Stere Documentation

Release 0.8.0

Joshua Fehler

Sep 22, 2020

Contents:

1 Documentation **1**

1.1 Getting Started . . . 1
1.2 Pages . . . 2
1.3 Fields . . . 4
1.4 Splinter Integration . . . 5
1.5 Appium Integration . . . 7
1.6 Areas . . . 8
1.7 Repeating . . . 9
1.8 Workflows . . . 9
1.9 Fields Returning Objects . . . 9
1.10 Best Practices . . . 10

2 Indices and tables **15**

Index **17**

CHAPTER 1

Documentation

1.1 Getting Started

1.1.1 Requirements

Python >= 3.6

1.1.2 Installation

Stere can be installed with pip using the following command:

```
pip install stere
```

1.1.3 Setup

Specifying the automation library

Using a *stere.ini* file, the automation library used can be specified. This determines which library specific Fields are loaded.

While only Splinter and Appium have custom Fields to take advantage of their specific capabilities, any automation library that implements an API similar to Selenium should be possible to connect to Stere.

splinter is used by default, and *appium* is supported. Any other value will be accepted, which will result in no specific Fields being loaded.

```
[stere]
library = appium
```

Stere.browser

Stere requires a browser (aka driver) to work with. This can be any class that ultimately drives automation. Pages, Fields, and Areas inherit their functionality from this object.

Here's an example with Splinter:

```
from stere import Stere
from splinter import Browser

Stere.browser = Browser()
```

As long as the base Stere object has the browser set, the browser's functionality is passed down to everything else.

Stere.base_url

Optionally, an attribute called base_url can be provided a string that will be used as the base for all urls returned by *Page.page_url*

```
from stere import Stere
from splinter import Browser

Stere.browser = Browser()
Stere.base_url = 'http://foobar.com/'

class MyPage(Page):
    def __init__(self):
        self.url_suffix = 'mysuffix'

>>> MyPage().page_url == 'http://foobar.com/mysuffix'
```

Stere.url_navigator

Optionally, an attribute called *url_navigator* can be provided a string that maps to the method in the browser that opens a page.

In Splinter's case, this is the *visit* method.

```
from stere import Stere
from splinter import Browser

Stere.browser = Browser()
Stere.url_navigator = 'visit'
```

This attribute is used by the *Page* class to make url navigation easier.

1.2 Pages

class stere.**Page**

Represents a single page in an application. The Page class is the base which all Page Objects should inherit from.

Inheriting from Page is not required for Fields or Areas to work.

All attribute calls that fail are then tried on the browser attribute. This allows classes inheriting from Page to act as a proxy to whichever browser/driver is being used.

Using Splinter's browser.url method as an example, the following methods are analogous:

```
>>> MyPage.url == MyPage.browser.url == browser.url
```

The choice of which syntax to use depends on how you want to write your test suite.

page_url
Get a full URL from stere's base_url and a Page's url_suffix.

Uses urllib.parse.urljoin to combine the two.

navigate()
When the base Stere object has been given the *url_navigator* attribute, and a Page Object has a *page_url* attribute, the *navigate()* method can be called.

This method will call the method defined in *url_navigator*, with *page_url* as the first parameter.

Returns The instance where navigate() was called from.

Return type *Page*

Example

```
>>> from splinter import Browser
>>> from stere import Page
>>>
>>>
>>> class Home(Page):
>>>     def __init__(self):
>>>         self.page_url = 'https://en.wikipedia.org/'
>>>
>>>
>>> Stere.browser = Browser()
>>> Stere.url_navigator = 'visit'
>>>
>>> home_page = Home()
>>> home_page.navigate()
```

1.2.1 Using Page as a Context Manager

Page contains __enter__() and __exit__() methods. This allows any page to be used as a Context Manager.

Example:

```
from pages import Home

with Home() as p:
    p.login_button.click()
```

1.3 Fields

1.3.1 Field

1.3.2 Root

1.3.3 Text

1.3.4 Performer method

A Field can have a single method be designated as a performer. This method will be called when the Field is inside an Area and that Area's perform() method is called.

For example, Input's performer is the fill() method, and Button's performer is the click() method. Given the following Area:

```
search = Area(
    query=Input('id', 'xsearch'),
    submit=Button('id', 'xsubmit'),
)
```

and the following script:

```
search.perform('Orange')
```

When `search.perform('Orange')` is called, `query.fill('Orange')` is called, followed by `submit.click()`.

See the documentation for Area for more details.

1.3.5 Calling the performer method explicitly

The performer method is available as `Field.perform()`. Calling it will run the performer method, but they are not aliases.

No matter what the return value of the performer method is, the return value from calling Field.perform() will always be the Field.returns attribute.

Using the splinter Button Field as an example, the only difference between *Button.click()* and *Button.perform()* is that perform will return the object set in the *Field.returns* attribute. See Returning Objects for more details.

1.3.6 Calling the performer method implicitly

When a page instance is called directly, the *perform()* method will be executed.

The following code will produce the same results:

```
button = Button()
button.perform()
```

```
button = Button()
button()
```

1.3.7 Subclassing Field

Field can be subclassed to suit your own requirements.

If the __init__() method is overwritten, make sure to call super() before your own code.

If your class needs specific behaviour when interacting with Areas, it must be wrapped with the @stere_performer decorator to specify a performer method.

When creating a new type of Field, the stere_performer class decorator should used to assign a performer method.

1.3.8 Field Decorators

1.4 Splinter Integration

Stere contains Fields designed specifically for when Splinter is connected. Each implements a specific performer method.

All Fields designed for Splinter also inherit the following convenience methods:

Example:

```
class Inventory(Page):
    def __init__(self):
        self.price = Link('css', '.priceLink')


assert Inventory().price.is_present(wait_time=6)
```

1.4.1 Fields

Button

class stere.fields.**Button**

Convenience Class on top of Field, it implements *click()* as its performer.

Checkbox

class stere.fields.**Checkbox**

By default, the Checkbox field works against HTML inputs with type="checkbox".

Can be initialized with the *default_checked* argument. If True, the Field assumes the checkbox's default state is checked.

It implements *opposite()* as its performer.

Dropdown

class stere.fields.**Dropdown**

By default, the Dropdown field works against HTML Dropdowns. However, it's possible to extend Dropdown to work with whatever implementation of a CSS Dropdown you need.

It implements *select()* as its performer.

The *option* argument can be provided to override the default implementation. This argument expects a Field. The Field should be the individual options in the dropdown you wish to target.

```
self.languages = Dropdown('id', 'langDrop', option=Button('xpath', '/h4/a/strong
↪'))
```

Input

class stere.fields.**Input**
A simple wrapper over Field, it implements *fill()* as its performer.

The *default_value* argument can be provided, which will be used if fill() is called with no arguments.

```
self.quantity = Dropdown('id', 'qty', default_value='555')
```

Link

class stere.fields.**Link**
A simple wrapper over Field, it implements *click()* as its performer.

Money

class stere.fields.**Money**
Money has methods for handling Fields where the text is a form of currency.

1.4.2 Locator Strategies

These represent the way a locator can be searched for.

By default, the strategies available with Splinter are:

- css
- xpath
- tag
- name
- text
- id
- value

These strategies can be overridden with a custom strategy (ie: You can create a custom css strategy with different behaviour).

1.4.3 Custom Locator Strategies

Custom strategies can be defined using the *@strategy* decorator on top of a Class.

Any class can be decorated with @strategy, as long as the _find_all and _find_all_in_parent methods are implemented.

In the following example, the 'data-test-id' strategy is defined. It wraps Splinter's find_by_xpath method to simplify the locator required on the Page Object.

```
from stere.strategy import strategy


@strategy('data-test-id')
class FindByDataTestId():
    def _find_all(self):
        """Find from page root."""
        return self.browser.find_by_xpath(f'.//*[@data-test-id="{self.locator}"]')

    def _find_all_in_parent(self):
        """Find from inside parent element."""
        return self.parent_locator.find_by_xpath(f'.//*[@data-test-id="{self.locator}
↪"]')
```

With this implemented, Fields can now be defined like so:

```
my_button = Button('data-test-id', 'MyButton')
```

Support for data-* attributes is also available via the *add_data_star_strategy* function:

```
from stere.strategy import add_data_star_strategy


add_data_star_strategy('data-test-id')
```

This will automatically add the desired data-* attribute to the valid Splinter strategies.

1.5 Appium Integration

Stere contains Fields designed specifically for when Appium is connected. Each implements a specific performer method.

1.5.1 Fields

Button

class stere.fields.**Button**

Convenience Class on top of Field, it implements *click()* as its performer.

Input

class stere.fields.**Input**

A simple wrapper over Field, it implements *send_keys()* as its performer.

The *default_value* argument can be provided, which will be used if send_keys() is called with no arguments.

```
self.quantity = Dropdown('id', 'qty', default_value='555')
```

Fills the element with value.

1.5.2 Locator Strategies

These represent the way a locator will be searched for.

By default, the strategies available are:

- accessibility_id
- android_uiautomator
- ios_class_chain
- ios_predicate
- ios_uiautomation

These strategies can be overridden with a custom strategy (ie: You can create a custom accessibility_id strategy with different behaviour).

1.6 Areas

Areas represent groupings of Fields on a Page.

The following Area objects are available:

- Area: A non-hierarchical, unique group of Fields.
- RepeatingArea: A hierarchical, non-unique group of Areas. They require a Root Field.

1.6.1 Reusing Areas

Sometimes an identical Area may be present on multiple pages. Areas do not need to be created inside a page object, they can be created outside and then called from inside a page.

```
header = Area(
    ...
)

class Items(Page):
    def __init__(self, *args, **kwargs):
        self.header = header
```

1.6.2 Subclassing Areas

If an Area appears on many pages and requires many custom methods, it may be better to subclass the Area instead of embedding the methods in the Page Object:

```
class Header(Area):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def my_custom_method(self, *args, **kwargs):
        ...


class Main(Page):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, *args, **kwargs):
        self.header = Header()


class Other(Page):
    def __init__(self, *args, **kwargs):
        self.header = Header()
```

1.7 Repeating

1.8 Workflows

When working with an Area that has multiple possible routes, there may be Fields which you do not want the .perform() method to call under certain circumstances.

Take the following example Page Object:

```
class AddSomething(Page):
    def __init__(self):
        self.form = Area(
            item_name=Input('id', 'itemName'),
            item_quantity=Input('id', 'itemQty'),
            save=Button('id', 'saveButton'),
            cancel=Button('id', 'cancelButton')
        )
```

Calling *AddSomething().form.perform()* would cause the save button and then the cancel button to be acted on.

In these sorts of cases, Workflows can be used to manage which Fields are called.

```
class AddSomething(Page):
    def __init__(self):
        self.form = Area(
            item_name=Input('id', 'itemName', workflows=["success", "failure"]),
            item_quantity=Input('id', 'itemQty', workflows=["success", "failure"]),
            save=Button('id', 'saveButton', workflows=["success"]),
            cancel=Button('id', 'cancelButton', workflows=["failure"])
        )
```

Calling *AddSomething().form.workflow("success").perform()* will ensure that only Fields with a matching workflow are called.

1.9 Fields Returning Objects

Fields take an optional *returns* argument. This can be any object. When the Field's perform() method is called, this object will be returned.

This can be used to return another Page Object.

```
class Navigation(Page):
    def __init__(self):
        self.goto_settings = Button('id', 'settingsLink', returns=NextPage())
```

```
def test_navigation():
    page = Navigation()
    next_page = page.goto_settings.perform()
```

1.9.1 Fields inside an Area

When a Field is inside an Area and has the returns argument set, only the object for the last Field in the Area will be returned when *Area.perform()* is called.

```
class Address(Page):
    def __init__(self):
        self.form = Area(
            address=Input('id', 'formAddress'),
            city=Input('id', 'formCity', returns=FooPage()),
            submit=Button('id', 'formsubmit', returns=NextPage()),
        )
```

```
def test_address_form():
    page = Address()
    next_page = page.form.perform()
```

1.10 Best Practices

A highly opinionated guide. Ignore at your own peril.

1.10.1 Favour adding methods to Fields and Areas over Page Objects

If a new method is acting on a specific Field, subclass the Field and add the method there instead of adding the method to the Page Object.

Wrong:

```
class Inventory(Page):
    def __init__(self):
        self.medals = Field('id', 'medals')

    def count_the_medals(self):
        return len(self.medals.find())


def test_you_got_the_medals():
    inventory = Inventory()
    assert 3 == inventory.count_the_medals()
```

Right:

```
class Medals(Field):
    def count(self):
        return len(self.find())
```

(continues on next page)

(continued from previous page)

```
class Inventory(Page):
    def __init__(self):
        self.medals = Medals('id', 'medals')


def test_you_got_the_medals():
    inventory = Inventory()
    assert 3 == inventory.medals.count()
```

Explanation:

Even if a Field or Area initially appears on only one page, subclassing will lead to code that is more easily reused and/or moved.

In this example, inventory.count_the_medals() may look easier to read than inventory.medals.count(). However, creating methods with long names and specific verbiage makes your Page Objects less predictable and more prone to inconsistency.

1.10.2 Favour page composition over inheritance

When building Page Objects for something with many reused pieces (such as a settings menu) don't build an abstract base Page Object. Build each component separately and call them in Page Objects that reflect the application.

Inheritance:

```
class BaseSettings(Page):
    def __init__(self):
        self.settings_menu = Area(...)


class SpecificSettings(BaseSettings):
    def __init__(self):
        super().__init__()
```

Composition:

```
from .another_module import settings_menu

class SpecificSettings(Page):
    def __init__(self):
        self.menu = settings_menu
```

Explanation:

Doing so maintains the benefits of reusing code, but prevents the creation of Page Objects that don't reflect actual pages in an application.

Creating abstract Page Objects to inherit from can make it confusing as to what Fields are available on a page.

1.10.3 Naming Fields

Describing the Field VS Describing the Field's Action

When naming a field instance, the choice is usually between a description of the field or a description of what the field does:

Describing the Field:

```
class Navigation(Page):
    def __init__(self):
        self.settings_button = Button('id', 'settingsLink')
```

Describing the Action:

```
class Navigation(Page):
    def __init__(self):
        self.goto_settings = Button('id', 'settingsLink')
```

At the outset, either option can seem appropriate. Consider the usage inside a test:

```
nav_page = Navigation()
nav_page.settings_button.click()
```

VS

```
nav_page = Navigation()
nav_page.goto_settings.click()
```

However, consider what happens when a Field returns a Page:

```
class Navigation(Page):
    def __init__(self):
        self.settings_page = Button('id', 'settingsLink', returns=NextPage())
```

```
class Navigation(Page):
    def __init__(self):
        self.goto_settings = Button('id', 'settingsLink', returns=NextPage())
```

```
nav_page = Navigation()
settings_page = nav_page.settings_button.perform()
```

```
nav_page = Navigation()
settings_page = nav_page.goto_settings.perform()
```

Or, calling the perform method implicitly:

```
nav_page = Navigation()
settings_page = nav_page.settings_button()
```

```
nav_page = Navigation()
settings_page = nav_page.goto_settings()
```

In the end, naming Fields will depend on what they do and how your tests use them.

1.10.4 Single blank line when changing page object

Wrong:

```
def test_the_widgets():
    knicknacks = Knicknacks()
    knicknacks.menu.gadgets.click()
```

(continues on next page)

(continued from previous page)

```
    knicknacks.gadgets.click()
    gadgets = Gadgets()
    gadgets.navigate()

    gadgets.add_widgets.click()
    gadgets.add_sprocket.click()
```

Right:

```
def test_the_widgets():
    knicknacks = Knicknacks()
    knicknacks.menu.gadgets.click()
    knicknacks.gadgets.click()

    gadgets = Gadgets()
    gadgets.navigate()
    gadgets.add_widgets.click()
    gadgets.add_sprocket.click()
```

Explanation:

Changing pages usually indicates a navigation action. Using a consistent line break style visually helps to indicate the steps of a test.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index

N

navigate() (*stere.Page method*), 3

P

Page (*class in stere*), 2
page_url (*stere.Page attribute*), 3

S

stere.fields.Button (*built-in class*), 5, 7
stere.fields.Checkbox (*built-in class*), 5
stere.fields.Dropdown (*built-in class*), 5
stere.fields.Input (*built-in class*), 6, 7
stere.fields.Link (*built-in class*), 6
stere.fields.Money (*built-in class*), 6