pypom_form Documentation

Release 0.1

Tierra QA team

Sep 14, 2017

Contents

`pypom_form` is a PyPOM based package that provides declarative schema based form interaction for page objects.

pypom_form aims to improve the developer experience for UI, E2E test automation when you have to interact with page object containing forms thanks to declarative schema models.

If you come from past experience with frameworks like SQLAlchemy, Dexterity (Plone) or the old Archetypes (Plone) you should be already familiar with this pattern: you simply define a model with a schema and you will be able to interact with your model saving or retrieving data. Same happens with pypom_form where the model is the page.

pypom_form it is internally based on:

- PyPOM
- colander
- Splinter

CHAPTER 1

How does it work?

Whith pypom_form you have just to:

- instanciate a page object instance whose class inherits from BaseFormPage provided by pypom_form
- declare the schema model

And you will be ready for interacting with your page driving the browser with your form just typing:

```
page.title = 'the title'
page.title
```

assuming that you have a `title` field in your form.

Main concepts

You might think about the `schema` concept as a set of named attributes (`fields`) that will be available on the `model` as regular properties.

Each `field` on the schema is defined with a `type` (eg: string, int, float, datetime, date, bool, etc) that defines the data type for the given field on the application domain level.

Fields has a reference to a `widget` defined imperatively or assigned by default depending on the field type. The inner implementation of widgets provided by pypom_form is based on PyPOM's Regions, so `widget regions` wraps and manage a DOM containing the widget.

Basically the widget translates data from the applicative domain to the browser domain and vice versa through serialization and deserialization.

You might thing about a widget as how you have to driver your browser when you set `True` to a boolean property or get the actual value on the form: basically it depends on the widget implementation. For example you might have a checkbox, yes/no radio buttons or combo select, etc and if you want to set `True` the way you drive the browser changes. Same for date widgets and so on.

You might have to deal with complex widgets too like:

- reference widgets (eg: hierarchical content navigation with search, filtering, etc)
- advanced multi selection widgets
- dictionary widgets (key value mapping)
- etc

For example, assuming you are dealing with a pretend advanced single selection choice field you can access to advanced logics provided by the `widget region`:

```
page.getWidgetRegion('state').filter('virg').select('Virginia')
```

or access to validation error messages, label text, etc.

Why pypom_form

Obviously you can drive your browser in automated tests with plain selenium/splinter or with a traditional plain page object model pattern but with pypom_form you have the following advantages:

- write once and reusable approach, very useful if you are testing CMS framework
- separation of concerns for page and widget logics
- declarative schema approach
- reusable schema and widgets, no code repetition
- widgets can be shared with other projects using pypom_form
- simple API based on auto generated getter and setters
- interact with advanced widget logics thanks to PyPOM based region widgets
- widget isolation. All element queries run against the root region, not the page root
- simpler input elements selectors, they are relative to the region widget root
- schema forms improves how you document page containing forms (attributes names, type, widgets, allowed vocabularies, etc). All you need to know is defined at schema level with the whole picture available at a glance
- reuse of existing schemas if you are going to test a colander/deform based application (probably you are testing a Pylons Pyramid Python based web application)
- page and schema inheritance supported as well
- easy test multi skin web applications with same data model, same or different selectors or widget types. So you can reuse all your page object classes as they are defined, it changes only the schema widget selector adn widget types
- widget regions are PyPOM regions, so if you want to access inner elements inside the widget container the resulting selectors will be simpler because they are relative to the widget region root. Also sub/nested regions or dynamic regions are supperted as well
- interact with your model with applicative domain data instead of browser domain data. It is more simple and easy to manage Python data (for example you set 12.9 instead of '12.9', same for datetimes values like `datetime.now()`)
- supports chained calls like `page.set('title', 'the title')`
- supports bulk field updates considering the order defined at schema level via `page.update(**values)`
- don't reinvent the wheel. It is based on existing and widely used components like the plain PyPOM or Colander libraries

- same user experience if you are already familiar with schema declarative models like `SQLAlchemy`, `Archetypes` (Plone), `Dexterity` (Plone) or form libraries like `deform`
- since widget implementation is based on regions, you can simply perform a `page.name = "the name"` on page load instead of having to call a wait method before setting the value: the widget is able to wait for the widget load before getting or setting data
- page objects classes more simple, with less code, more standard even if different test engineers will implement page form logics: there is a structural pattern

In addition:

- 100% test coverage
- both Python 2 and 3 support
- supports Splinter drivers (Selenium support not yet available)
- pytest setup ready thanks to `pytest-splinter`

CHAPTER 2

Code samples

The following code samples assumes that there is a navigation fixture providing the page instance built with a Splinter driver but you can build by yourself a page instance following the PyPOM documentation:

- http://pypom.readthedocs.io/en/latest/

Schema definition:

```
import colander

from pypom_form.form import BaseFormPage


class BaseEditSchema(colander.MappingSchema):
    """ This is the base edit mapping common for all pages """

    name = colander.SchemaNode(
        colander.String(),
        selector=('id', 'name-widget'),
    )


class BaseEditPage(BaseFormPage):
    """ This is the base edit class """

    schema_factory = BaseEditSchema
```

And assuming you have a page instance you can interact with the above page just setting an attribute:

```
@pytest_bdd.when(pytest_bdd.parsers.parse(
    'I set {name} as name field'))
def fill_name(navigation, name):
    page = navigation.page
    page.name = name
```

You can also define other pages with extended schema, for example an integer type:

```
class AnotherPageEditSchema(BaseEditSchema):

    duration = colander.SchemaNode(
        colander.Int(),
        missing=0,
        selector=('id',
                  'duration-widget'),
        validator=colander.Range(0, 9999))
```

but you can create also field types like `colander.Bool` or any other colander supported types.

And the test:

```
@pytest_bdd.when(pytest_bdd.parsers.cfparse(
    'I set {duration:Number} as Alarm duration',
    extra_types=dict(Number=int)))
def fill_alarm_duration(navigation, duration):
    page = navigation.page
    page.duration = duration
```

You might notice that in the above example you are setting an integer duration and not a string. So you can perform `page.duration += 10` for example.

You can also define custom widgets on fields if the default implementation does not match the one available on your application (for example a non standard checkbox for a boolean widget), for example a pretend `MyBooleanWidget`:

```
mybool = colander.SchemaNode(
    colander.Bool(),
    missing=False,
    selector=(
        'id',
        'mybool-widget'
    ),
    pypom_widget=MyBoolWidget()
)
```

Also chained calls are supported (eg: set the title, perform the pretend submit method and then set a boolean):

```
page.set('title', 'the title'). \
    .submit(). \
    .set('mybool', False)
```

or bulk updates. All changes occurs following the fields order at schema level:

```
page.update(**{'title': 'the title', 'mybool': True})
```

The `update` or `raw_update` can be used in test preconditions creation. Assuming you have a generic given step with parametrized with a complex configuration you can pass the raw json data and the `raw_update` will take care about the data conversion from browser model (eg: string) to the page model (strings, integers, datetimes, etc):

```
@pytest_bdd.given(pytest_bdd.parsers.cfparse(
    'I have a CAN bus protocol configured with:\n{raw_conf:json}',
    extra_types=dict(json=json.loads)))
def create_can_protocol(navigation, base_url, raw_conf):
    """ create a can protocol
    """

    navigation. \
```

```
        visit_page('CANBusProtocolsPage'). \
        wait_for_full_spinner(). \
        click_add(). \
        raw_update(**raw_conf). \
        save(). \
        wait_for_success_pop_up_appears(). \
        click_on_ok_pop_up()
```

assuming that the `raw_conf` is specified in json format in the `.feature` file, for example:

```
@UI @edit @CANBusParameter
Scenario: Add a CAN bus parameter
  Given I am logged in as Administrator
  And I have a CAN bus protocol configured with:
      {"name": "The name",
       "baudrate": "250",
       ...
      }
  And ...
```

As you can see in the above code examples there is no need to perform wait calls before interacting with a form on page load because each widget is able to wait until its controlled input element is ready. Wait logics are already defined on widget level and you can override them.

Index

pypom_form

`pypom_form` is a PyPOM based package that provides declarative schema based form interaction for page objects.

pypom_form aims to improve the developer experience for UI, E2E test automation when you have to interact with page object containing forms thanks to declarative schema models.

If you come from past experience with frameworks like SQLAlchemy, Dexterity (Plone) or the old Archetypes (Plone) you should be already familiar with this pattern: you simply define a model with a schema and you will be able to interact with your model saving or retrieving data. Same happens with pypom_form where the model is the page.

pypom_form it is internally based on:

- PyPOM
- colander
- Splinter

How does it work?

Whith pypom_form you have just to:

- instanciate a page object instance whose class inherits from BaseFormPage provided by pypom_form
- declare the schema model

And you will be ready for interacting with your page driving the browser with your form just typing:

```
page.title = 'the title'
page.title
```

assuming that you have a `title` field in your form.

Main concepts

You might think about the `schema` concept as a set of named attributes (`fields`) that will be available on the `model` as regular properties.

Each `field` on the schema is defined with a `type` (eg: string, int, float, datetime, date, bool, etc) that defines the data type for the given field on the application domain level.

Fields has a reference to a `widget` defined imperatively or assigned by default depending on the field type. The inner implementation of widgets provided by pypom_form is based on PyPOM's Regions, so `widget regions` wraps and manage a DOM containing the widget.

Basically the widget translates data from the applicative domain to the browser domain and vice versa through serialization and deserialization.

You might thing about a widget as how you have to driver your browser when you set `True` to a boolean property or get the actual value on the form: basically it depends on the widget implementation. For example you might have a checkbox, yes/no radio buttons or combo select, etc and if you want to set `True` the way you drive the browser changes. Same for date widgets and so on.

You might have to deal with complex widgets too like:

- reference widgets (eg: hierarchical content navigation with search, filtering, etc)
- advanced multi selection widgets
- dictionary widgets (key value mapping)
- etc

For example, assuming you are dealing with a pretend advanced single selection choice field you can access to advanced logics provided by the `widget region`:

```
page.getWidgetRegion('state').filter('virg').select('Virginia')
```

or access to validation error messages, label text, etc.

Why pypom_form

Obviously you can drive your browser in automated tests with plain selenium/splinter or with a traditional plain page object model pattern but with pypom_form you have the following advantages:

- write once and reusable approach, very useful if you are testing CMS framework
- separation of concerns for page and widget logics
- declarative schema approach
- reusable schema and widgets, no code repetition
- widgets can be shared with other projects using pypom_form
- simple API based on auto generated getter and setters
- interact with advanced widget logics thanks to PyPOM based region widgets
- widget isolation. All element queries run against the root region, not the page root
- simpler input elements selectors, they are relative to the region widget root

- schema forms improves how you document page containing forms (attributes names, type, widgets, allowed vocabularies, etc). All you need to know is defined at schema level with the whole picture available at a glance
- reuse of existing schemas if you are going to test a colander/deform based application (probably you are testing a Pylons Pyramid Python based web application)
- page and schema inheritance supported as well
- easy test multi skin web applications with same data model, same or different selectors or widget types. So you can reuse all your page object classes as they are defined, it changes only the schema widget selector adn widget types
- widget regions are PyPOM regions, so if you want to access inner elements inside the widget container the resulting selectors will be simpler because they are relative to the widget region root. Also sub/nested regions or dynamic regions are supperted as well
- interact with your model with applicative domain data instead of browser domain data. It is more simple and easy to manage Python data (for example you set 12.9 instead of '12.9', same for datetimes values like `datetime.now()`)
- supports chained calls like `page.set('title', 'the title')`
- supports bulk field updates considering the order defined at schema level via `page.update(**values)`
- don't reinvent the wheel. It is based on existing and widely used components like the plain PyPOM or Colander libraries
- same user experience if you are already familiar with schema declarative models like `SQLAlchemy`, `Archetypes` (Plone), `Dexterity` (Plone) or form libraries like `deform`
- since widget implementation is based on regions, you can simply perform a `page.name = "the name"` on page load instead of having to call a wait method before setting the value: the widget is able to wait for the widget load before getting or setting data
- page objects classes more simple, with less code, more standard even if different test engineers will implement page form logics: there is a structural pattern

In addition:

- 100% test coverage
- both Python 2 and 3 support
- supports Splinter drivers (Selenium support not yet available)
- pytest setup ready thanks to `pytest-splinter`

Code samples

The following code samples assumes that there is a navigation fixture providing the page instance built with a Splinter driver but you can build by yourself a page instance following the PyPOM documentation:

- http://pypom.readthedocs.io/en/latest/

Schema definition:

```
import colander

from pypom_form.form import BaseFormPage


class BaseEditSchema(colander.MappingSchema):
    """ This is the base edit mapping common for all pages """
```

```
    name = colander.SchemaNode(
        colander.String(),
        selector=('id', 'name-widget'),
    )


class BaseEditPage(BaseFormPage):
    """ This is the base edit class """

    schema_factory = BaseEditSchema
```

And assuming you have a page instance you can interact with the above page just setting an attribute:

```
@pytest_bdd.when(pytest_bdd.parsers.parse(
    'I set {name} as name field'))
def fill_name(navigation, name):
    page = navigation.page
    page.name = name
```

You can also define other pages with extended schema, for example an integer type:

```
class AnotherPageEditSchema(BaseEditSchema):

    duration = colander.SchemaNode(
        colander.Int(),
        missing=0,
        selector=('id',
                  'duration-widget'),
        validator=colander.Range(0, 9999))
```

but you can create also field types like `colander.Bool` or any other colander supported types.

And the test:

```
@pytest_bdd.when(pytest_bdd.parsers.cfparse(
    'I set {duration:Number} as Alarm duration',
    extra_types=dict(Number=int)))
def fill_alarm_duration(navigation, duration):
    page = navigation.page
    page.duration = duration
```

You might notice that in the above example you are setting an integer duration and not a string. So you can perform `page.duration += 10` for example.

You can also define custom widgets on fields if the default implementation does not match the one available on your application (for example a non standard checkbox for a boolean widget), for example a pretend `MyBooleanWidget`:

```
mybool = colander.SchemaNode(
    colander.Bool(),
    missing=False,
    selector=(
        'id',
        'mybool-widget'
    ),
    pypom_widget=MyBoolWidget()
)
```

Also chained calls are supported (eg: set the title, perform the pretend submit method and then set a boolean):

```
page.set('title', 'the title'). \
    .submit(). \
    .set('mybool', False)
```

or bulk updates. All changes occurs following the fields order at schema level:

```
page.update(**{'title': 'the title', 'mybool': True})
```

The `update` or `raw_update` can be used in test preconditions creation. Assuming you have a generic given step with parametrized with a complex configuration you can pass the raw json data and the `raw_update` will take care about the data conversion from browser model (eg: string) to the page model (strings, integers, datetimes, etc):

```
@pytest_bdd.given(pytest_bdd.parsers.cfparse(
    'I have a CAN bus protocol configured with:\n{raw_conf:json}',
    extra_types=dict(json=json.loads)))
def create_can_protocol(navigation, base_url, raw_conf):
    """ create a can protocol
    """

    navigation. \
        visit_page('CANBusProtocolsPage'). \
        wait_for_full_spinner(). \
        click_add(). \
        raw_update(**raw_conf). \
        save(). \
        wait_for_success_pop_up_appears(). \
        click_on_ok_pop_up()
```

assuming that the `raw_conf` is specified in json format in the `.feature` file, for example:

```
@UI @edit @CANBusParameter
Scenario: Add a CAN bus parameter
  Given I am logged in as Administrator
  And I have a CAN bus protocol configured with:
      {"name": "The name",
       "baudrate": "250",
       ...
      }
  And ...
```

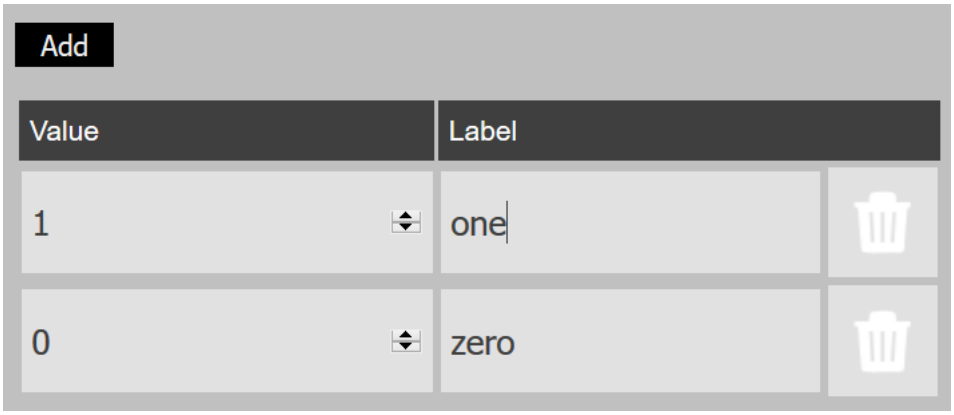
As you can see in the above code examples there is no need to perform wait calls before interacting with a form on page load because each widget is able to wait until its controlled input element is ready. Wait logics are already defined on widget level and you can override them.

Advanced

Here you can see how to create a custom widget or create your own colander types with validators.

Encoded values widget

Let's pretend we have to manage a simple key-value widget: a sort of dictionary like structure where both keys and values are string types like shown in the following picture.



Final interaction with the widget

For example you might have a `I set the encoded values field with:` BDD statement like the following one:

```
@UI @edit @CANBusFamily @encoded
Scenario: Add a CAN bus family encoded
  Given I am logged in as Administrator
  And I am on the CANBusFamiliesPage page
  When I click on the Add button
  And I fill in the name of the form
  And I set the encoded values field with:
      {"0": "zero", "1": "one"}
  And I submit the form
  Then a success popup message appears
```

implemented just with:

```
@pytest_bdd.when(pytest_bdd.parsers.cfparse(
    'I set the encoded values field with:\n{encoded_values:json}',
    extra_types=dict(json=json.loads)))
def set_encoded_values(navigation, encoded_values):
    """ Set encoded values """
    navigation.page.encoded_values = encoded_values
```

On the page instance you can simply set the values you want to apply in one shot:

```
page.encoded_values = {'company1': 'Company ONE', 'company2': 'Company TWO'}
...
```

or interact step by step thanks to the widget region:

```
page.getWidgetRegion('encoded_values').click_add()
...
```

Final page form setup configuration

You can add a new row, delete a row, add a key and a value for each row. If you want you can also create some validators and contraints to your values.

On the page side we need a schema and a page or region object inheriting from the base classes provided by `pypom_form` with a dictionary like colander type (`Mapping`) and a custom widget `EncodedValuesWidget`:

```
class MyEditPageSchema(BaseEditSchema):

    encoded_values = colander.SchemaNode(
        colander.Mapping(unknown='preserve'),
        selector=('css', '#metric-tabs-2'),
        pypom_widget=EncodedValuesWidget(),
        )
    )


class MyEditPage(BaseEditPage):

    schema_factory = MyEditPageSchema
```

Widget implementation

And now let's see our pretend custom widget implementation. The widget itself is based on:

- a widget `EncodedValuesWidget`, it will let you interact with the input elements if you want to set a value or read the actual value on the browser (eg: `{'company1': 'Company 1'}`). The widget is internally based on a widget region called `EncodedValuesWidgetRegion`
- a widget region `EncodedValuesWidgetRegion`, providing the main logics used by the widget itself and available on the page if you want to interact step by step instead of assign a whole dictionary like. For example you can add a new row, delete it, change a single value or key and so on. So you can interact with the `EncodedValuesWidgetRegion` like a dictionary: set or get values, iterate on them, etc. The widget region inner logics are demanded to subregions `EncodedValueRegion` (dynamic regions) for each row. The subregions controls how to set a key, a value, delete the item row.
- the inner element is the region `EncodedValueRegion`. They are instanciated dynamically by the widget region and they provides a schema_factory containing a `key` and a `value` string properties you can interact with

Each key-value pair Here you can see how to create a custom widget, for example a dictionary like widget with a key and a value (encoded values widget) or create your own

Let's see the resulting code:

```
import colander
from pypom_form.form import BaseFormRegion
from pypom_form.widgets import (
    BaseWidget,
    BaseWidgetRegion,
)


class EncodedValueRegionSchema(colander.MappingSchema):
    """ EncodedValueRegion schema for encoded values """

    key = colander.SchemaNode(
        colander.String(),
        selector=('css', 'input[type="number"]'),
    )

    value = colander.SchemaNode(
```

```
        colander.String(),
        selector=('css', 'input[type="text"]'),
    )


class EncodedValueRegion(BaseFormRegion):
    """ Single encoded value region with key, value and delete button.

        This is a subregion returned dynamically by
        the EncodedValuesWidgetRegion for each key-value pair.

        Each subregion exposes a key and a value.


        You can delete subregion instance through the ``delete`` method,
    """

    schema_factory = EncodedValueRegionSchema

    DELETE_SELECTOR = ('css', '.administration_list_delete')

    def delete(self):
        """ Delete region """
        self.find_element(*self.DELETE_SELECTOR).click()


class EncodedValuesWidgetRegion(BaseWidgetRegion):
    """ Encoded values widget region
        You can interact with your page using dictionary-like
        operations.

        >>> region = page.getWidgetRegion('encoded_values')
        >>> region['0'] = 'ZERO'
        >>> region['0']

        You can also iterate on subregions for each key-value pair:

        >>> region.encoded_value_regions[0].key = '1'
        >>> region.encoded_value_regions[0].value = 'one'

        Or add a new key-value pair without interact:

        >>> subregion = region.click_add()
        >>> subregion.key = '1'
        >>> subregion.value = 'ONE'

        Access to one key-value pair and interact with it:

        >>> region.encoded_value_regions[0].value = 'one'

        Or delete a mapping:

        >>> del region['0']
    """

    REGIONS_ROW_SELECTOR = ('css', 'tbody > tr')
    ADD_BUTTON_SELECTOR = ('css', '.add_button')
```

```
    def click_add(self):
        """ Click add and returns a subregion """
        previous_len = len(self)
        self.find_element(*self.ADD_BUTTON_SELECTOR).click()
        self.wait.until(lambda s: len(self) == previous_len+1)
        return self.encoded_value_regions[0]

    @property
    def encoded_value_regions(self):
        """ Encoded values regions"""
        return [EncodedValueRegion(self, root=root) for root in
                self.find_elements(*self.REGIONS_ROW_SELECTOR)]

    def clear(self):
        """ clear all values """
        for region in self.encoded_value_regions:
            region.delete()

    def copy(self):
        values = {}
        for key, value in self.items():
            values[key] = value
        return values

    def items(self):
        return [(key, self[key]) for key in self]

    def update(self, **values):
        for key, value in values.items():
            self[key] = value

    def __getitem__(self, key):
        for region in self.encoded_value_regions:
            if region.key == key:
                return region.value
        raise KeyError

    def __setitem__(self, key, value):
        regions = [item for item in self.encoded_value_regions
                   if item.key == key]
        if not regions:
            regions = [self.click_add()]
        region = regions[0]
        region.value = value
        if region.key != key:
            region.key = key

    def __delitem__(self, key):
        self[key].delete()

    def __contains__(self, key):
        for key_item in self:
            if key_item == key:
                return True
        return False

    def __len__(self):
        return len(self.encoded_value_regions)
```

```
    def __iter__(self):
        for region in self.encoded_value_regions:
            yield region.key

    def __repr__(self):
        return "%r(%r)" % (self.__class__, self.copy())


class EncodedValuesWidget(BaseWidget):
    """ This is the EncodedValuesWidget """
    region_class = EncodedValuesWidgetRegion

    def getter_factory(self):
        def _getter(page):
            reg = self.getWidgetRegion(page)
            value = reg.copy()
            return self.field.deserialize(value)
        return _getter

    def setter_factory(self):
        def _setter(page, value):
            reg = self.getWidgetRegion(page)
            reg.clear()
            value = self.field.serialize(value)
            reg.update(**value)
        return _setter
```

Final considerations

Now you have a dictionary like edit widget reusable across different page objects sharing the same data structures powered by regions and subregions. The widget interaction on page objects empowered by `pypom_form` widgets is as easy as dealing with a Python dictionary but you can also perform custom interactions using the widget region API.

So thanks to `pypom_form` widgets you can deal with rich UI widgets hiding the complexity making things easy for a great development and testing experience.

Extending Colander

We won't cover how to add your own custom colander types or validators, instead we'll address you to the Colander documentation online:

- http://docs.pylonsproject.org/projects/colander/en/latest/extending.html

API reference

Here you can see the technical documentation.

class `pypom_form.widgets.`**`BaseWidgetRegion`**(*page*, *root=None*)

This is the base widget region associated to a widget.

You can lookup the base widget region from a page or region form with `page.getWidgetRegion('field_name')` in order to enable advanced widget interactions.

For example you can get the field label, the help text, validation errors or if you have a complex widget you can expose the widget API. For example a multi selection widget with a filter feature or a reference field with a navigation popup with all the portal contents.

This way you are not only able to set or get field values but you can incapsulate complex logics in the widget region and expose them on the page or region form ready to be used

get_label()
Return the widget label

get_help()
Return the widget text help

get_validation_errors()
Return the widget validation error

wait_for_region_to_load()
Wait for the page region to load.

class pypom_form.widgets.**BaseWidget**(*field=None*, *region_class=None*, *options={}*)
This is the base widget. It is not intended to be used itself but you can use it as base class for your own widgets.

You can associate widgets on schema level or the form metaclass will associate a fallback depending on the schema type if `pypom_widget` is missing.

You can provide your own `region_class` on the schema definition of overriding the `region_class` class attribute in your own widget implementations.

region_class
alias of *BaseWidgetRegion*

input_selector
Returns the input selector inside the field container.

Most or times is a tuple with `('tag', 'input')` but it might change depending on the widget type.

It's up to you providing the input selector that matches your input type.

get_input_element(*page*)
Return the input element.

If you provide a selector for the container of the input element, it will return the input element itself (preferred way if you want to use advanced widget features).

Otherwise the region root will be returned

getWidgetRegion(*page*)
Returns a dynamic widget region containing the root selector.

This is an internal method used by the page or region metaclass in order to be able to expose the widget region simply calling `page.getWidgetRegion()`.

It also sets a reference to the widget itself containing the widget options on the widget region.

getter_factory()
Returns a generated method to be attached on the PyPOM page or region.

This is an internal method used by the page or region metaclass in order to be able to generate the getter method for the field.

setter_factory()
Returns a generated method to be attached on the PyPOM page or region.

This is an internal method used by the page or region metaclass in order to be able to generate the getter method for the field.

serialize(*value*)
Serialize value. This method returns the value serialized (from model value to browser or widget internal representation).

deserialize(*value*)
Deserialize value. This method returns the value serialized (from browser or internal widget representation to model).

class pypom_form.widgets.**StringWidget**(*field=None*, *region_class=None*, *options={}*)
String widget

class pypom_form.widgets.**TextAreaWidget**(*field=None*, *region_class=None*, *options={}*)
TextArea widget

class pypom_form.widgets.**CheckboxWidget**(*field=None*, *region_class=None*, *options={}*)
Checkbox widget

Contributors

- Full contributors list

Changelog

0.2.3 (2017-09-14)

- Now you can override automatically generated methods and invoke `super` inside them

0.2.2 (2017-07-03)

- Moved `test_fields.py` to right location.
- Don't override `update` and `raw_update` methods on pages and regions if already exist.

0.2.1 (2017-06-01)

- Update email project

0.2.0 (2017-01-24)

Features

- Added `serialize` and `deserialize` methods on widgets base implementation for advanced usage. This way you can implement complex/composed widgets more easily just overriding the above methods (eg: perform an intermediate conversion from page model data to browser or widget internal representation data)
- Added an `ObjectType` field for advanced usage when you have to implement complex or composed widgets
- Added support for `readonly` fields. If a field is marked as read only, no browser interaction will be performed

0.1.0 (2017-01-03)

Features

- Added widget reference to the widget region so you can navigate to the widget from the widget region and access to widget options specified on the schema
- Added TextAreaWidget

Documentation

- Improved documentation

0.0.1 (2016-12-22)

- Initial release

Python Module Index

p

Index