in DOM context.

Example:

```
var num = 21;

nick.evaluateAsync(function(arg, done) {
    return done(null, arg.n * 2;)
}, {
    'n': num
}, function(err, ret) {
    if (err) {
        console.log(err);
        nick.exit(1);
    }
    console.log("Evaluation succeeded. Return value is", ret); //
↪"Evaluation succeeded. Return value is 42"
    nick.exit(0);
});
```

8.15 inject()

```
nick.inject(url, callback);
```

Inject a script in the current DOM page context. The script can be hosted locally on the agent's disk or on a remote server.

This method is asynchronous and returns nothing. Use the `callback` to know when it has finished.

url (object) Path to a script hosted locally or remotely.

callback (Function(String err)) Function called when finished. When there is no error, `err` is null.

Example:

```
nick.inject("https://code.jquery.com/jquery-2.1.4.min.js", function(err) {
    if (err) {
        console.log(err);
        nick.exit(1);
    }
    console.log("Jquery script inserted!");
    nick.exit(0);
});
```

8.16 click()

```
nick.click(selector, callback);
```

Performs a click on the element matching the provided `selector` expression.

This method is asynchronous and returns nothing. Use the `callback` to know when it has finished.

More info: http://docs.casperjs.org/en/latest/modules/casper.html#click

selector (string) A CSS3 or XPath expression that describe the path to DOM elements.

callback (Function(String err)) Function called when finished. When there is no error, `err` is *null* and object is a valid object (which may be empty but never null).

Example:

```
var selector = "a.btn-warning";

nick.open("https://phantombuster.com/cloud-services", function() {
    nick.waitUntilVisible(selector, 2000, function(err) {
        if (err) {
            console.log(err)
            nick.exit(1);
        }
        nick.click(selector, function(err) {
            if (err) {
                console.log(err)
                nick.exit(1);
            }
            console.log("Click on 'TRY FREE' button done.");
            nick.exit(0);
        });
    });
});
```

8.17 getCurrentUrl()

```
nick.getCurrentUrl(callback)
```

Retrieves current page URL and calls the `callback` function with the URL in second argument. Note that the url will be url-decoded.

This method is asynchronous and returns nothing. Use the `callback` to know when it has finished.

More info: http://docs.casperjs.org/en/latest/modules/casper.html#getcurrenturl

callback (Function(String err, String decodedUrl)) Function called when finished. When there is no error, `err` is *null* and `decodedUrl` is a url-decoded string.

Example:

```
nick.open("https://phantombuster.com/cloud-services", function() {
    nick.getCurrentUrl(function(err, url) {
        if (err) {
            console.log(err);
            nick.exit(1);
        }
        console.log("Current Url: ", url);
        nick.exit(0);
    });
});
```

8.18 getCurrentUrlOrNull()

```
nick.getCurrentUrlOrNull()
```

This method is synchronous and returns *null* if it fails otherwise it returns a the current URL as a string. Note that the url will be url-decoded.

More info: http://docs.casperjs.org/en/latest/modules/casper.html#getcurrenturl

This function takes no arguments.

Example:

```
nick.open("https://phantombuster.com/cloud-services", function() {
    console.log(nick.getCurrentUrlOrNull());
    nick.exit();
});
```

8.19 getHtml()

```
nick.getHtml(callback)
```

Retrieves current page HTML and calls the `callback` function with the HTML as a string in second argument.

This method is asynchronous and returns nothing. Use the `callback` to know when it has finished.

More info: http://docs.casperjs.org/en/latest/modules/casper.html#gethtml

callback (Function(String err, String html)) Function called when finished. When there is no error, `err` is *null* and `html` is the HTML string.

Example:

```
nick.open("https://phantombuster.com/cloud-services", function() {
    nick.getHtml(function(err, html) {
        if (err) {
            console.log(err);
            nick.exit(1);
        }
        console.log("HTML: ", html);
        nick.exit(0);
    });
});
```

8.20 getHtmlOrNull()

```
nick.getHtmlOrNull()
```

This method is synchronous and returns *null* if it fails otherwise it returns the page HTML as a string.

More info: http://docs.casperjs.org/en/latest/modules/casper.html#gethtml

This function takes no arguments.

Example:

```
nick.open("https://phantombuster.com/cloud-services", function() {
    console.log(nick.getHtmlOrNull());
    nick.exit();
});
```

8.21 getContent()

```
nick.getContent(callback)
```

Retrieves current page content and call the `callback` function with the page content as a string in the second argument.

This method is asynchronous and returns nothing. Use the `callback` to know when it has finished.

More info: http://docs.casperjs.org/en/latest/modules/casper.html#getpagecontent

callback (Function(String err, String html)) Function called when finished. When there is no error, `err` is *null* and `html` is the HTML string.

Example:

```
nick.open("https://phantombuster.com/cloud-services", function() {
    nick.getPageContent(function(err, content) {
        if (err) {
            console.log(err);
            nick.exit(1);
        }
        console.log("Page content: ", content);
        nick.exit(0);
    });
});
```

8.22 getContentOrNull()

```
nick.getContentOrNull()
```

This method is synchronous and returns *null* if it fails otherwise it returns the page content (string).

More info: http://docs.casperjs.org/en/latest/modules/casper.html#getpagecontent

This function takes no arguments.

Example:

```
nick.open("https://phantombuster.com/cloud-services", function() {
    var content = nick.getPageContentOrNull();

    if (content == null) {
        console.log("content is null");
        nick.exit(1);
    }
    console.log("Content: ", content);
    nick.exit(0);
});
```

8.23 getTitle()

```
nick.getTitle(callback)
```

Retrieves current page title and call the `callback` function with the title in second argument.

This method is asynchronous and returns nothing. Use the `callback` to know when it has finished.

More info: http://docs.casperjs.org/en/latest/modules/casper.html#gettitle

callback (Function (String err, String title)) Function called when finished. When there is no error, `err` is *null* and `title` is the current page title string.

Example:

```
nick.open("https://phantombuster.com/cloud-services", function() {
    nick.getTitle(function(err, title) {
        if (err) {
            console.log(err);
            nick.exit(1);
        }
        console.log("Page title: ", title);
        nick.exit(0);
    });
});
```

8.24 getTitleOrNull()

```
nick.getTitleOrNull()
```

This method is synchronous and returns *null* if it fails otherwise it returns a the current page title string.

More info: http://docs.casperjs.org/en/latest/modules/casper.html#gettitle

This function takes no arguments.

Example:

```
nick.open("https://phantombuster.com/cloud-services", function() {
    var title = nick.getTitleOrNull();

    if (title == null) {
        console.log("title is null");
        nick.exit(1);
    }
    console.log("Title: ", title);
    nick.exit(0);
});
```

8.25 fill()

```
nick.fill(selector, inputs [, submit], callback);
```

Fills form inputs with the given values and optionally submits it. Inputs are referenced by their name attribute.

This method is asynchronous and returns nothing. Use the `callback` to know when it has finished.

More info: http://docs.casperjs.org/en/latest/modules/casper.html#gettitle

selector (String) A CSS3 or XPath expression that describe the path to DOM elements.

inputs (PlainObject) An object composed by name:value, with name, the input name and value, the value to set.

submit (Boolean) If `true` the form will be automatically sent.

callback (Function(String err)) Function called when finished. When there is no error, `err` is *null*.

Example with simple HTML form:

```
<form action="/contact" id="contact-form" enctype="multipart/form-data">
    <input type="text" name="subject"/>
    <textearea name="content"></textearea>
    <input type="radio" name="civility" value="Mr"/> Mr
    <input type="radio" name="civility" value="Mrs"/> Mrs
    <input type="text" name="name"/>
    <input type="email" name="email"/>
    <input type="file" name="attachment"/>
    <input type="checkbox" name="cc"/> Receive a copy
    <input type="submit"/>
</form>
```

A Nick script filling the form and sending it:

```
nick.open("https://example.com", function() {
    nick.fill('form#contact-form', {
        'subject': 'I am watching you',
        'content': 'So be careful.',
        'civility': 'Mr',
        'name': 'Chuck Norris',
        'email': 'chuck@norris.com',
        'cc': true,
        'attachment': '/Users/chuck/roundhousekick.doc'
    }, true, function(err) {
        if (err) {
            console.log(err);
            nick.exit(1);
        }
        console.log("Form sent!");
        nick.exit(0);
    });
});
```

8.26 screenshot()

```
nick.screenshot(filename [, clipRect, imgOptions], callback)
```

Take a screenshot of the current page. Without optional arguments, this method take a screenshot of the entire page.

This method is asynchronous and returns nothing. Use the `callback` to know when it has finished.

More info: http://docs.casperjs.org/en/latest/modules/casper.html#capture

path (String) The path of the screenshot. The format is defined by the file extention. 'image.jpg' will create a JPEG image in the current folder.

clipRect (PlainObject) This optional argument set the position and the size of the screenshot square.

Example:

```
clipRect = {
    top: 100,
    left: 100,
    width: 500,
    height: 400
}
```

imgOptions (PlainObject) This optional argument set the two avalaible image options. Such as the format and the quality of the screenshot image.

Example:

```
imgOptions = {
    format: 'jpg',
    quality: 50
}
```

callback (Function(String err)) Function called when finished. When there is no error, `err` is *null*.

Example:

```
nick.open("https://phantombuster.com/cloud-services", function() {
    nick.screenshot('./image.jpg', function(err) {
        if (err) {
            console.log(err);
            nick.exit(1);
        }
        console.log("Screenshot saved!")
        nick.exit(0);
    });
});
```

Example with options:

```
var buster = require('phantombuster').create()

nick.open("https://phantombuster.com/cloud-services", function() {
    nick.screenshot('./image.jpg'
    , {
        top: 90,
        left: 190,
        width: 900,
        height: 360
    }
    , {
        format: 'png',
        quality: 100
    }
    , function(err) {
        if (err) {
            console.log(err);
            nick.exit(1);
        }
        console.log("Screenshot saved!")
        buster.saveFolder(function(err) {
            if (err) {
                console.log(err);
                nick.exit(1);
            }
            nick.exit(0);
        });
    });
});
```

8.27 selectorScreenshot()

8.28 sendKeys()

```
nick.sendKeys(selector, keys [, options], callback)
```

Write keys in an `<input>`, `<textarea>` or any DOM element with `contenteditable="true"` in the current page.

This method is asynchronous and returns nothing. Use the `callback` to know when it has finished.

More info: http://docs.casperjs.org/en/latest/modules/casper.html#sendkeys

`selector (String)` A CSS3 or XPath expression that describes the path to DOM elements.

`keys (String)` Keys to send to the editable DOM element.

`options (PlainObject)`

The three options avalable are:

- `reset (Boolean)`: remove the content of the targeted element before sending key presses.
- `keepFocus (Boolean)`: keep the focus in the editable DOM element after keys have been sent.
- `modifiers (PlainObject)`: modifier string concatenated with a + (available modifiers are `ctrl`, `alt`, `shift`, `meta` and `keypad`).

`callback (Function(String err))` Function called when finished. When there is no error, `err` is *null*.

Example:

```
nick.open("https://phantombuster.com/cloud-services", function() {
    nick.sendKeys('#message', "Boo!", function(err) {
        if (err) {
            console.log(err);
            nick.exit(1);
        }
        console.log("Keys sent!")
        nick.exit(0);
    });
});
```

Example with optional argument:

```
nick.open("https://phantombuster.com/cloud-services", function() {
    nick.sendKeys('#message', "s", {
        reset: false,
        keepFocus: true,
        modifiers: "ctrl+alt+shift"
    }, function(err) {
        if (err) {
            console.log(err);
            nick.exit(1);
        }
        console.log("Keys sent!")
        nick.exit(0);
    });
});
```

CHAPTER 9

API

The Phantombuster API gives you control over your account. It is composed of HTTPS endpoints returning JSON data.

Here's a short list of what the API allows:

- Launch and abort agents
- Get console output, status, progress and messages from an agent
- Get real-time console output from an agent
- Get user, agent and script records
- Write scripts
- ...

We deliberately made the API extremely simple to use. Any developer should be able to get responses in a matter of minutes.

9.1 Versioning

All API endpoints URLs start with `https://phantombuster.com/api/v1/`.

Only version `1` exists for now.

9.2 Response format

All endpoints return JSON following the JSend specification. It basically means all successfull responses have HTTP code `2XX` and look like this:

```
{
    "status": "success",
    "data": {
        ...
    }
}
```

Date/time fields are Unix/POSIX timestamps (in seconds).

9.3 Authentication and request format

Authentication is dead simple: put your API key in the `X-Phantombuster-Key-1` HTTP header (or in the `key` parameter) of every request you make.

To get your API key, simply go to your settings page and click *Reveal*.

Parameters can be put in the query string or in the request body for `POST` requests. Here is how a typical request looks like:

```
GET /api/v1/agent/785/launch?command=casperjs&saveLaunchOptions=1 HTTP/1.1
Host: phantombuster.com
X-Phantombuster-Key-1: YOUR_API_KEY
```

You can also put your API key as a parameter. This is not recommended because your key might show up in log files:

```
GET /api/v1/agent/785/launch?command=casperjs&saveLaunchOptions=1&key=YOUR_API_KEY␣
↪HTTP/1.1
Host: phantombuster.com
```

Please be aware that your key is precious as anyone who knows it can launch your agents (and do other mean things). Do not hesitate to generate a new one if you think it has been compromised.

9.4 Errors

If something bad happens, the HTTP code will be `4XX` or `5XX` and the response will look like this:

```
{
    "status": "error",
    "message": "Description of what happened"
}
```

The error response might also contain a `code` field and a `data` field describing the error in more details.

Here are some error HTTP codes you might encounter:

- `400`: missing parameter or something else wrong with the given parameters
- `401`: missing API key or wrong API key
- `404`: the requested object was not found (bad ID?)
- `500`: for some reason our servers could not handle your request

9.5 Get an agent record

```
GET /api/v1/agent/{id}.json
```

Get an agent record.

{id} (Number) ID of the agent to retrieve.

withScript (String) If present and not empty, and if the agent has an associated script, also return the script record.

Sample response:

```
{
    "status": "success",
    "data": {
        "id": 1763,
        "name": "Nice Agent",
        "scriptId": 1902,
        "proxy": "none",
        "proxyAddress": null,
        "proxyUsername": null,
        "proxyPassword": null,
        "disableWebSecurity": false,
        "ignoreSslErrors": false,
        "loadImages": true,
        "launch": "manually",
        "nbLaunches": 94,
        "showDebug": true,
        "awsFolder": "nVFRid8kvsuPeuCL80DnBg",
        "executionTimeLimit": 5,
        "fileMgmt": "folders",
        "fileMgmtMaxFolders": 10,
        "lastEndMessage": "Execution time limit reached",
        "lastEndStatus": "error",
        "maxParallelExecs": 1,
        "userAwsFolder": "QwYH17CB0Xj",
        "nonce": 123,
        "script": {
            "id": 1902,
            "name": "nice_agent.coffee",
            "source": "phantombuster",
            "url": null,
            "text": " ... script contents ... ",
            "httpHeaders": null,
        }
    }
}
```

9.6 Launch an agent

```
POST /api/v1/agent/{id}/launch
```

Add an agent to the launch queue.

This endpoint supports three types of outputs:

- Standard JSON output (by setting `output` to `json`, which is the default) to get back a `containerId` in JSON.

  This ID can later be used to track this launch and get console output by calling `/api/v1/agent/{agentId}/output.json?containerId={containerId}`.

~ or ~

- Result object output (by setting `output` to `result-object`) to get a blocking JSON response which will close when your agent finishes.

  The response will contain your agent's exit code (`Number`) and its result object (`PlainObject`) if it was set (using *setResultObject()*). This endpoint is very useful for getting a response from your agents "synchronously" — just make a single HTTP request and wait for your result object/exit code.

  Use `first-result-object` instead to have the request terminate immediately after the first call to *setResultObject()*. This is the fastest way to get a response from an agent using the API. However you will only get the result object and nothing else (no exit code or console output for example).

  Use `result-object-with-output` instead to get the console output of your agent in addition to all the other fields.

  Obviously this endpoint can be very slow to terminate (if your agent takes a long time or is queued). To prevent any risk of timeout, a space character is sent every 10 seconds to keep the HTTP socket alive (spaces do not prevent JSON parsing).

  Note: The HTTP headers are sent before your agent finishes, so you'll get a `200 OK` even if your agent fails during execution (but not if it fails to queue).

~ or ~

- Event stream output (by setting `output` to `event-stream`) to get a `text/event-stream` HTTP response.

  Each line of console output is sent as an event stream message starting with `data:`. When you receive the first message, you know the agent has started. When the agent has finished, the connection is closed. At regular intervals, event stream comments (starting with `:`) are sent to keep the connection alive.

  See a demo of this endpoint in action.

~ or ~

- Raw output (by setting `output` to `raw`) to get an HTTP `text/plain`, chunked, streaming response of the raw console output of the agent.

  This is not recommended as almost all HTTP clients will timeout at one point or another, especially if your agent stays in queue for a few minutes (in which case the endpoint will send *zero* bytes for a few minutes, waiting for the agent to start — even cURL and Wget struggle to handle non-transmitting HTTP responses).

`{id}` (`Number`) ID of the agent to launch.

`output` (`String`) One of `json`, `result-object`, `first-result-object`, `result-object-with-output`, `event-stream` or `raw` (optional, default to `json`). This allows you to choose what type of response to receive.

`command` (`String`) Command to use when launching the agent (optional). Can be either `casperjs`, `phantomjs` or `node`.

argument (String) JSON argument as a `String` (optional). The argument can be retrieved with `buster.argument` in the agent's script.

saveLaunchOptions (String) If present and not empty, `command` and `argument` will be saved as the default launch options for the agent.

Note: `command` and `argument` work together. When setting one, always set the other. When one or both are set, the saved launch options of the agent are ignored.

Note: The `GET` HTTP method is also allowed for this endpoint.

Sample response of `json` output:

```
{
    "status": "success",
    "data": {
        "containerId": 76426
    }
}
```

Sample response of `result-object` output:

```
{
    "status": "success",
    "message": "Agent finished (success)",
    "data": {
        "containerId": 76426,
        "executionTime": 17,
        "exitCode": 0,
        "resultObject": {
            "your": "data",
            "is": {
                "here": [123]
            }
        }
    }
}
```

Sample response of `first-result-object` output:

```
{
    "status": "success",
    "data": {
        "containerId": 76426,
        "resultObject": {
            "your": "data",
            "is": {
                "here": [123]
            }
        }
    }
}
```

Sample response of `result-object-with-output` output:

```
{
    "status": "success",
    "message": "Agent finished (success)",
    "data": {
        "containerId": 76426,
        "executionTime": 17,
        "exitCode": 0,
        "resultObject": {
            "your": "data",
            "is": {
                "here": [123]
            }
        }
        "output": "This is a console output line!\r\nAnd this is another one :)\r\n"
    }
}
```

Sample response of `event-stream` output:

```
: container 76426 in queue

: container 76426 in queue

data: This a console output line!
data:

: container 76426 still running

data: And this is

data: another one :)
data:

: container 76426 ended
```

Sample response of `raw` output:

```
This is a console output line!
And this is another one :)
```

9.7 Abort an agent

```
POST /api/v1/agent/{id}/abort.json
```

Abort all running instances of the agent.

`{id}` (`Number`) ID of the agent to stop.

Note: The `GET` HTTP method is also allowed for this endpoint.

Sample response:

```
{
    "status": "success",
    "data": null
}
```

9.8 Get data from a running agent

```
GET /api/v1/agent/{id}/output.json
```

Get data from an agent: console output, status, progress and messages. This API endpoint is specifically designed so that it's easy to get incremental data from an agent.

This endpoint has two modes:

- "Track" mode (by setting `mode` to `track`, which is the default when a `containerId` is specified) to get console output from a particular instance of the agent.

    In this mode, requests must have the `containerId` parameter set to the instance's ID from which you wish to get console output.

~ or ~

- "Most Recent" mode (by setting `mode` to `most-recent`, which is the default when `containerId` is left at `0`) to get console output from the most recent instance of the agent.

    In this mode, your first call should have parameter `containerId` left at `0`. From then on, all subsequent calls must have parameter `containerId` set to the previously returned container ID (when a new instance of the agent is started, a different `containerId` will be returned).

`{id}` (`Number`) ID of the agent from which to retrieve the output, status and messages.

`mode` (`String`) Either `track` or `most-recent` (optional, defaults to `most-recent` if `containerId` is left at `0`, otherwise defaults to `track`). This controls from which instance of the agent the console output is returned. In "Most Recent" mode, the most recent instance is selected each time a request is made. In "Track" mode, the console output from a particular instance is returned, as specified by the `containerId` parameter.

`containerId` (`Number`) ID of the instance from which to get console output (optional, `0` by default). In "Most Recent" mode, always use the last `containerId` you received on a previous call or `0` for the first call. In "Track" mode, always set this parameter to the instance's ID from which you wish to get console ouput.

`fromMessageId` (`Number`) Return the agent's messages starting from this ID (optional, `-1` by default). If not present or `-1`, no messages are returned. Use the biggest message ID you received on a previous call to only get fresh messages.

`fromOutputPos` (`Number`) Return the agent's console output starting from this position (optional, `0` by default). This number corresponds to the number of bytes emitted by the agent. Use the last `outputPos` you received on a previous call to only get new output data.

Note: The `agentStatus` and `containerStatus` fields have 3 possible values: `running`, `queued` or `not running`.

Note: The `containerStatus` field is only present in "Track" mode and represents the status of the tracked agent instance.

Note: The `resultObject` field is only present when a result object was set using *buster.setResultObject()*.

Sample response:

```
{
    "status": "success",
    "data": {
        "agentStatus": "running",
        "containerStatus": "running",
        "runningContainers": 1,
        "queuedContainers": 0,
        "containerId": 76427,
        "progress": {
            "progress": 0.1,
            "label": "Initializing...",
            "runtime": 3
        },
        "messages": [
            {
                "id": 65444,
                "date": 1414080820,
                "text": "Agent started",
                "type": "normal",
                "context": [
                    "Launch type: manual",
                    "Execution time limit: 60s"
                ]
            }
        ],
        "output": "* Container a255b8220379 started in directory /home/phantom/agent",
        "outputPos": 245,
        "resultObject": {
            "your": "data",
            "is": {
                "here": [123]
            }
        }
    }
}
```

9.9 Get container records

```
GET /api/v1/agent/{id}/containers.json
```

Get a list of ended containers for an agent, ordered by date. Useful for listing the last available output logs from an agent.

Container history is saved for up to 7 days.

{id} (Number) ID of the agent from which to retrieve the containers.

Sample response:

```
{
    "status": "success",
    "data": [
        {
            "id": 195119,
            "queueDate": 1427810471,
            "launchDate": 1427810471,
            "launchType": "automatic",
            "launchNumber": 476,
            "endDate": 1427812088,
            "lastEndMessage": "Agent finished (error)",
            "lastEndStatus": "error",
            "exitCode": 1
        },
        {
            "id": 195050,
            "queueDate": 1427806874,
            "launchDate": 1427806874,
            "launchType": "automatic",
            "launchNumber": 475,
            "endDate": 1427810029,
            "lastEndMessage": "Agent finished (success)",
            "lastEndStatus": "success",
            "exitCode": 0
        }
    ]
}
```

9.10 Get a script by its ID

```
GET /api/v1/script/by-id/{mode}/{id}
```

Get a script record by its ID.

{id} (Number) ID of the script to retrieve.

{mode} (String) Either `json` or `raw`. If `raw` is used, the script is returned as raw text data, without any JSON.

withoutText (String) If present and not empty, do not send the script's contents but only its metadata (only in JSON mode).

Sample response:

```
{
    "status": "success",
    "data": {
        "id": 1902,
        "name": "nice_agent.coffee",
        "source": "phantombuster",
        "url": null,
        "text": " ... script contents ... ",
        "httpHeaders": null,
        "lastSaveDate": 1427806874,
        "nonce": 123
    }
}
```

9.11 Get a script by its name

```
GET /api/v1/script/by-name/{mode}/{name}
```

Get a script record by its name.

{name} (String) Name of the script to retrieve, with its extension (`.js` or `.coffee`).

{mode} (String) Either `json` or `raw`. If `raw` is used, the script is returned as raw text data, without any JSON.

withoutText (String) If present and not empty, do not send the script's contents but only its metadata (only in JSON mode).

Sample response:

```
{
    "status": "success",
    "data": {
        "id": 1902,
        "name": "nice_agent.coffee",
        "source": "phantombuster",
        "url": null,
        "text": " ... script contents ... ",
        "httpHeaders": null,
        "lastSaveDate": 1427806874,
        "nonce": 123
    }
}
```

9.12 List scripts

```
GET /api/v1/scripts.json
```

Get the list of all your scripts without text. To get a script contents, fetch it individually by its *ID* or *name*.

Sample response:

```
{
    "status": "success",
    "data": [
        {
            "id": 450,
            "name": "script1.coffee",
            "source": "phantombuster",
            "url": "",
            "httpHeaders": null,
            "lastSaveDate": 1446562593,
            "nonce": 12
        },
        {
            "id": 452,
            "name": "script2.js",
            "source": "sdk",
            "url": "",
            "httpHeaders": null,
            "lastSaveDate": 1446562789,
            "nonce": 4
        }
    ]
}
```

9.13 Delete a script

```
DELETE /api/v1/script/{id}.json
```

Delete one of your script.

{id} (Number) ID of the script to delete.

Sample response:

```
{
    "status": "success",
    "data": null
}
```

9.14 Update or create a script

```
POST /api/v1/script/{name}
```

Update an existing script or create a new one if it does not exist (in this case, the new script ID is returned in the `data` field).

{name} (String) Name of the script to update or create, with its extension (`.js` or `.coffee`).

text (String) Full text contents of the script. This parameter must be in the request body in `x-www-form-urlencoded` format.

insertOnly (String) If present and not empty, make sure that we don't update an existing script (optional). An error will be returned if a script with the same name already exists.

source (String) Optional `String` describing from where the script comes from. Reserved sources keywords are `phantombuster`, `web`, `sdk` and `bot builder`. Only 20 alpha-numeric characters (and space) are allowed.

Sample response:

```
{
    "status": "success",
    "data": 345
}
```

9.15 List running agents and account info

```
GET /api/v1/user.json
```

Get information about your Phantombuster account and your agents.

Sample response:

```
{
    "status": "success",
    "data": {
        "email": "excellent.customer@gmail.com",
        "timeLeft": 14087,
        "emailsLeft": 100,
        "captchasLeft": 10,
        "storageLeft": 9991347906,
        "databaseLeft": 239222784,
        "agents": [
            {
                "id": 1388,
                "name": "My first agent",
                "scriptId": 0,
                "lastEndMessage": "Agent has no associated script",
                "lastEndStatus": "launch failed",
                "queuedContainers": 2,
                "runningContainers": 0
            },
            {
                "id": 1713,
                "name": "My second agent",
                "scriptId": 2003,
                "lastEndMessage": "Agent finished with exit code 0",
                "lastEndStatus": "success",
                "queuedContainers": 0,
                "runningContainers": 1,
                "progress": {
                    "progress": 0.544,
                    "label": "A progress label",
                    "runtime": 477
                }
            }
        ]
    }
}
```

CHAPTER 10

SDK

All your scripts can easily be written right on our website, in the provided CoffeeScript/JavaScript web editor.

However, you might prefer using your own editor, locally on your machine. We made Phantombuster's SDK specifically for this.

The SDK will **monitor a directory** on your disk for changes in your scripts. As soon as a change is detected, the script will be uploaded in your Phantombuster account.

10.1 Installation

Because we made the SDK as a Node module, you need to have `npm` (Node Package Manager) installed on your machine. Then type:

```
# npm install -g phantombuster-sdk
```

This will globally install the `phantombuster` command.

10.2 Project directory

All your scripts will go in a project directory. You can arrange the contents of this directory however you like (by creating a sub-directory for each category of scripts for example).

You can choose any name for this directory. For example:

```
$ mkdir phantom-project
```

10.3 Configuration

All the SDK's configuration goes into a single file named `phantombuster.cson` at the root of your project directory. It's written in CoffeeScript Object Notation (`.cson`).

Open/create this file with the editor of your choice:

```
$ vi phantom-project/phantombuster.cson
```

Configuration is very simple:

```
[
    name: 'account name'
    apiKey: 'YOUR_API_KEY_HERE'
    scripts:
        'my-script.js': 'project/test/my-script.js'
        'another-script.coffee': 'folder/script.coffee'
]
```

- `name` is an arbitrary `String` of your choice. People generally put in the name of their Phantombuster account.

  This name is shown each time a script is successfully saved on Phantombuster's servers.

- `apiKey` must contain the API key of your Phantombuster account as a `String`.

  You can find it in your settings page (click *Reveal*). It's important to keep this key secret because it allows anyone in its possession to manipulate your Phantombuster account (`phantombuster.cson` is a private file and should not be shared). If you think your API key has been compromised, do not hesitate to generate a new one.

- `scripts` is a `PlainObject` containing a list of scripts that you wish to manage with the SDK.

  Each line (key/value pair) represents a script. The key (left side) is the script's name on Phantombuster (therefore it must not contain any special characters or any slashes). The value (right side) is the location of this script's file on your machine's disk, in your project directory (in fact, relative to the location of `phantombuster.cson`).

10.4 Launch

To start the SDK, simply navigate to your project directory and execute the `phantombuster` command.

```
$ cd phantom-project
$ phantombuster
```

This will start a monitoring process. If any modification happens to your script's files, they will automatically be saved in your Phantombuster account.

It means you can now start the editor of your choice and start developping your bots in an environment you like.

10.5 Advanced usage

```
phantombuster [-c config.cson] [script1.coffee [script2.coffee...]]
```

You can specify another configuration file with the `-c` option.

You can also provide one or more paths to scripts to upload them to your Phantombuster account (without starting the monitoring process).