arsenic Documentation

Release 0.1

HDE Inc

May 29, 2018

Tutorials

Warning: While this library is asynchronous, web drivers are **not**. You must call the APIs in sequence. The purpose of this library is to allow you to control multiple web drivers asynchronously or to use a web driver in the same thread as an asynchronous web server.

Arsenic is a library that allows you to control a web browser from async Python code. Use cases include testing of web applications, load testing, automating websites, web scraping or anything else you need a web browser for. It uses real web browsers using the Webdriver specification.

While it is built on top of aiohttp, it can be used from any asyncio-compatible framework, such as Tornado.

You can find the code of arsenic on Github and PyPI.

CHAPTER 1

Hello Arsenic

This tutorial will show you how to install arsenic and write a simple script that will use Firefox and Google Images to search for pictures of cats.

1.1 Prerequisites

This tutorial assumes you already have Python 3.6 and Firefox installed.

You will also need to install geckodriver. Download the latest release for your operating system from the releases page. Extract the binary executable from the archive and place it in the current directory. On OS X or Linux you might need to mark it as an executable using `chmod +x geckodriver` in your terminal.

1.2 Creating a virtual env

We will create a virtual env to install arsenic:

```
python3.6 -m venv env
```

Let's make sure that `pip` is up to date:

```
env/bin/pip install --upgrade pip
```

Let's install arsenic:

```
env/bin/pip install --pre arsenic
```

1.3 Writing the script

In your favourite text editor, create a file named `cats.py` and insert the following code:

```
import asyncio
import sys

from arsenic import get_session, keys, browsers, services

if sys.platform.startswith('win'):
    GECKODRIVER = './geckodriver.exe'
else:
    GECKODRIVER = './geckodriver'


async def hello_world():
    service = services.Geckodriver(binary=GECKODRIVER)
    browser = browsers.Firefox()
    async with get_session(service, browser) as session:
        await session.get('https://images.google.com/')
        search_box = await session.wait_for_element(5, 'input[name=q]')
        await search_box.send_keys('Cats')
        await search_box.send_keys(keys.ENTER)
        await asyncio.sleep(10)


def main():
    loop = asyncio.get_event_loop()
    loop.run_until_complete(hello_world())


if __name__ == '__main__':
    main()
```

Save it and in the terminal, run `python cats.py`. You should see an instance of Firefox starting, navigating to `https://images.google.com` and entering `Cats` in the search box, then submitting the search. The browser will then wait for 10 seconds for you to look at the cats before exiting.

CHAPTER 2

Arsenic with pytest

A common usage of webdrivers is for testing web applications. Thanks to the async nature of arsenic, you can test your async web applications from the same process and thread as you run your application.

In this guide, we will have a small aiohttp based web application and test it using pytest and pytest-asyncio.

2.1 Prerequisites

This guide assumes you are familiar with pytest and it's terminology.

2.2 Setup

You should already have Firefox and Geckodriver installed. Make sure your geckodriver is in your `PATH`.

Create a virtualenv and install the required dependencies:

> python3.6 -m venv env env/bin/pip install –upgrade pip env/bin/pip install –pre arsenic env/bin/pip install pytest-asyncio

2.3 App

Our app will have a single handler:

```
async def index(request):
    data = await request.post()
    name = data.get('name', 'World')
    return Response(status=200, content_type='text/html', body=f'''<html>
    <body>
        <h1>Hello {name}</h1>
        <form method='post' action='/'>
```

(continues on next page)

(continued from previous page)

```
            <input name='name' />
            <input type='submit' />
        </form>
    </body>
</html>''')


def build_app():
    app = Application()
    app.router.add_route('*', '/', index)
    return app
```

2.3.1 Fixture

To make our app easily available in tests, we'll write a pytest fixture which runs our app and provides the base url to it, since we will run it on a random port:

```
@pytest.fixture
async def app(event_loop):
    application = build_app()
    server = await event_loop.create_server(
        application.make_handler(),
        '127.0.0.1',
        0
    )
    try:
        for socket in server.sockets:
            host, port = socket.getsockname()
        yield f'http://{host}:{port}'
    finally:
        server.close()
```

2.4 Arsenic Fixture

We will also write a fixture for arsenic, which depends on the app fixture and provides a running Firefox session bound to the app:

```
@pytest.fixture
async def session(app):
    session = await start_session(
        services.Geckodriver(),
        browsers.Firefox(),
        bind=app
    )
    try:
        yield session
    finally:
        await stop_session(session)
```

2.5 Test

We will add a simple test which shows that the title on a GET request is Hello World, and if we submit the form it will become Hello followed by what we put into the text field:

```
async def test_index(session):
    await session.get('/')
    title = await session.wait_for_element(5, 'h1')
    text = await title.get_text()
    assert text == 'Hello World'
    form_field = await session.get_element('input[name="name"]')
    await form_field.send_keys('test')
    submit = await session.get_element('input[type="submit"]')
    await submit.click()
    title = await session.wait_for_element(5, 'h1')
    text = await title.get_text()
    assert text == 'Hello test'
```

2.6 Putting it all together

For this all to work, we'll need a few imports:

```
import pytest

from aiohttp.web import Application, Response

from arsenic import start_session, services, browsers, stop_session
```

And we also need to mark the file as asyncio for pytest to support async functions:

```
pytestmark = pytest.mark.asyncio
```

Now to put it all together, create a file called `test_pytest.py` and insert the following code:

```
import pytest

from aiohttp.web import Application, Response

from arsenic import start_session, services, browsers, stop_session

pytestmark = pytest.mark.asyncio


async def index(request):
    data = await request.post()
    name = data.get('name', 'World')
    return Response(status=200, content_type='text/html', body=f'''<html>
    <body>
        <h1>Hello {name}</h1>
        <form method='post' action='/'>
            <input name='name' />
            <input type='submit' />
        </form>
    </body>
```

(continues on next page)

(continued from previous page)

```
</html>''')


def build_app():
    app = Application()
    app.router.add_route('*', '/', index)
    return app


@pytest.fixture
async def app(event_loop):
    application = build_app()
    server = await event_loop.create_server(
        application.make_handler(),
        '127.0.0.1',
        0
    )
    try:
        for socket in server.sockets:
            host, port = socket.getsockname()
        yield f'http://{host}:{port}'
    finally:
        server.close()


@pytest.fixture
async def session(app):
    session = await start_session(
        services.Geckodriver(),
        browsers.Firefox(),
        bind=app
    )
    try:
        yield session
    finally:
        await stop_session(session)


async def test_index(session):
    await session.get('/')
    title = await session.wait_for_element(5, 'h1')
    text = await title.get_text()
    assert text == 'Hello World'
    form_field = await session.get_element('input[name="name"]')
    await form_field.send_keys('test')
    submit = await session.get_element('input[type="submit"]')
    await submit.click()
    title = await session.wait_for_element(5, 'h1')
    text = await title.get_text()
    assert text == 'Hello test'
```

To run it, simply execute `pytest test_pytest.py`.

CHAPTER 3

Arsenic on Windows

3.1 Internet Explorer 11

If you're trying to run Internet Explorer 11 using IEDriverServer, you must configure your computer a certain way. A helper function `arsenic.helpers.check_ie11_environment()` is provided and a helper script `arsenic-check-ie11.exe` can also be called from the command line.

To manually check the environment, ensure the following are all true:

- The *Protected Mode* setting of all zones in *Internet Options* must be set to the same value.
- *Enhanced Protected Mode* me *disabled.*
- The *Zoom Level* of Internet Explorer must be set to *100%*.
- The *Scale factor* in *Settings -> Display* must be set to *100*.

All of these, except for the *Scale factor* can be set with `arsenic-configure-ie11.exe`. This will disable *Protected Mode* for all zones! You may require elevated privileges to run this command.

3.2 Explicitly specify binaries

On unix systems, local services will work out of the box. On Windows, you need to explicitly pass the absolute path to the services binary to the service.

For example, if you installed `geckodriver.exe` to `C:\geckodriver\geckodriver.exe`, you have to instantiate your arsenic session like this:

```
from arsenic import get_session, services, browsers

async def example():
    service = services.Geckodriver(
        binary='C:\\geckodriver\\geckodriver.exe'
    )
```

(continues on next page)

(continued from previous page)

```
    browser = browsers.Firefox()
    async with get_session(service, browser) as session:
        ...
```

CHAPTER 4

Action Chains

Arsenic supports action chains that allow you to define a sequence of mouse and/or keyboard actions to perform in sequence (and in parallel).

Note: Action chains are only supported by a few browsers so far. For other browsers, arsenic will attempt to emulate them using older APIs, however it can only emulate a single mouse input.

4.1 Drag and drop

Drag and drop functionality can be implemented using `arsenic.actions.Mouse` together with `arsenic.actions.chain()` and `arsenic.session.Session.preform_actions()`.

Here is an example helper function which moves the mouse to an element and drags it by a specified amount of pixels:

```
from arsenic.actions import Mouse, chain
from arsenic.session import Element, Session


async def drag_and_drop(session: Session, source_element: Element, x_offset: int, y_
↪offset: int):
    mouse = Mouse()
    actions = chain(
        mouse.move_to(source_element),
        mouse.down(),
        mouse.move_by(x_offset, y_offset),
        mouse.up(),
    )
    await session.perform_actions(actions)
```

CHAPTER 5

Supported Browsers

Note: A Browser is considered supported if it is tested in continuous integration. Other browsers and browser versions might also work, but are not tested.

Table 1: Browsers

Browser Name	Supported Versions	Supported Service	OS
Firefox	59	Geckodriver 0.20.0	Linux, macOS, Windows 10
PhantomJS	1.9.8	PhantomJS 1.9.8	Linux, macOS, Windows 10
Google Chrome	65	Chromedriver 2.37	Linux, macOS, Windows 10
Internet Explorer	11 (See *Internet Explorer 11*)	IEDriverServer	Windows 10

Remote sessions are available via the *arsenic.services.Remote* but not all APIs may be available.

5.1 Headless Google Chrome

To use Google Chrome headless, use:

```
service = services.Chromedriver()
browser = browsers.Chrome(chromeOptions={
    'args': ['--headless', '--disable-gpu']
})
async with get_session(service, browser) as session:
    ...
```

5.2 Headless Firefox

To use Firefox headless, use:

```
service = services.Geckodriver()
browser = browsers.Firefox(firefoxOptions={
    'args': ['-headless']
})
async with get_session(service, browser) as session:
    ...
```

CHAPTER 6

API Reference

6.1 Main APIs

get_session(service, browser, bind=''):
Async context manager API to start/stop a browser session.

Parameters

- **service** (*arsenic.services.Service*) – The service which manages the browser instance.
- **browser** (*arsenic.browsers.Browser*) – The browser to start.
- **bind** (*str*) – Optional URL to bind the browser to.

Returns An async context manager.

start_session(service, browser, bind=''):
Coroutine to start new session.

Parameters

- **service** (*arsenic.services.Service*) – The service which manages the browser instance.
- **browser** (*arsenic.browsers.Browser*) – The browser to start.
- **bind** (*str*) – Optional URL to bind the br

Returns An object which can be passed to `stop_session()` to stop the session.

stop_session(session):
Coroutine to stop a session.

Parameters **session** (Object returned from `start_session()`.) – The session to stop.

Returns Nothing.

6.2 All APIs

6.2.1 arsenic.services

arsenic.services.**stop_process**(*process*)
Coroutine that stops a process.

Parameters **process** (asyncio.subprocess.Process) – Process to stop

arsenic.services.**sync_factory**(*func*)
Factory function which returns a coroutine which calls the function passed in. Useful to wrap a non-coroutine function as a coroutine for APIs that require coroutines.

arsenic.services.**subprocess_baed_service**(*cmd*, *service_url*, *log_file*)
Helper function for services that run a local subprocess.

Parameters

- **cmd** (*List[str]*) – Command to run.
- **service_url** (*str*) – URL at which the service will be available after starting.
- **log_file** (*io.TextIO*) – Log file for the service.

Return type *arsenic.webdriver.WebDriver*

class arsenic.services.**Service**
Abstract base class for services.

start()
Abstract method to start a service.

Return type *arsenic.webdriver.WebDriver*

class arsenic.services.**Geckodriver**(*log_file=os.devnull*, *binary='geckodriver'*)
Geckodriver service. Requires geckodriver 0.17 or higher.

Parameters

- **log_file** (*io.TextIO*) – Log file to use.
- **binary** (*str*) – Path to the geckodriver binary.
- **version_check** (*bool*) – Optional flag to disable version checking.

class arsenic.services.**Chromedriver**(*log_file=os.devnull*, *binary='chromedriver'*)
Chromedriver service.

Parameters

- **log_file** (*io.TextIO*) – Log file to use.
- **binary** (*str*) – Path to the chromedriver binary.

class arsenic.services.**Remote**(*url*, *auth=None*)
Remote service.

Parameters

- **url** (*str*) – URL of the remote webdriver.
- **auth** (*arsenic.http.Auth* or str.) – Optional authentication.

class arsenic.services.**PhantomJS**(*log_file=os.devnull*, *binary='phantomjs'*)
PhantomJS service.

Parameters

- **log_file** (*io.TextIO*) – Log file to use.
- **binary** (*str*) – Path to the PhantomJS binary.

class arsenic.services.**IEDriverServer**(*log_file=os.devnull*, *binary='IEDriverServer.exe'*)
Internet Explorer service.

Parameters

- **log_file** (*io.TextIO*) – Log file to use.
- **binary** (*str*) – Path to the IEDriverServer binary.

6.2.2 arsenic.browsers

class arsenic.browsers.**Browser**
Base browser class.

session_class
Session class to use for this browser. Should be *arsenic.session.Session* or a subclass thereof.

capabilities
A JSON serializable dictionary of capabilities to request.

defaults
Default capabilities.

__init__(overrides):**
When initializing a browser, you can override or extend the default capabilities of the browser.

class arsenic.browsers.**Firefox**(*Browser*)
Firefox with default capabilities.

class arsenic.browsers.**Chrome**(*Browser*)
Chrome with default capabilities.

class arsenic.browsers.**PhantomJS**(*Browser*)
PhantomJS with default capabilities.

class arsenic.browsers.**InternetExplorer**(*Browser*)
Internet Explorer with default capabilities.

6.2.3 arsenic.session

class arsenic.session.**Element**
A web element. You should not create instances of this class yourself, instead use *Session.get_element()* or *Session.get_elements()*.

get_text()
Coroutine to get the text of this element.

Return type str

send_keys(*keys*)
Coroutine to send a sequence of keys to this element. Useful for text inputs.

Parameters **keys** (*str*) – The keys to send. Use `arsenic.keys` for special keys.

send_file(path):
Coroutine to send a file to this element. Useful for file inputs.

Parameters **path** (*pathlib.Path*) – The local path to the file.

clear()
Coroutine to clear this element. Useful for form inputs.

click()
Coroutine to click on this element.

is_displayed()
Coroutine to check if this element is displayed or not.

Return type bool

is_enabled()
Coroutine to check if this element is enabled.

Return type bool

get_attribute(*name*)
Coroutine which returns the value of a given attribute of this element.

Parameters **name** (*str*) – Name of the attribute to get.

Return type str

select_by_value(*value*)
Coroutine to select an option by value. This is useful if this element is a select input.

Parameters **value** (*str*) – Value of the option to select.

get_rect()
Coroutine to get the location and size of the element.

Return type *arsenic.utils.Rect*

get_element(*selector*)
Coroutine to get a child element of this element via CSS selector.

Parameters **selector** (*str*) – CSS selector.

Return type *Element*

get_elements(*selector*)
Coroutine to get a list of child elements of this element via CSS selector.

Parameters **selector** (*str*) – CSS selector.

Return type List of *Element* instances.

class arsenic.session.**Session**

A webdriver session. You should not create instances of this class yourself, instead use `arsenic.get_session()` or `arsenic.start_session()`.

request(url, method='GET', data=UNSET):
Coroutine to perform a direct webdriver request.

Parameters

- **url** (*str*) – URL to call.
- **method** (*str*) – method to use
- **Any]** (*Dict[str,*) – data to send

get(url):

Coroutine to navigate to a given url.

param str url URL to navigate to.

get_url()
Coroutine to get the current URL.
Return type str

get_page_source()
Coroutine to get the source of the current page.
Return type str

get_element(*selector*)
Coroutine to get an element via CSS selector.
Parameters **selector** (*str*) – CSS selector of the element.
Return type *Element*

get_elements(*selector*)
Coroutine to get a list of elements via CSS selector.
Parameters **selector** (*str*) – CSS selector of the elements.
Return type List of *Element* instances.

wait_for_element(*timeout*, *selector*)
Coroutine like *get_element()*, but waits up to `timeout` seconds for the element to appear.
Parameters
- **timeout** (*int*) – Timeout in seconds.
- **selector** (*str*) – CSS selector.

Return type *Element*

wait_for_element_gone(*timeout*, *selector*)
Coroutine that waits up to `timeout` seconds for the element for the given CSS selector to no longer be available.
Parameters
- **timeout** (*int*) – Timeout in seconds.
- **selector** (*str*) – CSS Selector.

Return type None

add_cookie(*name*, *value*, *, *path=UNSET*, *domain=UNSET*, *secure=UNSET*, *expiry=UNSET*)
Coroutine to set a cookie.
Parameters
- **name** (*str*) – Name of the cookie.
- **value** (*str*) – Value of the cookie.
- **path** (*str*) – Optional, keyword-only path of the cookie.
- **domain** (*str*) – Optional, keyword-only domain of the cookie.
- **secure** (*bool*) – Optional, keyword-only secure flag of the cookie.
- **expiry** (*int*) – Optional, keyword-only expiration of the cookie.

Return type None

get_cookie(*name*)
Coroutine to get the value of a cookie.
Parameters **name** (*str*) – Name of the cookie.
Return type str

get_all_cookies()
Coroutine to get all cookies.
Return type dict

delete_cookie(*name*)
Coroutine to delete a specific cookie.
Parameters **name** (*str*) – Name of the cookie to delete.

delete_all_cookies()
Coroutine to delete all cookies.

execute_script(*script*, **args*)
Coroutine which executes a javascript script with the given arguments.
Parameters
- **script** (*str*) – Javascript script source to execute.
- **args** – Arguments to pass to the script. Must be JSON serializable.

set_window_size(*width*, *height*, *handle='current'*)
Coroutine to set the size of a given window.
Parameters
- **width** (*int*) – Width in pixels.
- **height** (*int*) – Height in pixels.
- **handle** (*str*) – ID of the window.

get_window_size(*handle='current'*)
Coroutine to get the size of a given window.
Parameters **handle** (*str*) – ID of the window.
Return type Tuple[int, int]

get_window_handle()
Coroutine to get the handle of the current window
Return type str

switch_to_window(*handle*)
Coroutine to set the handle of the current window
Parameters **handle** (*str*) – ID of the window.
Return type str

get_window_handles()
Coroutine to get the handles of all windows
Return type List[str]

get_alert_text()
Coroutine to return the text of an alert message.
Return type str

send_alert_text(*value*)
Coroutine to send text to an alert message.
Parameters **value** (*str*) – Value to send.

dismiss_alert()
Coroutine to dismiss an active alert.

accept_alert()
Coroutine to accept an active alert.

perform_actions(*actions*)
Coroutine to perform a series of actions. Use `arsenic.actions.chain()` to build the actions object.

get_screenshot()
Coroutine to take a screenshot of the top-level browsing context's viewport.
Return type `io.BytesIO`

close()
Coroutine to close this session.

`arsenic.session.`**CompatSession**()
Session subclass for webdrivers that do not support certain APIs.

6.2.4 arsenic.keys

This module holds constants of the special codes used for keyboard keys. These can be used in *arsenic.session.Element.send_keys()*.

For example, to send `'hello world'` followed by the enter key, you could do:

```
element.send_keys(f'hello world{keys.ENTER}')
```

NULL

CANCEL

HELP

BACKSPACE

TAB

CLEAR

RETURN

ENTER

SHIFT

CONTROL

ALT

PAUSE

ESCAPE

SPACE

PAGE_UP

PAGE_DOWN

END

HOME

LEFT

UP

RIGHT

DOWN

INSERT

DELETE

SEMICOLON

EQUALS

NUMPAD0

NUMPAD1

NUMPAD2

NUMPAD3

NUMPAD4

NUMPAD5

NUMPAD6

NUMPAD7

NUMPAD8

NUMPAD9

MULTIPLY

ADD

SEPARATOR

SUBTRACT

DECIMAL

DIVIDE

F1

F2

F3

F4

F5

F6

F7

F8

F9

F10

F11

F12

META

COMMAND

6.2.5 arsenic.actions

APIs to construct actions to be used in *arsenic.session.Session.perform_actions()*.

chain(*ticks):

Main API to construct actions for *arsenic.session.Session.perform_actions()*.

Parameters **ticks** (*Tick*) – A sequence of actions to chain.

Returns An object which can be passed to *arsenic.session.Session.perform_actions()*.

class arsenic.actions.**Button**(*Enum*)

left
Left mouse button.

middle
Middle mouse button.

right
Right mouse button.

class arsenic.actions.**Tick**
Class holding an action tick, which can have multiple actions of different devices.

You should not create instances of this class yourself, but rather use APIs on *Device* subclasses to create them.

Tick can be used with the OR operator (|) to combine actions of multiple devices.

class arsenic.actions.**Device**
Abstract base class for devices.

pause(duration):
Pause this device (do nothing) for a duration in milliseconds.

This is primarily used implicitly in action chains that have multiple devices but not all devices perform an action on each tick.

Parameters **duration** (*int*) – Duration in milliseconds of the pause.

Return type *Tick*

class arsenic.actions.**Pointer**(*Device*)
Base class for pointer devices.

move_to(element, duration=250) :
Moves the pointer to an element.

Parameters

- **element** (*arsenic.session.Element*) – Element to move to.
- **duration** (*int*) – Duration in milliseconds of this action.

Return type *Tick*

move_by(x, y, duration=250):
Move the pointer by a given number of pixels relative to the viewport.

Parameters

- **x** (*int*) – Number of pixels to move in the x-axis.
- **x** – Number of pixels to move in the y-axis.
- **duration** (*int*) – Duration in milliseconds for this action.

Return type *Tick*

down():
Holds the pointer down.

Return type *Tick*

up():
Lifts the pointer up.

Return type *Tick*

Mouse(Pointer):
Mouse device.

down(button=Button.left):
Hold down a mouse button.

Parameters **button** (*Button*) – Which button to hold down.

Return type *Tick*

up(button=Button.right):
Releases a mouse button.

Parameters **button** (*Button*) – Which button to release.

Return type *Tick*

Pen(Pointer):
A pen device.

Touch(Pointer):
A touch device.

Keyboard(Device):
A keyboard device.

down(key):
Holds down a specific key.

Parameters **key** (*str*) – Which key to hold down.

Return type *Tick*

up(key):
Releases a specific key.

Parameters **key** (*str*) – Which key to release.

Return type *Tick*

6.2.6 arsenic.errors

exception ArsenicError
Base exception class used by arsenic.

exception OperationNotSupported
Exception raised for operations not supported by a given webdriver and/or browser.

exception WebdriverError
Base class for webdriver-side errors.

exception UnkownArsenicError
Exception used when there was a webdriver-side error, but arsenic could not figure out what the error was.

exception ArsenicTimeout
Raised when *arsenic.webdriver.WebDriver.wait()*, arsenic.session.Session.wait() or a higher level wait API times out.

The following are specific exceptions which may be returned from a webdriver. Consult the webdriver specification for details.

exception NoSuchElement

exception NoSuchFrame

exception UnknownCommand

exception StaleElementReference

exception ElementNotVisible

exception InvalidElementState

exception UnknownError

exception ElementNotInteractable

exception ElementIsNotSelectable

exception JavascriptError

exception Timeout

exception NoSuchWindow

exception InvalidCookieDomain

exception UnableToSetCookie

exception UnexpectedAlertOpen

exception NoSuchAlert

exception ScriptTimeout

exception InvalidElementCoordinates

exception IMENotAvailable

exception IMEEngineActivationFailed

exception InvalidSelector

exception MoveTargetOutOfBounds

6.2.7 arsenic.webdriver

class arsenic.webdriver.**WebDriver**

Class representing a webdriver. You should not create instances of this class yourself, instead let `arsenic.get_session()` or `arsenic.start_session()` create and manage one for you.

wait(*timeout*, *func*, **exceptions*)

Coroutine which waits for up to `timeout` seconds for the coroutine function `func` to return a truthy value, calling it repeatedly.

If the coroutine function `func` raises an exception given in `exceptions`, it is ignored.

If `func` returns a truthy value, it is returned to the caller of this method.

Parameters

- **timeout** (*int*) – Timeout in seconds.
- **func** (*Coroutine function taking no arguments and returning a truthy value if the condition is met.*) – Callback which checks if the condition is met.
- **exceptions** (*Exception*) – Optional exceptions to ignore.

6.2.8 arsenic.connection

arsenic.connection.**WEB_ELEMENT**
Constant string used to denote web elements in the web driver protocol.

class arsenic.connection.**Connection**(*session*, *prefix*)
Connection class to use for communication with a webdriver. This class operates with a `prefix` to make it easier to use internally.

Parameters

- **session** (`aiohttp.client.ClientSession`) – Aiohttp client session.
- **prefix** (*str*) – Prefix for this connection.

request(*, *url*, *method*, *data=None*, *raw=False*)
Coroutine to do an HTTP request. All arguments are keyword only.

Parameters

- **url** (*str*) – The URL to send the request to.
- **method** (*str*) – HTTP method to use.
- **data** – Optional data to send. Must be JSON serializable.
- **raw** (*bool*) – Optional flag to get the raw response data instead of unwrapped data.

upload_file(path):
Coroutine that uploads a file. This is used for remote webdrivers. This is used internally by `arsenic.session.Element.send_file()`.

This method is no-op by default.

Parameters **path** (`pathlib.Path`) – The (local) path of the file to upload.

Returns A path indicating the remote path.

Return type `pathlib.Path`

prefixed(prefix):
Returns a new connection, inheriting the HTTP session, with the extra prefix given.

Parameters **prefix** (*str*) – Prefix to add to the current prefix.

Return type *Connection*

class arsenic.connection.**RemoteConnection**
Connection class for remote webdrivers. Most notably, `Connection.upload_file()` is no longer a no-op operation.

6.2.9 arsenic.http

This module holds http authentication helpers.

class arsenic.http.**Auth**
Abstract base class for authentication.

get_headers()
Abstract method which returns a dictionary of headers.

class arsenic.http.**BasicAuth**(*username*, *password*)
Basic auth implementation of the *Auth* abstract class.

6.2.10 arsenic.utils

arsenic.utils.**free_port**()
Returns a free port.

Return type int

class arsenic.utils.**Rect**

x

y

width

height

CHAPTER 7

Testing

7.1 Basics

7.1.1 Native

This is the easiest way to get started running the tests. The test runner will try to find all local browsers it supports and run the tests on them (headlessly when possible). Note that on top of having the browsers (eg Firefox) installed, you also need their webdriver installed (eg Geckodriver).

- Install the test requirements using `pip install -r tests/requirements.txt`.
- Install arsenic itself `pip install -e .`.
- Run `pytest`.

7.1.2 Docker/CircleCI

- Install the CircleCI command line tool.
- Run `circleci build`

Warning: This will pull a massive (1GB+) docker file. Make sure you have the bandwidth and disk space for this.

7.2 Advanced

7.2.1 Explicitly specify drivers

The test suite will try to find the webdrivers and browsers automatically, but sometimes this will fail. You can specify the location of the drivers yourself via the following environment variables:

- `GECKODRIVER_BINARY`: Path to the geckodriver
- `CHROMEDRIVER_BINARY`: Path to the chromedriver
- `PHANTOMJS_BINARY`: Path to phantomjs
- `IEDRIVERSERVER_BINARY`: Path to IEDriverServer.exe

7.2.2 Only run tests on one browser

Use the `-k` flag of `pytest` to select a specific browser. For example, to only run on chrome, use `-k chrome_session`.

7.2.3 Testing remote drivers

The test suite is set up to test remotely via Browserstack. To do so, set the following environment variables:

- `BROWSERSTACK_API_KEY`: Your Browserstack API key.
- `BROWSERSTACK_LOCAL_IDENTIFIER`: Identifier for your build.
- **`REMOTE_BROWSER`: A JSON encoded object. It requires at least the `"type"` key, which is the name of the browser class** in arsenic. All other keys will be passed to the arsenic browser class.
- `REMOTE_SERVICE`: URL to the remote service executor. Include the username/password here. Eg `https://<user>:<pass>@hub.browserstack.com/wd/hub`

7.2.4 Adding a new browser to test

Open the file `tests/conftest.py`. Add an extra function to the `params` list in the `pytest.fixture` decorator call for `async def session`. The function should be an async context manager, which takes the root url to the app to test as an argument and yields a `Session`.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

H

I

J

L

M

N

O

P

R

S

T

U

W

X

Y