selenium-docker Documentation

Release 0.5.0

Blake VandeMerwe

Jan 18, 2018

Contents

Selenium-Docker is a glue library that combines the functionality of Docker and Selenium using gevent. Its primary aim is to be a reliable drop-in replacement for traditional Selenium testing by hiding the browser windows and enabling advanced features such as screen recording and automatic proxy caching.

Apache 2.0 licensed source code is available in the `LICENSE` file.

Code is hosted on Github, vivint/selenium-docker.

CHAPTER 1

Contents

1.1 Installation and Dependencies

1.1.1 The Package

Selenium-docker and its required modules can be found on pypi and intalled via `pip`:

```
pip install selenium-docker
```

For the most up to date release you can install from source:

```
pip install git+git//<URL>
```

If the required dependancies install correctly the next step is to ensure Docker is properly installed and configured. By default `selenium-docker` will attempt to connect to a Docker Engine running on the local machine. Alternatively a remote docker engine can be used instead with additional setup and overhead.

1.1.2 Installing Docker

The easiest way to install Docker is by following the instructions for your preferred operating system and distribution.

Following the official downloads and instructions will yield the best results.

1.1.3 Docker Images

Terminal:

```
docker pull vivint/selenium-chrome-ffmpeg
docker pull vivint/selenium-firefox-ffmpeg
```

Links to the Dockerfiles can be found on Github.

1.2 API

1.2.1 Base

class selenium_docker.base.**ContainerFactory**(*engine*, *namespace*, *make_default=True*, *logger=None*)

Used as an interface for interacting with Container instances.

Example:

```
from selenium_docker.base import ContainerFactory

factory = ContainerFactory.get_default_factory('reusable')
factory.stop_all_containers()
```

Will attempt to connect to the local Docker Engine, including the word `reusable` as part of each new container's name. Calling `factory.stop_all_containers()` will stop and remove containers assocated with that namespace.

Reusing the same `namespace` value will allow the factory to inherit the correct containers from Docker when the program is reset.

Parameters

- **engine** (`docker.client.DockerClient`) – connection to the Docker Engine the application will interact with. If `engine` is `None` then `docker.client.from_env()` will be called to attempt connecting locally.
- **namespace** (*str*) – common name included in all the new docker containers to allow tracking their status and cleaning up reliably.
- **make_default** (*bool*) – when `True` this instance will become the default, used as a singleton, when requested via *get_default_factory()*.
- **logger** (`logging.Logger`) – logging module Logger instance.

DEFAULT = None

ContainerFactory – singleton instance to a container factory that can be used to spawn new containers accross a single connected Docker engine.

This is the instance returned by *get_default_factory()*.

_ContainerFactory__bootstrap(*container*, ***kwargs*)

Adds additional attributes and functions to Container instance.

Parameters

- **container** (*Container*) – instance of `Container` that is being fixed up with expected values.
- **kwargs** (*dict*) – arbitrary attribute names and their values to attach to the `container` instance.

Returns the exact instance passed in.

Return type `Container`

as_json()

JSON representation of our factory metadata.

Returns that is a `json.dumps()` compatible dictionary instance.

Return type dict

containers
dict – `Container` instances mapped by name.

docker
`docker.client.DockerClient` – reference to the connected Docker engine.

gen_name(*key=None*)
Generate the name of a new container we want to run.

This method is used to keep names consistent as well as to ensure the name/identity of the `ContainerFactory` is included. When a `ContainerFactory` is loaded on a machine with containers already running with its name it'll inherit those instances to re-manage between application runs.

Parameters **key** (*str*) – the identifiable portion of a container name. If one isn't supplied (the default) then one is randomly generated.

Returns in the format of `selenium-<FACTORY_NAMESPACE>-<KEY>`.

Return type str

classmethod get_default_factory(*namespace=None*, *logger=None*)
Creates a default connection to the local Docker engine.

This `classmethod` acts as a singleton. If one hasn't been made it will attempt to create it and attach the instance to the class definition. Because of this the method is the preferable way to obtain the default connection so it doesn't get overwritten or modified by accident.

Note: By default this method will attempt to connect to the **local** Docker engine only. Do not use this when attempting to use a remote engine on a different machine.

Parameters

- **namespace** (*str*) – use this namespace if we're creating a new default factory instance.
- **logger** (`logging.Logger`) – instance of logger to attach to this factory instance.

Returns instance to interact with Docker engine.

Return type *ContainerFactory*

get_namespace_containers(**args*, ***kwargs*)
Glean the running containers from the environment that are using our factory's namespace.

Parameters **namespace** (*str*) – word identifying ContainerFactory containers represented in the Docker Engine.

Returns `Container` instances mapped by name.

Return type dict

load_image(**args*, ***kwargs*)
Issue a `docker pull` command before attempting to start/run containers. This could potentially increase startup time, as well as ensure the containers are up-to-date.

Parameters

- **image** (*str*) – name of the container we're downloading.
- **tag** (*str*) – tag/version of the container.
- **insecure_registry** (*bool*) – allow downloading image templates from insecure Docker registries.

- **background** (*bool*) – spawn the download in a background thread.

Raises docker.errors.DockerException – if anything goes wrong during the image template download.

Returns the Image controlled by the connected Docker engine. Containers are spawned based off this template.

Return type docker.models.images.Image

namespace
str – ready-only property for this instance's namespace, used for generating names.

scrub_containers(**args*, ***kwargs*)
Remove **all** containers that were dynamically created.

Parameters **labels** (*str*) – labels to include in our search for finding containers to scrub from the connected Docker engine.

Returns the number of containers stopped and removed.

Return type int

start_container(**args*, ***kwargs*)
Creates and runs a new container defined by spec.

Parameters

- **spec** (*dict*) – the specification of our docker container. This can include things such as the name, labels, image, restart conditions, etc. The built-in driver containers already have this defined in their class declaration.
- **kwargs** (*[str, str]*) – additional arguments that will be added to spec; generally dynamic attributes modifying a static container definition.

Raises docker.errors.DockerException – when there's any problem performing start and run on the container we're attemping to create.

Returns the newly created and managed container instance.

Return type docker.models.containers.Container

stop_all_containers(**args*, ***kwargs*)
Remove all containers from this namespace.

Raises

- APIError – when there's a problem communicating with the Docker Engine.
- NotFound – when a tracked container cannot be found in the Docker Engine.

Returns None

stop_container(**args*, ***kwargs*)
Remove an individual container by name or key.

Parameters

- **name** (*str*) – name of the container.
- **key** (*str*) – partial reference to the container. (Optional)
- **timeout** (*int*) – time in seconds to wait before sending SIGKILL to a running container.

Raises

- `ValueError` – when `key` and `name` are both `None`.
- `APIError` – when there's a problem communicating with Docker engine.
- `NotFound` – when no such container by `name` exists.

Returns None

class selenium_docker.base.**ContainerInterface**
Required functionality for implementing a custom object that has an underlying container.

selenium_docker.base.**check_engine**(*fn*)
Pre-check our engine connection by sending a ping before our intended operation.

Parameters **fn** (*Callable*) – wrapped function.

Returns Callable

Example:

```
@check_engine
def do_something_with_docker(self):
    # will raise APIError before getting here
    #  if there's a problem with the Docker Engine connection.
    return True
```

1.2.2 Drivers

Base

class selenium_docker.drivers.**DockerDriverBase**(*user_agent=None*, *proxy=None*, *cargs=None*, *ckwargs=None*, *extensions=None*, *logger=None*, *factory=None*, *flags=None*)
Base class for all drivers that want to implement Webdriver functionality that maps to a running Docker container.

BASE_URL = 'http://{host}:{port}/wd/hub'
str – connection URL used to bind with docker container.

BROWSER = 'Default'
str – name of the underlying browser being used. Classes that inhert from *DockerDriverBase* should overwrite this attribute.

CONTAINER = None
dict – default specification for the underlying container. This definition is passed to the Docker Engine and is responsible for defining resources and metadata.

DEFAULT_ARGUMENTS = None
list – default arguments to apply to the WebDriver binary inside the Docker container at startup. This can be used for changing the user agent or turning off advanced features.

class Flags(**args*, ***kwds*)
Default bit flags to enable or disable all extra features.

_generate_next_value_(*name*, *start*, *count*, *last_values*)
Generate the next value when not given.

name: the name of the member start: the initital start value or None count: the number of existing members last_value: the last value assigned or None

IMPLICIT_WAIT_SECONDS = 10.0
float – this can only be called once per WebDriver instance. The value here is applied at the end of `__init__` to prevent the WebDriver instance from hanging inside the container.

SELENIUM_PORT = '4444/tcp'
str – identifier for extracting the host port that's bound to Docker's internal port for the underlying container. This string is in the format `PORT/PROTOCOL`.

__init__(*user_agent=None*, *proxy=None*, *cargs=None*, *ckwargs=None*, *extensions=None*, *logger=None*, *factory=None*, *flags=None*)
Selenium compatible Remote Driver instance.

Parameters

- **user_agent** (*str or Callable*) – overwrite browser's default user agent. If `user_agent` is a Callable then the result will be used as the user agent string for this browser instance.
- **proxy** (*Proxy or SquidProxy*) – Proxy (or SquidProxy) instance that routes container traffic.
- **cargs** (*list*) – container creation arguments.
- **ckwargs** (*dict*) – container creation keyword arguments.
- **extensions** (*list*) – list of file locations loaded as browser extensions.
- **logger** (Logger) – logging module Logger instance.
- **factory** (*ContainerFactory*) – abstract connection to a Docker Engine that does the primary interaction with starting and stopping containers.
- **flags** (`aenum.Flag`) – bit flags used to turn advanced features on or off.

Raises

- `ValueError` – when `proxy` is an unknown/invalid value.
- `Exception` – when any problem occurs connecting the driver to its underlying container.

_make_container(**args*, ***kwargs*)
Create a running container on the given Docker engine.

This container will contain the Selenium runtime, and ideally a browser instance to connect with.

Parameters ****kwargs** (*dict*) – the specification of the docker container.

Returns Container

_perform_check_container_ready()
Checks if the container is ready to use by calling a separate function. This function `check_container_ready` must manage its own retry logic if the check is to be performed more than once or over a span of time.

Raises `DockerException` – when the container's creation and state cannot be verified.

Returns `True` when `check_container_ready()` returns `True`.

Return type bool

base_url
str – read-only property of Selenium's base url.

check_container_ready(**args*, ***kw*)
Function that continuously checks if a container is ready.

Note: This function should be wrapped in a *tenacity.retry* for continuously checking the status without failing.

Raises `requests.RequestException` – for any *requests* related exception.

Returns `True` when the status is good. `False` if it cannot be verified or is in an unusable state.

Return type bool

close_container()
Removes the running container from the connected engine via `DockerDriverBase.factory`.

Returns None

docker
`docker.client.DockerClient` – reference

f(*flag*)
Helper function for checking if we included a flag.

Parameters **flag** (`aenum.Flag`) – instance of `Flag`.

Returns logical AND on an individual flag and a bit-flag set.

Return type bool

Example:

```
from selenium_docker.drivers.chrome import ChromeDriver, Flags

driver = ChromeDriver(flags=Flags.ALL)
driver.get('https://python.org')

if driver.f(Flags.X_IMG):  # no images allowed
    # do something
    pass

driver.quit()
```

get_url()
Extract the hostname and port from a running docker container, return it as a URL-string we can connect to.

References

selenium_docker.utils.ip_port()

Returns str

identity
str – reference to the parent class' name.

name
str – read-only property of the container's name.

quit()
Alias for *DockerDriverBase.close_container()*.

Generally this is called in a Selenium tests when you want to completely close and quit the active browser.

Returns None

selenium_docker.drivers.**check_container**(*fn*)

Ensure we're not trying to double up an external container with a Python instance that already has one. This would create dangling containers that may not get stopped programmatically.

Note: This method is placed under `base` to prevent circular imports.

Parameters **fn** (*Callable*) – wrapped function.

Returns Callable

Video Base

class selenium_docker.drivers.**VideoDriver**(*path='/tmp'*, **args*, ***kwargs*)

Chrome browser inside Docker with video recording of its lifetime.

Parameters **path** (*str*) – directory where finished video recording should be stored.

save_path
str – directory to save video recording.

_time
int – time stamp of when the class was instatiated.

__is_recording
bool – flag for internal recording state.

__recording_path
str – Docker internal path for saved files.

commands
`dotmap.DotMap` – aliases for commands that run inside the docker container for starting and stopping ffmpeg.

start_ffmpeg
using X11 and LibX264.

stop_ffmpeg
killing the process will correctly stop video recording.

filename
str – filename to apply to the extracted video stream.

The filename will be formatted, `<BROWSER>-docker-<TIMESTAMP>.mkv`.

is_recording
bool – the container is recording video right now.

quit()
Stop video recording before closing the driver instance and removing the Docker container.

Returns None

start_recording(**args*, ***kwargs*)
Starts the ffmpeg video recording inside the container.

Parameters

- **metadata** (*dict*) – arbitrary data to attach to the video file.

- **environment** (*dict*) – environment variables to inject inside the running container before launching ffmpeg.

Returns the absolute file path of the file being recorded, inside the Docker container.

Return type str

stop_recording(**args*, ***kwargs*)
Stops the ffmpeg video recording inside the container.

Parameters

- **path** (*str*) – local directory where the video file should be stored.
- **shard_by_date** (*bool*) – when `True` video files will be placed in a folder structure under `path` in the format of `YYYY/MM/DD/<files>`.
- **environment** (*dict*) – environment variables to inject inside the container before executing the commands to stop recording.

Raises

- `ValueError` – when `path` is not an existing folder path.
- `IOError` – when there's a problem creating the folder for video recorded files.

Returns file path to completed recording. This value is adjusted for `shard_by_date`.

Return type str

Proxy

class selenium_docker.proxy.**SquidProxy**(*logger=None*, *factory=None*)

CONTAINER = {'publish_all_ports': True, 'detach': True, 'image': 'minimum2scp/squid
dict – default specification for the underlying container.

SQUID_PORT = '3128/tcp'
str – identifier for extracting the host port that's bound to Docker.

__init__(*logger=None*, *factory=None*)

Parameters

- **logger** –
- **factory** (*selenium_docker.base.ContainerFactory*) –

_make_container(**args*, ***kwargs*)
Create a running container on the given Docker engine.

Returns `Container`

close_container()
Removes the running container from the connected engine via `DockerDriverBase.factory`.

Returns None

name
str – read-only property of the container's name.

quit()
Alias for *close_container()*.

Returns None

Chrome

class selenium_docker.drivers.chrome.**ChromeDriver**(*user_agent=None*, *proxy=None*, *cargs=None*, *ckwargs=None*, *extensions=None*, *logger=None*, *factory=None*, *flags=None*)

Chrome browser inside Docker.

Inherits from *DockerDriverBase*.

_capabilities(*arguments*, *extensions*, *proxy*, *user_agent*)
Compile the capabilities of ChromeDriver inside the Container.

Parameters

- **arguments** (*list*) –
- **extensions** (*list*) –
- **proxy** (*Proxy*) –
- **user_agent** (*str*) –

Returns dict

_final(*arguments*, *extensions*, *proxy*, *user_agent*)
Configuration applied after the driver has been created.

Parameters

- **arguments** (*list*) – unused.
- **extensions** (*list*) – unused.
- **proxy** (*Proxy*) – adds proxy instance to DesiredCapabilities.
- **user_agent** (*str*) – unused.

Returns None

_profile(*arguments*, *extensions*, *proxy*, *user_agent*)
No-op for ChromeDriver.

class selenium_docker.drivers.chrome.**ChromeVideoDriver**(*path='/tmp'*, **args*, ***kwargs*)

Chrome browser inside Docker with video recording.

Inherits from *VideoDriver*.

Firefox

class selenium_docker.drivers.firefox.**FirefoxDriver**(*user_agent=None*, *proxy=None*, *cargs=None*, *ckwargs=None*, *extensions=None*, *logger=None*, *factory=None*, *flags=None*)

Firefox browser inside Docker.

Inherits from *DockerDriverBase*.

_capabilities(*arguments*, *extensions*, *proxy*, *user_agent*)
Compile the capabilities of FirefoxDriver inside the Container.

Parameters

- **arguments** (*list*) – unused.

- **extensions** (*list*) – unused.
- **proxy** (*Proxy*) – adds proxy instance to DesiredCapabilities.
- **user_agent** (*str*) – unused.

Returns dict

_final(*arguments*, *extensions*, *proxy*, *user_agent*)
Configuration applied after the driver has been created.

Parameters

- **arguments** (*list*) – unused.
- **extensions** (*list*) – unused.
- **proxy** (*Proxy*) – adds proxy instance to DesiredCapabilities.
- **user_agent** (*str*) – unused.

Returns None

_profile(*arguments*, *extensions*, *proxy*, *user_agent*)
Compile the capabilities of ChromeDriver inside the Container.

Parameters

- **arguments** (*list*) –
- **extensions** (*list*) –
- **proxy** (*Proxy*) – unused.
- **user_agent** (*str*) –

Returns FirefoxProfile

class selenium_docker.drivers.firefox.**FirefoxVideoDriver**(*path='/tmp'*, **args*, ***kwargs*)
Firefox browser inside Docker with video recording.

Inherits from *VideoDriver*.

1.2.3 Driver Pools

class selenium_docker.pool.**DriverPool**(*size*, *driver_cls=<class 'selenium_docker.drivers.chrome.ChromeDriver'>*, *driver_cls_args=None*, *driver_cls_kw=None*, *use_proxy=True*, *factory=None*, *name=None*, *logger=None*)
Create a pool of available Selenium containers for processing.

Parameters

- **size** (*int*) – maximum concurrent tasks. Must be at least 2.
- **driver_cls** (*WebDriver*) –
- **driver_cls_args** (*tuple*) –
- **driver_cls_kw** (*dict*) –
- **use_proxy** (*bool*) –
- **factory** (*ContainerFactory*) –

- **name** (*str*) –
- **logger** (logging.Logger) –

Example:

```
pool = DriverPool(size=2)

urls = [
    'https://google.com',
    'https://reddit.com',
    'https://yahoo.com',
    'http://ksl.com',
    'http://cnn.com'
]

def get_title(driver, url):
    driver.get(url)
    return driver.title

for result in pool.execute(get_title, urls):
    print(result)
```

INNER_THREAD_SLEEP = 0.5
float – essentially our polling interval between tasks and checking when tasks have completed.

PROXY_CLS
AbstractProxy: created for the pool when use_proxy=True during pool instantiation.

alias of SquidProxy

_DriverPool__bootstrap()
Prepare this driver pool instance to batch execute task items.

_DriverPool__cleanup(*force=False*)
Stop and remove the web drivers and their containers. This function should not remove pending tasks or results. It should be possible to cleanup all the external resources of a driver pool and still extract the results of the work that was completed.

Raises

- *DriverPoolRuntimeException* – when attempting to cleanup an environment while processing is still happening, and forcing the cleanup is set to False.
- SeleniumDockerException – when a driver instance or container cannot be closed properly.

Returns None

_load_driver(*and_add=True*)
Load a single web driver instance and container.

_load_drivers()
Load the web driver instances and containers.

Raises *DriverPoolRuntimeException* – when the requested number of drivers for the given pool size cannot be created for some reason.

Returns None

add_async(**items*)
Add additional items to the asynchronous processing queue.

Parameters **items** (*list(Any)*) – list of items that need processing. Each item is applied one at a time to an available driver from the pool.

Raises StopIteration – when all items have been added.

close()
Force close all the drivers and cleanup their containers.

Returns None

execute(*fn*, *items*, *preserve_order=False*, *auto_clean=True*, *no_wait=False*)
Execute a fixed function, blocking for results.

Parameters

- **fn** (*Callable*) – function that takes two parameters, driver and task.
- **items** (*list(Any)*) – list of items that need processing. Each item is applied one at a time to an available driver from the pool.
- **preserve_order** (*bool*) – should the results be returned in the order they were supplied via items. It's more performant to allow results to return in any order.
- **auto_clean** (*bool*) – cleanup docker containers after executing. If multiple processing tasks are going to be used, it's more performant to leave the containers running and reuse them.
- **no_wait** (*bool*) – forgo a small sleep interval between finishing a task and putting the driver back in the available drivers pool.

Yields *results* – the result for each item as they're finished.

execute_async(*fn*, *items=None*, *callback=None*, *catch=(<class 'selenium.common.exceptions.WebDriverException'>,)*, *requeue_task=False*)
Execute a fixed function in the background, streaming results.

Parameters

- **fn** (*Callable*) – function that takes two parameters, driver and task.
- **items** (*list(Any)*) – list of items that need processing. Each item is applied one at a time to an available driver from the pool.
- **callback** (*Callable*) – function that takes a single parameter, the return value of fn when its finished processing and has returned the driver to the queue.
- **catch** (*tuple[Exception]*) – tuple of Exception classes to catch during task execution. If one of these Exception classes is caught during fn execution the driver that crashed will attempt to be recycled.
- **requeue_task** (*bool*) – in the event of an Exception being caught should the task/item that was being worked on be re-added to the queue of items being processed.

Raises *DriverPoolValueError* – if callback is not None or callable.

Returns None

is_async
bool – returns True when asynchronous processing is happening.

is_processing
bool – whether or not we're currently processing tasks.

quit()
Alias for *close()*. Included for consistency with driver instances that generally call quit when they're no longer needed.

Returns None

results(*block=True*)
Iterate over available results from processed tasks.

Parameters **block** (*bool*) – when `True`, block this call until all tasks have been processed and all results have been returned. Otherwise this will continue indefinitely while tasks are dynamically added to the async processing queue.

Yields *results* – one result at a time as they're finished.

Raises `StopIteration` – when the processing is finished.

stop_async(*timeout=None*, *auto_clean=True*)
Stop all the async worker processing from executing.

Parameters

- **timeout** (*float*) – number of seconds to wait for pool to finish processing before killing and closing out the execution.
- **auto_clean** (*bool*) – cleanup docker containers after executing. If multiple processing tasks are going to be used, it's more performant to leave the containers running and reuse them.

Returns None

exception selenium_docker.pool.**DriverPoolRuntimeException**
Pool RunTime Exception.

exception selenium_docker.pool.**DriverPoolValueError**
Pool interaction ValueError.

1.2.4 Helpers

JsonFlags(*args, **kwds)	`aenum.Flag` mixin to return members as JSON dict.
OperationsMixin	Optional mixin object to extend default driver functionality.

selenium_docker.helpers.**HTML_TAG = ('tag name', 'html')**
(str, str) – tuple representing an <HTML> tag.

class selenium_docker.helpers.**JsonFlags**(**args*, ***kwds*)
`aenum.Flag` mixin to return members as JSON dict.

_generate_next_value_(*name*, *start*, *count*, *last_values*)
Generate the next value when not given.

name: the name of the member start: the initital start value or None count: the number of existing members last_value: the last value assigned or None

classmethod as_json()
Converts the Flag enumeration to a JSON structure.

Returns Flag names and their corresponding integer-bit-value.

Return type dict(str, int)

classmethod from_values(**values*)
Creates a compound Flag instance.

Logically OR's the integer/string `values` and returns a bit-flag that represents the features we want enabled in our Driver instance.

Parameters **values** (*int or str*) – the integer-bit value or the flag name.

Returns Compound Flag instance with the features we requested.

Return type aenum.Flag

class selenium_docker.helpers.**OperationsMixin**
Optional mixin object to extend default driver functionality.

switch_to_frame(*selector*, *wait_for=('tag name', 'html')*, *max_time=30*)
Wait for a frame to load then switch to it.

Note: Because there are two waits being performed in this operation the max_wait time could be doubled at most the value applied.

Parameters

- **selector** (*tuple*) – iFrame we're looking for, in the form of (By, str).
- **wait_for** (*tuple*) – element to wait for inside the iFrame, in the form of (By, str).
- **max_time** (*int*) – time in seconds to wait for each element.

Raises Exception; when anything goes wrong.

Returns when there were no exceptions and the operation completed successfully.

Return type WebElement

1.2.5 Utils

gen_uuid([length])	Generate a random ID.
in_container()	Determines if we're running in an lxc/docker container.
ip_port(container, port)	Returns an updated HostIp and HostPort from the container's network properties.
load_docker_image(_docker, image[, tag, ...])	Issue a *docker pull* command before attempting to start/run containers.
parse_metadata(meta)	Convert a dictionary into proper formatting for ffmpeg.

selenium_docker.utils.**gen_uuid**(*length=4*)
Generate a random ID.

Parameters **length** (*int*) – length of generated ID.

Returns of length length.

Return type str

selenium_docker.utils.**in_container**()
Determines if we're running in an lxc/docker container.

Checks in various locations with different methods. If any one of these default operations are successful the function returns True. This is not an infallible method and can be faked easy.

Returns bool

selenium_docker.utils.**ip_port**(*container*, *port*)
Returns an updated HostIp and HostPort from the container's network properties. Calls container reload on-call.

Parameters

- **container** (*Container*) –
- **port** (*str*) –

Returns IP/hostname and port.

Return type tuple(str, int)

selenium_docker.utils.**load_docker_image**(*_docker*, *image*, *tag=None*, *insecure_registry=False*, *background=False*)

Issue a *docker pull* command before attempting to start/run containers. This could potentially alliviate startup time, as well as ensure the containers are up-to-date.

Parameters

- **_docker** (*DockerClient*) –
- **image** (*str*) –
- **tag** (*str*) –
- **insecure_registry** (*bool*) –
- **background** (*bool*) –

Returns Image

selenium_docker.utils.**parse_metadata**(*meta*)

Convert a dictionary into proper formatting for ffmpeg.

Parameters **meta** (*dict*) – data to convert.

Returns post-formatted string generated from `meta`.

Return type str

1.3 Example Code

1.3.1 Cleanup

Getting rid of all dynamically created containers on Docker host:

```
from selenium_docker.base import ContainerFactory

factory = ContainerFactory.get_default_factory()
factory.scrub_containers()
```

1.4 Attribution

1.4.1 Projects

- Docker: Container creation and management software.
- Selenium: Browser automation and web driver bindings

1.4.2 Libraries

These libraries are integral to the functionality of selenium-docker.

- docker-py
- selenium
- gevent

Python Module Index

S

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

N

O

P

Q

R

S

V