widgetastic.core

Release 0.31.0

Dec 17, 2018

Contents

Contents:

CHAPTER 1

Introduction

Widgetastic is a Python library designed to abstract out web UI widgets into a nice object-oriented layer. This library includes the core classes and some basic widgets that are universal enough to exist in this core repository.

1.1 Features

- Individual interactive and non-interactive elements on the web pages are represented as widgets; that is, classes with defined behaviour. A good candidate for a widget might be something a like custom HTML button.
- Widgets are grouped on Views. A View descends from the Widget class but it is specifically designed to hold other widgets.
- All Widgets (including Views because they descend from them) have a read/fill interface useful for filling in forms etc. This interface works recursively.
- Views can be nested.
- Widgets defined on Views are read/filled in exact order that they were defined. The only exception to this default behaviour is for nested Views as there is limitation in the language. However, this can be worked around by using `View.nested` decorator on the nested View.
- Includes a wrapper around selenium functionality that tries to make the experience as hassle-free as possible including customizable hooks and built-in "JavaScript wait" code.
- Views can define their root locators and those are automatically honoured in the element lookup in the child Widgets.
- Supports *Parametrized views*.
- Supports *Switchable conditional views*.
- Supports *Widget including*.
- Supports *Version picking*.
- Supports automatic *Constructor object collapsing* for objects passed into the widget constructors.

- Supports *Fillable objects* that can coerce themselves into an appropriate filling value.
- Supports many Pythons! 2.7, 3.5, 3.6 and PyPy are officially supported and unit-tested in CI.

1.2 What this project does NOT do

- A complete testing solution. In spirit of modularity, we have intentionally designed our testing system modular, so if a different team likes one library, but wants to do other things different way, the system does not stand in its way.
- UI navigation. As per previous bullet, it is up to you what you use. In CFME QE, we use a library called navmazing, which is an evolution of the system we used before. You can devise your own system, use ours, or adapt something else.
- UI models representation. Doing nontrivial testing usually requires some sort of representation of the stuff in the product in the testing system. Usually, people use classes and instances of these with their methods corresponding to the real actions you can do with the entities in the UI. Widgetastic offers integration for such functionality (Fillable objects), but does not provide any framework to use.
- Test execution. We use pytest to drive our testing system. If you put the two previous bullets together and have a system of representing, navigating and interacting, then writing a simple boilerplate code to make the system's usage from pytest straightforward is the last and possibly simplest thing to do.

CHAPTER 2

Basic usage

ATTENTION!: Read the *Widgetastic usage guidelines* carefully before starting out.

This sample only represents simple UI interaction.

```
from selenium import webdriver
from widgetastic.browser import Browser
from widgetastic.widget import View, Text, TextInput


# Subclass the default browser, add product_version property, plug in the hooks ...
class CustomBrowser(Browser):
    pass

# Create a view that represents a page
class MyView(View):
    a_text = Text(locator='.//h3[@id="title"]')
    an_input = TextInput(name='my_input')

    # Or a portion of it
    @View.nested  # not necessary but you need it if you need to keep things ordered
    class my_subview(View):
        # You can specify a root locator, then this view responds to is_displayed and␣
↪can be
        # used as a parent for widget lookup
        ROOT = 'div#somediv'
        another_text = Text(locator='#h2')  # See "Automatic simple CSS locator␣
↪detection"

selenium = webdriver.Firefox()  # For example
browser = CustomBrowser(selenium)

# Now we have the widgetastic browser ready for work
# Let's instantiate a view.
a_view = MyView(browser)
# ^^ you would typically come up with some way of integrating this in your framework.
```

(continues on next page)

(continued from previous page)

```
# The defined widgets now work as you would expect
a_view.read()  # returns a recursive dictionary of values that all widgets provide␣
↪via read()
a_view.a_text.text  # Accesses the text
# but the .text is widget-specific, so you might like to use just .read()
a_view.fill({'an_input': 'foo'})  # Fills an_input with foo and returns boolean␣
↪whether anything changed
# Basically equivalent to:
a_view.an_input.fill('foo')  # Since views just dispatch fill to the widgets based on␣
↪the order
a_view.an_input.is_displayed
```

Typically, you want to incorporate a system that would do the navigation (like navmazing for example), as Widgetastic only facilitates UI interactions.

An example of such integration is currently **TODO**, but it will eventually appear here once a PoC for a different project will happen.

CHAPTER 3

Advanced usage

3.1 Simplified nested form fill

When you want to separate widgets into logical groups but you don't want to have a visual clutter in the code, you can use dots in fill keys to signify the dictionary boundaries:

```
# This:
view.fill({
    'x': 1,
    'foo.bar': 2,
    'foo.baz': 3,
})

# Is equivalent to this:
view.fill({
    'x': 1,
    'foo': {
        'bar': 2,
        'baz': 3,
    }
})
```

3.2 Version picking

By version picking you can tackle the challenge of widgets changing between versions.

In order to use this feature, you have to provide `product_version` property in the Browser which should return the current version (ideally `widgetastic.utils.Version`, otherwise you would need to redefine the `widgetastic.utils.VersionPick.VERSION_CLASS` to point at you version handling class of choice) of the product tested.

Then you can version pick widgets on a view for example:

```
from widgetastic.utils import Version, VersionPick
from widgetastic.widget import View, TextInput

class MyVerpickedView(View):
    hostname = VersionPick({
        # Version.lowest will match anything lower than 2.0.0 here.
        Version.lowest(): TextInput(name='hostname'),
        '2.0.0': TextInput(name='host_name'),
    })
```

When you instantiate the `MyVerpickedView` and then subsequently access `hostname` it will automatically pick the right widget under the hood.

`widgetastic.utils.VersionPick` is not limited to resolving widgets and can be used for anything.

You can also pass the `widgetastic.utils.VersionPick` instance as a constructor parameter into widget instantiation on the view class. Because it utilizes *Constructor object collapsing*, it will resolve itself automatically.

3.3 Parametrized views

If there is a repeated pattern on a page that differs only by eg. a title or an id, widgetastic has a solution for that. You can use a `widgetastic.widget.ParametrizedView` that takes an arbitrary number of parameters and then you can use the parameters eg. in locators.

```
from widgetastic.utils import ParametrizedLocator, ParametrizedString
from widgetastic.widget import ParametrizedView, TextInput

class MyParametrizedView(ParametrizedView):
    # Defining one parameter
    PARAMETERS = ('thing_id', )
    # ParametrizedLocator coerces to a string upon access
    # It follows similar formatting syntax as .format
    # You can use the xpath quote filter as shown
    ROOT = ParametrizedLocator('.//thing[@id={thing_id|quote}]')

    # Widget definition *args and values of **kwargs (only the first level) are␣
↪processed as well
    widget = TextInput(name=ParametrizedString('#asdf_{thing_id}'))

# Then for invoking this:
view = MyParametrizedView(browser, additional_context={'thing_id': 'foo'})
```

It is also possible to nest the parametrized view inside another view, parametrized or otherwise. In this case the invocation of a nested view looks like a method call, instead of looking like a property. The invocation supports passing the arguments both ways, positional and keyword based.

```
from widgetastic.utils import ParametrizedLocator, ParametrizedString
from widgetastic.widget import ParametrizedView, TextInput, View

class MyView(View):
    class this_is_parametrized(ParametrizedView):
        # Defining one parameter
        PARAMETERS = ('thing_id', )
        # ParametrizedLocator coerces to a string upon access
        # It follows similar formatting syntax as .format
```

(continues on next page)

(continued from previous page)

```
        # You can use the xpath quote filter as shown
        ROOT = ParametrizedLocator('.//thing[@id={thing_id|quote}]')

        # Widget definition *args and values of **kwargs (only the first level) are␣
↪processed as well
        the_widget = TextInput(name=ParametrizedString('#asdf_{thing_id}'))

# We create the root view
view = MyView(browser)
# Now if it was an ordinary nested view, view.this_is_parametrized.the_widget would␣
↪give us the
# nested view instance directly and then the the_widget widget. But this is a␣
↪parametrized view
# and it will give us an intermediate object whose task is to collect the parameters␣
↪upon
# calling and then pass them through into the real view object.
# This example will be invoking the parametrized view with the exactly same param␣
↪like the
# previous example:
view.this_is_parametrized('foo')
# So, when we have that view, you can use it as you are used to
view.this_is_parametrized('foo').the_widget.do_something()
# Or with keyword params
view.this_is_parametrized(thing_id='foo').the_widget.do_something()
```

The parametrized views also support list-like access using square braces. For that to work, you need the `all` classmethod defined on the view so Widgetastic would be aware of all the items. You can access the parametrized views by member index `[i]` and slice `[i:j]`.

It is also possible to iterate through all the occurences of the parametrized view. Let's assume the previous code sample is still loaded and the `this_is_parametrized` class has the `all()` defined. In that case, the code would like like this:

```
for p_view in view.this_is_parametrized:
    print(p_view.the_widget.read())
```

This sample code would go through all the occurences of the parametrization. Remember that the `all` classmethod IS REQUIRED in this case.

You can also pass the :py:class:utils‘ParametrizedString‘ instance as a constructor parameter into widget instantiation on the view class. Because it utilizes *Constructor object collapsing*, it will resolve itself automatically.

3.4 Constructor object collapsing

By using `widgetastic.utils.ConstructorResolvable` you can create an object that can lazily resolve itself into a different object upon widget instantiation. This is used eg. for the *Version picking* where `widgetastic.utils.VersionPick` descends from this class or for the parametrized strings. Just subclass this class and implement `.resolve(self, parent_object)` where `parent_object` is the to-be parent of the widget.

3.5 Fillable objects

I bet that if you have ever used modelling approach to the entities represented in the product, you have come across filling values in the UI and if you wanted to select the item representing given object in the UI, you had to pick a

correct attribute and know it. So you had to do something like this (simplified example)

```
some_form.item.fill(o.description)
```

If you let the class of `o` implement `widgetastic.utils.Fillable`, you can implement the method `.as_fill_value` which should return such value that is used in the UI. In that case, the simplification is as follows.

```
some_form.item.fill(o)
```

You no longer have to care, the object itself know how it will be displayed in the UI. Unfortunately this does not work the other way (automatic instantiation of objects based on values read) as that would involve knowledge of metadata etc. That is a possible future feature.

3.6 Widget including

DRY is useful, right? Widgetastic thinks so, so it supports including widgets into other widgets. Think about it more like C-style include, what it does is that it makes the receiving widget aware of the other widgets that are going to be included and generates accessors for the widgets in included widgets so if "flattens" the structure. All the ordering is kept. A simple example.

```
class FormButtonsAdd(View):
    add = Button('Add')
    reset = Button('Reset')
    cancel = Button('Cancel')

class ItemAddForm(View):
    name = TextInput(...)
    description = TextInput(...)

    # ...
    # ...

    buttons = View.include(FormButtonsAdd)
```

This has the same effect like putting the buttons directly in `ItemAddForm`.

You **ABSOLUTELY MUST** be aware that in background this is not including in its literal sense. It does not take the widget definitions and put them in the receiving class. If you access the widget that has been included, what happens is that you actually access a descriptor proxy that looks up the correct included hosting widget where the requested widget is hosted (it actually creates it on demand), then the correct widget is returned. This has its benefit in the fact that any logical structure that is built inside the included class is retained and works as one would expect, like parametrized locators and such.

All the included widgets in the structure share their parent with the widget where you started including. So when instantiated, the underlying `FormButtonsAdd` has the same parent widget as the `ItemAddForm`. I did not think it would be wise to make the including widget a parent for the included widgets due to the fact widgetastic fences the element lookup if `ROOT` is present on a widget/view. However, `widgetastic.widget.View.include` supports `use_parent=True` option which makes included widgets use including widget as a parent for rare cases when it is really necessary.

3.7 Switchable conditional views

If you have forms in your product whose parts change depending on previous selections, you might like to use the `widgetastic.widget.ConditionalSwitchableView`. It will allow you to represent different kinds of

views under one widget name. An example might be a view of items that can use icons, table, or something else. You can make views that have the same interface for all the variants and then put them together using this tool. That will allow you to interact with the different views the same way. They display the same informations in the end.

```
class SomeForm(View):
    foo = Input('...')
    action_type = Select(name='action_type')

    action_form = ConditionalSwitchableView(reference='action_type')

    # Simple value matching. If Action type 1 is selected in the select, use this␣
↪view.
    # And if the action_type value does not get matched, use this view as default
    @action_form.register('Action type 1', default=True)
    class ActionType1Form(View):
        widget = Widget()

    # You can use a callable to declare the widget values to compare
    @action_form.register(lambda action_type: action_type == 'Action type 2')
    class ActionType2Form(View):
        widget = Widget()

    # With callable, you can use values from multiple widgets
    @action_form.register(
        lambda action_type, foo: action_type == 'Action type 2' and foo == 2)
    class ActionType2Form(View):
        widget = Widget()
```

You can see it gives you the flexibility of decision based on the values in the view.

This example as shown (with Views) will behave like the `action_form` was a nested view. You can also make a switchable widget. You can use it like this:

```
class SomeForm(View):
    foo = Input('...')
    bar = Select(name='bar')

    switched_widget = ConditionalSwitchableView(reference='bar')

    switched_widget.register('Action type 1', default=True, widget=Widget())
```

Then instead of switching views, it switches widgets.

CHAPTER 4

Internal structure of Widgetastic

Widgetastic consists of 2 main parts:

- *Selenium browser wrapper*
- *Widget system*

CHAPTER 5

Selenium browser wrapper

This part of the framework serves the purpose of simplifying the interactions with Selenium and also handling some of the quirks we have discovered during development of our testing framework. It also supports "nesting" of the browsers in relation to specific widgets, so it is then easier in the widget layer to implement the lookup fencing. Majority of this functionality is implemented in `widgetastic.browser.Browser`.

Lookup fencing is a technique that enables the programmer to write locators that are relative to its hosting object. When such locator gets resolved, the parent element is resolved first (and it continues recursively until you hit an "unwrapped" browser that is just a browser). This behaviour is not visible to the outside under normal circumstances and it is achieved by `widgetastic.browser.BrowserParentWrapper`.

The `widgetastic.browser.Browser` class has some convenience features like *Automatic detection of simple CSS locators* and *Automatic visibility precedence selection*.

5.1 Automatic detection of simple CSS locators

By default, all string locators are considered XPath, but in each place where a locator gets passed into Widgetastic you can leverage automatic simple CSS locator detection. If a string corresponds to the pattern of *tagname#id.class1.class2* where the tag is optional and at least one *id* or *class* is present, it considers it a CSS locator.

If you want to use a complex CSS locator or a different lookup type, you can use selenium-smart-locator library that is used underneath to process all the locators. You can consult the documentation and pass instances of `Locator` instead of a string.

This library is already in the requirements, so it is not necessary to install it.

5.2 Automatic visibility precedence selection

Under normal circumstances, Selenium's `find_element` always returns the first element from the query result. But what if there are multiple such elements matching the query, the first one is invisible for a reason and the second one is displayed?

Widgetastic's `widgetastic.browser.Browser.element()` checks for visibility if there are multiple elements as a result of the query. It returns the first visible element, and if none of the elements are visible, it returns the first one as in the raw Selenium.

CHAPTER 6

Widget system

The widget system consists of number of supporting classes and finally the `widgetastic.widget.Widget` class itself.

Let's first talk about how Widgetastic makes sure that although the user "instantiates" the widgets without any additional context, the widgets themselves receive everything they need in a consistent manner.

The important thing is in `widgetastic.widget.Widget.__new__()`. `__new__` is the dunder method responsible for creating the object from the class and it is called before `__init__` gets called. Widgetastic exploits this functionality. The `Widget` class needs to know the instance of another `Widget` or `widgetastic.browser.Browser` to be instantiated. Since we do not know it at the moment of class definition, we need to **defer** it. And that is where `widgetastic.widget.WidgetDescriptor` comes into play.

6.1 How the WidgetDescriptor works?

The beforementioned `__new__` method checks if the first argument or the `parent` kwargument is specified. If yes, it then lets python create the object as usual. If it is not passed, an instance of `widgetastic.widget.WidgetDescriptor` is returned instead. The descriptor class contains these three most important informations:

- The class object (*yes, class, not an instance*)
- args
- kwargs

The `WidgetDescriptor` is named a descriptor for a reason. Because it implements the `__get__` method, it is a Python descriptor. Descriptors allow you to be in the access loop when you access an attribute on an object. This brings us to the deferring and how it is done.

Simply said, once you access the widget (`view.widget`), the descriptor implementation in the `WidgetDescriptor` just instantiates the class with the args and kwargs that were stored on definition and returns it instead of returning itself.

In real implementation, caching and other things make this process more complex, but under the hood this is what happens.

`widgetastic.widget.WidgetDescriptor` is also ordinal. Each one has a unique `_seq_id` attribute which increments for each new `widgetastic.widget.WidgetDescriptor` created. Therefore although it is not possible with pure Python facilities, Widgetastic can order the widgets in the order as they were defined.

All this also means that if you are playing with single widgets in eg. IPython, you always need to stick a browser obejct or another widget as the first parameter. You also need to make sure `parent` and `logger` are passed to `super()` so the widget object can be properly initialized.

```
class MyNewWidget(Widget):
    def __init__(self, parent, myarg1, logger=None):
        Widget.__init__(self, parent, logger=logger)
        self.myarg1 = myarg1
```

6.2 The magic of metaclasses

`widgetastic.widget.Widget` class has a custom metaclass - `widgetastic.widget.WidgetMetaclass`. Metaclasses create classes the same way classes create instances. `widgetastic.widget.WidgetMetaclass` processes the class definition and builds a couple of helper attributes to facilitate eg. name resolution, since the widget definition cannot know by itself what was the name you assigned it on the class. It also wraps fill/read with logging, generates a `widgetastic.widget.Widget.__locator__()` if `ROOT` is present, ...

6.3 Caching of widgets

Widget instances are cached on the hosting widget. Only plain widgets get cached, because the caching system is too simple so far to support parametrized views and such advanced functionality. The descriptor object is used as the cache key, the widget instance is the value.

6.4 `__locator__()` and `__element__()` protocol

To ensure good structure, a protocol of two methods was introduced. Let's talk a bit about them.

`__locator__()` method is not implemented by default on `Widget` class. Its sole purpose is to serve a locator of the object itself, so when the object is thrown in element lookup, it returns the result for the locator returned by this method. This method must return a locator, be it a valid locator string, tuple or another locatable object. If a webelement is returned by `__locator__()`, a warning will be produced into the log.

`__locator__()` is auto-generated when `ROOT` attribute is present on the class with a valid locator.

`__element__()` method has a default implementation on every widget. Its purpose is to look up the root element from `__locator__()`. It is present because the machinery that digests the objects for element lookup will try it first. `__element__()`'s default implementation looks up the `__locator__()` in the *parent browser*. That is important, because that allows simpler structure for the browser wrapper.

Combination of these methods ensures, that while the widget's root element is looked up in parent browser, which fences the lookup into the parent widget, all lookups inside the widget, like child widgets or other browser operations operate within the widget's root element, eliminating the need of passing the parent element.

CHAPTER 7

Widgetastic usage guidelines

Anyone using this library should consult these guidelines whether one is not violating any of them.

- While writing new widgets:
  - They must have the standard read/fill interface
    * `read()` -> `object`
      - Whatever is returned from `read()` must be compatible with `fill()`. Eg. `obj.fill(obj.read())` must work at any time.
      - `read()` may throw a `DoNotReadThisWidget` exception if reading the widget is pointless (eg. in current form state it is hidden). That is achieved by invoking the `do_not_read_this_widget()` function.
    * `fill(value)` -> `True|False`
      - `fill(value)` must be able to ingest whatever was returned by `read()`. Eg. `obj.fill(obj.read())` must work at any time.
      - An exception to this rule is only acceptable in the case where this 1:1 direct mapping would cause severe inconvenience.
      - `fill` MUST return `True` if it changed anything during filling
      - `fill` MUST return `False` if it has not changed anything during filling
    * Any of these methods may be omitted if it is appropriate based on the UI widget interactions.
    * It is recommended that all widgets have at least `read()` but in cases like buttons where you don't read or fill, it is understandable that there is neither of those.
  - `__init__` must be in accordance to the concept
    * If you want your widget to accept parameters `a` and `b`, you have to create signature like this:
      - `__init__(self, parent, a, b, logger=None)`
    * The first line of the widget must call out to the root class in order to set things up properly:
      - `Widget.__init__(self, parent, logger=logger)`

  - Widgets MUST define `__locator__` in some way. Views do not have to, but can do it to fence the element lookup in its child widgets.
    * You can write `__locator__` method yourself. It should return anything that can be turned into a locator by `smartloc.Locator`
      · `'#foo'`
      · `'//div[@id="foo"]'`
      · `smartloc.Locator(xpath='...')`
      · et cetera
    * `__locator__` MUST NOT return `WebElement` instances to prevent `StaleElementReferenceException`
    * If you use a `ROOT` class attribute, especially in combination with `ParametrizedLocator`, a `__locator__` is generated automatically for you.
  - Widgets should keep its internal state in reasonable size. Ideally none, but eg. caching header names of tables is perfectly acceptable. Saving `WebElement` instances in the widget instance is not recommended.
    * Think about what to cache and when to invalidate
    * Never store `WebElement` objects.
    * Try to shorten the lifetime of any single `WebElement` as much as possible
      · This will help against `StaleElementReferenceException`
  - Widgets shall log using `self.logger`. That ensures the log message is prefixed with the widget name and location and gives more insight about what is happening.

- When using Widgets (and Views)
  - Bear in mind that when you do `MySuperWidget('foo', 'bar')` in ipython, you are not getting an actual widget object, but rather an instance of WidgetDescriptor
  - In order to create a real widget object, you have to have widgetastic `Browser` instance around and prepend it to the arguments, so the call to create a real widget instance would look like:
    * `MySuperWidget(wt_browser, 'foo', 'bar')`
  - This browser prepending is done automatically by `WidgetDescriptor` when you access it on a `View` or another `Widget`
    * All of these means that the widget objects are created lazily.
  - Views can be nested
    * Filling and reading nested views is simple, each view is read/filled as a dictionary, so the required dictionary structure is exactly the same as the nested class structure
  - Views remember the order in which the Widgets were placed on it. Each `WidgetDescriptor` has a sequential number on it. This is used when filling or reading widgets, ensuring proper filling order.
    * This would normally also apply to Views since they are also descendants of `Widget`, but since you are not instantiating the view when creating nested views, this mechanism does not work.
      · You can ensure the `View` gets wrapped in a `WidgetDescriptor` and therefore in correct order by placing a `@View.nested` decorator on the nested view.
  - Views can optionally define `before_fill(values)` and `after_fill(was_change)`
    * `before_fill` is invoked right before filling gets started. You receive the filling dictionary in the values parameter and you can act appropriately.

    * `after_fill` is invoked right after the fill ended, `was_change` tells you whether there was any change or not.

- When using `Browser` (also applies when writing Widgets)
  - Ensure you don't invoke methods or attributes on the `WebElement` instances returned by `element()` or `elements()`
  - Eg. instead of `element.text` use `browser.text(element)` (applies for all such circumstances). These calls usually do not invoke more than their original counterparts. They only invoke some workarounds if some know issue arises. Check what the `Browser` (sub)class offers and if you miss something, create a PR
  - You don't necessarily have to specify `self.browser.element(..., parent=self)` when you are writing a query inside a widget implementation as widgetastic figures this out and does it automatically.
  - Most of the methods that implement the getters, that would normally be on the element object, take an argument or two for themselves and the rest of `*args` and `**kwargs` is shoved inside `element()` method for resolution, so constructs like `self.browser.get_attribute('id', self.browser.element('locator', parent=foo))` are not needed. Just write `self.browser.get_attribute('id', 'locator', parent=foo)`. Check the method definitions on the `Browser` class to see that.
  - `element()` method tries to apply a rudimentary intelligence on the element it resolves. If a locator resolves to a single element, it returns it. If the locator resolves to multiple elements, it tries to filter out the invisible elements and return the first visible one. If none of them is visible, it just returns the first one. Under normal circumstances, standard selenium resolution always returns the first of the resolved elements.
  - DO NOT use `element.find_elements_by_<method>('locator')`, use `self.browser.element('locator', parent=element)`. It is about as same long and safer.
    * Eventually I might wrap the elements as well but I decided to not complicate things for now.

No current exceptions are to be taken as a precedent.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search