Django SubUI Tests Documentation

Release 0.2.2

Miroslav Shubernetskiy

Jul 28, 2017

Contents

CHAPTER 1

subui package

SubUI is a framework to ease the pain of writing and running integration tests. The "SubUI" part means that it is not meant to test any of the UI (like html validation) but instead allows to make complete workflow server integration tests.

Introduction

The framework consists of 3 main components:

1. **SubUI test runner**

   This is the interface layer of the SubUI framework. In other words, test methods will instantiate the runner and use it to interact with the SubUI integration tests. Primary job of a SubUI test runner is to execute test steps (described below) in correct order and maintain state between steps if necessary.

   Note: Even though it is called test runner, it does not replace or even relate to `nosetests` test runner.

2. **Test Steps**

   In the most part, this framework is meant to make integration tests for complete workflows (e.g. go to page 1 -> submit form and assert redirect to page 2 -> go to page 2). A test step is a self-contained piece of the complete workflow like "go to page 1". Combinations of multiple steps then make up the workflow. Since steps are independent, they should know how to complete their task (e.g. submit form via `POST`) and validate that they got expected result from the server. To allow flexible validation, they themselves do not validate anything but use validators (described below) to inspect server response in very similar way to how Django Form Field uses validators to verify user-input.

3. **Validator**

   Validator's task is to make assertions about the response from the server. All validators are pretty straight forwards like assert that the response status code is `Redirect - 302` or that redirect header `Location` is returned. More complex assertions can be made by either using multiple validators in each test step or make more complex validator via multiple class inheritance.

Example

An example should show some of the advantages of using this framework for a hypothetical todo application:

```
# define steps
class GoToLogin(TestStep):
    url_name = 'login'
    request_method = 'get'
    validators = [StatusCodeOkValidator]

class SubmitLoginForm(TestStep):
    url_name = 'form'
    validators = [RedirectToRouteValidator(expected_route_name='list')]
    data = {
        'username': 'user',
        'password': 'password',
    }

class GoToList(TestStep):
    url_name = 'list'
    request_method = 'get'
    validators = [StatusCodeOkValidator]

class CreateToDo(TestStep):
    url_name = 'create'
    validators = [RedirectToRouteValidator(expected_route_name='list')]
    data = {
        'notes': 'need to finish something',
        'due date': '2015-01-01',
    }

# integration tests
class TestWorkflow(TestCase):
    def test_login_and_create_todo(self):
        runner = SubUITestRunner(
            OrderedDict((
                ('login', GoToLogin),
                ('login_submit', SubmitLoginForm),
                ('list1', GoToList),
                ('create', CreateToDo),
                ('list2', GoToList),
            )),
            client=self.client
        ).run()

        self.assertNotContains(runner.steps['list1'].response,
                               'need to finish something')
        self.assertContains(runner.steps['list2'].response,
                            'need to finish something')

    def test_just_create(self):
        data = {
            'notes': 'other task here to complete',
            'due date': '2015-01-01',
        }
        runner = SubUITestRunner(
            OrderedDict((
                ('list1', GoToList),
```

```
                ('create', CreateToDo(data=data)),
                ('list2', GoToList),
            )),
            client=self.client
        ).run()

        self.assertNotContains(runner.steps['list1'].response,
                               'other task here to complete')
        self.assertContains(runner.steps['list2'].response,
                            'other task here to complete')
```

some useful things to note about what happened above:

- Reuse of test steps. Since each step is self-contained, they can be combined in different ways to make different integration tests. They can even be reused multiple times within the same integration test.
- Step attributes can easily be overwritten if need to like in `test_just_create` test method - `CreateToDo`'s data is overwritten to post different values.
- Assertions on steps can be performed outside of the test runner. After steps are executed, all steps can be accessed via `runner.steps` attribute which will be an instance of `collections.OrderedDict`.
- *`subui.validators.RedirectToRouteValidator`* is combined validator via multiple inheritance which verifies that the response status code is 302 - Redirect; `Location` response header is present; and that the page redirects to a particular route as determined by Django's `resolve`.

Advanced Use-Cases

More advanced things can be accomplished with the framework. In the previous example, all steps had a fixed url without any parameters. This example will use state to pass information between steps:

```
# login returns redirect to user profile with user id
class LoginStep(TestStep):
    url_name = 'login'
    validators = [RedirectToRouteValidator(expected_route_name='profile')]
    data = {
        'username': 'username',
        'password': 'password',
    }

    def post_test_response(self):
        # extract user kwargs from redirect location
        resolved = resolve(self.response['Location'])
        self.state.push({
            'url_kwargs': resolved.kwargs,
        })

class ProfileStep(StatefulUrlParamsTestStep):
    url_name = 'profile'  # requires url kwargs of username
    request_method = 'get'
    validators = [StatusCodeOkValidator]

class EditProfileStep(StatefulUrlParamsTestStep):
    url_name = 'edit_profile'  # requires url kwarg of username
    validators = [StatusCodeOkValidator]

class TestWorkflow(TestCase):
```

```
    def test_login_and_edit(self):
        data = {
            'username': 'otheruser',
            'password': 'otherpassword',
        }
        runner = SubUITestRunner(
            [
                LoginStep,                    # 0
                ProfileStep,                  # 1
                EditProfileStep(data=data),   # 2
                ProfileStep,                  # 3
            ],
            client=self.client
        ).run()

        self.assertContains(runner.steps[3].response,
                            'otheruser')
```

some notes about what happened:

- `LoginStep` uses a hook *subui.step.TestStep.post_test_response()* to add data to a state. Since state is global for all steps within test runner, other steps can access it.
- `ProfileStep` and `EditProfileStep` subclass *subui.step.StatefulUrlParamsTestStep* which uses state to get url args and kwargs.
- When using `resolve` in `post_test_response`, there is no need to do `try: except Resolver404` since that will be executed after validator verifications hence it is guaranteed that the url will resolve without issues.
- Steps are provided as `list()` instead of `collections.OrderedDict`. Test runner automatically converts the steps into `collections.OrderedDict` with keys as indexes which allows to type test runner a bit faster ;-) in case you don't need to reference steps with particular keys.

Submodules

subui.step module

class subui.step.**StatefulUrlParamsTestStep**(***kwargs*)

Bases: *subui.step.TestStep*

Test step same as *TestStep* except it references `url_args` and `url_kwargs` from the state.

Having url computed from the state, allows for a particular step to change `url_args` or `url_kwargs` hence future steps will fetch different resources.

get_url_args()

Get URL args for Django's `reverse`

Similar to *TestStep.get_url_args()* except url args are retrieved by default from state and if not available get args from class attribute.

Returns tuple of url_args

get_url_kwargs()

Get URL kwargs fpr Django's `reverse`

Similar to *TestStep.get_url_kwargs()* except url args are retrieved by default from state and if not available get kwargs from class attribute.

Returns dict of url_kwargs

class subui.step.**TestStep**(***kwargs*)
Bases: object

Test step for *subui.test_runner.SubUITestRunner*.

The step is responsible for executing a self-contained task such as submitting a form to a particular URL and then make assertions regarding the server response.

Variables

- ***TestStep.test*** (*unittest.TestCase*) – Test class instance with which validators are going to run all their assertions with. By default it is an instance of unittest.TestCase however can be changed to any other class to add additional assertion methods.
- ***url_name*** (*str*) – Name of the url as defined in urls.py by which the URL is going to be calculated.
- ***url_args*** (*tuple*) – URL args to be used while calculating the URL using Django's reverse.
- ***url_kwargs*** (*dict*) – URL kwargs to be used while calculating the URL using Django's reverse.
- ***request_method*** (*str*) – HTTP method to use for the request. Default is "post"
- ***urlconf*** (*str*) – Django URL Configuration.
- ***content_type*** (*str*) – Content-Type of the request.
- ***overriden_settings*** (*dict*) – Dictionary of settings to be overriden for the test request.
- ***data*** (*dict*) – Data to be sent to the server in the request
- ***state*** (*pycontext.context.Context*) – Reference to a global state from the test runner.
- ***TestStep.validators*** (*list*) – List of response validators
- **response** – Server response for the made request. This attribute is only available after *request()* is called.

Parameters **kwargs** (*dict*) – A dictionary of values which will overwrite any instance attributes. This allows to pass additional data to the test step without necessarily subclassing and manually instantiating step instance.

content_type = None

data = None

get_content_type()
Get content_type which will be used when making the test request.

By default this returns *content_type*, if defined, else empty string.

Return type str

get_override_settings()
Get overriden_settings which will be used to decorate the request with the defined settings to be overriden.

By default this returns *overriden_settings*, if defined, else empty dict

Return type dict

get_request_data(*data=None*)
Get data dict to be sent to the server.

Parameters **data** – Data to be used while sending server request. If not defined, *data* is returned.

Return type dict

get_request_kwargs()
Get kwargs to be passed to the `client`.

By default this returns dict of format:

```
{
    'path': ...,
    'data': ...
}
```

Can be overwritten in case additional parameters need to be passed to the client to make the request.

Return type dict

get_url()
Compute the URL to request using Django's `reverse`.

Reverse is called using *url_name*, *get_url_args()* and *get_url_args()*.

Return type str

get_url_args()
Get `url_args` which will be used to compute the URL using `reverse`.

By default this returns *url_args*, if defined, else empty tuple.

Return type tuple

get_url_kwargs()
Get `url_kwargs` which will be used to compute the URL using `reverse`.

By default this returns *url_kwargs*, if defined, else empty dict.

Return type dict

get_urlconf()
Get `urlconf` which will be used to compute the URL using `reverse`

By default this returns *urlconf*, if defined, else None

Return type str

get_validators()
Get all validators.

By default returns *validators* however can be used as a hook to returns additional validators dynamically.

Return type list

init(*client*, *steps*, *step_index*, *step_key*, *state*)
Initialize the step with necessary values from the test runner.

Parameters

- **client** (*django.test.client.Client*) – Django test client to use to make server requests
- **steps** (*collections.OrderedDict*) – All steps from the test runner. This and step index allows to get previous and/or next steps.
- **step_index** (*int*) – Index of the step within all steps test runner will execute.
- **step_key** (*str*) – Key of the state of how it was provided to the test runner in case test step needs to reference other steps within the runner by their.
- **state** (*platform_utils.utils.dt_context.BaseContext*) – Global state reference from the test runner.

next_step
Get previous step instance, if any.

Return type *TestStep*

next_steps
Get `collections.OrderedDict` of next steps excluding itself, if any.

Return type `collections.OrderedDict`

overriden_settings = None

post_request_hook()
Hook which is executed after server request is sent.

post_test_response()
Hook which is executed after validating the response.

pre_request_hook()
Hook which is executed before server request is sent.

pre_test_response()
Hook which is executed before validating the response.

prev_step
Get previous step instance, if any.

Return type *TestStep*

prev_steps
Get `collections.OrderedDict` of previous steps excluding itself, if any.

Note: The steps are returned in order of adjacency from the current step. For example (using list instead of OrederedDict in example):

```
> step = TestStep()
> step.steps = [0, 1, 2, 3, 4]
> step.step_index = 3
> step.prev_steps
[2, 1, 0]
```

Return type `collections.OrderedDict`

request()
Make the server request. Server response is then saved in *request*.

Before making the request, *pre_request_hook()* is called and *post_request_hook()* is called after the request.

Returns server response

request_method = u'post'

state = None

test = <unittest.case.TestCase testMethod=__init__>

test_response()

Test the server response by looping over all validators as returned by *get_validators()*.

Before assertions, *pre_test_response()* is called and *post_test_response()* is called after assertions.

url_args = None

url_kwargs = None

url_name = None

urlconf = None

validators = []

subui.test_runner module

class subui.test_runner.**SubUITestRunner**(*steps*, *client*, *state=None*, ***kwargs*)

Bases: object

SubUI Test Runner.

This is the interface class of the SubUI framework. It runs all of the provided steps in the provided order. Since some steps might need state from previous executed steps, the runner maintains reference to a global state which it then passes to each step during execution.

Variables

- **steps** (*collections.OrderedDict*) – Test steps to be executed
- **client** – Django test client to run tests
- **kwargs** (*dict*) – Additional kwargs given to class. These kwargs will be provided to each step when executed.
- ***state*** (*pycontext.context.Context*) – State to be shared between test step executions.

Parameters

- **steps** (*list, tuple, collections.OrderedDict*) – Steps to be executed. Executed in the provided order hence need to be provided in order-maintaining data-structure.
- **client** (*django.test.client.Client*) – Django test Client to query server with
- **state** (*dict*) – State to be shared between step executions. Optional and by default is empty dict.
- **kwargs** – Additional kwargs to be passed to each step during initialization if it is not already provided as initialized object.

run()
Run the test runner.

This executes all steps in the order there are defined in `SubUITestRunner.steps`. Before each step is executed *subui.step.TestStep.init()* is called providing all the necessary attributes to execute the step like test client, steps, and state).

Returns Reference to the test runner

subui.validators module

class subui.validators.**BaseValidator**(*test_step=None*, ***kwargs*)
Bases: object

Base validator which should be sub-classed to create custom validators.

Variables ***BaseValidator.expected_attrs*** (*tuple*) – Required attributes for the validator. `_check_improper_configuration()` will verify that all of these attributes are defined.

Note: Each validator should only define required attributes for itself. `_get_expected_attrs()` will automatically return required attributes from all current validator and base classes.

Parameters **test_step** – *subui.step.TestStep* instance which will be used to make assertions on.

Note: This parameter is not really required in the `__init__` because the same step will be passed in *test()* however is useful in `__init__` in case subclass validator needs to apply custom login according to values from the `step`.

---

expected_attrs = None

test(*test_step*)
Test the step's server response by making all the necessary assertions. This method by default saves the `test_step` parameter into `step` and validates the validator by using `_check_improper_configuration()`. All subclass validators should actually implement assertions in this method.

Parameters **test_step** – Test step

class subui.validators.**FormInitialDataValidator**(*test_step=None*, ***kwargs*)
Bases: *subui.validators.BaseValidator*

Validator checks that form in response has expected data in initial data.

Variables

- ***initial_data_key*** (*str*) – Expected initial data key to be present in form initial data
- ***expected_initial_data_value*** – Expected value initial value should be set to
- ***context_data_form_name*** (*str*) – Template context data key for form data
- ***test_initial_value*** (*bool*) – Test if the initial value matched expected value
- ***test_initial_value_present*** (*bool*) – Test if the initial value key is present in initial data

- **test_initial_value_not_none** (*bool*) – Test if the initial value is not None

context_data_form_name = u'form'

expected_attrs = (u'initial_data_key',)

expected_initial_data_value = None

initial_data_key = None

test(*test_step*)

test_initial_value = False

test_initial_value_not_none = True

test_initial_value_present = True

class subui.validators.**HeaderContentTypeValidator**(*test_step=None*, ***kwargs*)
Bases: *subui.validators.HeaderValidator*

Validator to check that the expected "Content-Type" header is returned.

header_name = u'Content-Type'

class subui.validators.**HeaderLocationValidator**(*test_step=None*, ***kwargs*)
Bases: *subui.validators.HeaderValidator*

Validator to check that the redirect "Location" header is returned

header_name = u'Location'

class subui.validators.**HeaderValidator**(*test_step=None*, ***kwargs*)
Bases: *subui.validators.BaseValidator*

Validator which can check that a particular header is returned and that it is of particular value.

Variables

- **HeaderValidator.header_name** (*str*) – Name of the header which must be returned
- **expected_header** (*str*) – Expected header value to be returned
- **HeaderValidator.test_header_value** (*bool*) – Whether to test the header value or simply check its existence
- **test_contains_value** – Whether to test if the header value contains another value, or simply check equality to that value

expected_attrs = (u'header_name', u'expected_header')

expected_header = None

header_name = None

test(*test_step*)
Test the response returned with :py:attr:header_name header and that its value is equal to :py:attr:expected_header if :py:attr:test_header_value is True. If :py:attr:test_contains_value is True, header value will be tested to contain expected value.

test_contains_value = False

test_header_value = True

class subui.validators.**RedirectToRouteValidator**(*test_step=None*, ***kwargs*)
Bases: *subui.validators.StatusCodeRedirectValidator*

Validator which also checks that the server returns a redirect to an expected Django route.

Variables ***expected_route_name*** (*str*) – Route name to which the server should redirect to

expected_attrs = (u'expected_route_name',)

expected_route_name = None

test(*test_step*)
Test the response by additionally testing that the response redirects to an expected route as defined by *expected_route_name*.

class subui.validators.**ResponseContentContainsValidator**(*test_step=None*, ***kwargs*)
Bases: *subui.validators.StatusCodeOkValidator*

Validator which also checks that returned response content contains expect string.

Variables ***expected_content*** (*str*) – Expected string in the server response

expected_attrs = (u'expected_content',)

expected_content = None

test(*test_step*)
Test the response by additionally testing that the response context contains expected string as defined by *expected_content*.

class subui.validators.**ResponseContentNotContainsValidator**(*test_step=None*, ***kwargs*)
Bases: *subui.validators.StatusCodeOkValidator*

Validator checks that returned response content does not contain unexpected string.

Variables ***unexpected_content*** (*str*) – Unexpected string in the server response

expected_attrs = (u'unexpected_content',)

test(*test_step*)
Test the response by additionally testing that the response context does not contain the unexpected string as defined by *unexpected_content*.

unexpected_content = None

class subui.validators.**SessionDataValidator**(*test_step=None*, ***kwargs*)
Bases: *subui.validators.BaseValidator*

Validator which allows to verify the data in the session based on session key.

Variables

- ***expected_session_key*** (*str*) – Expected session key to be present in session
- ***expected_session_secondary_keys*** (*list*) – List of Expected session key to be present in session

expected_attrs = (u'expected_session_key',)

expected_session_key = None

expected_session_secondary_keys = []

test(*test_step*)
Test that expected session key is present. If expected session data provided, ensure the expected session key data matches what is there currently.

class subui.validators.**StatusCodeCreatedValidator**(*test_step=None*, ***kwargs*)
Bases: *subui.validators.StatusCodeValidator*

Validator to check that the returned status code is created - 201

expected_status_code = 201

class subui.validators.**StatusCodeOkValidator**(*test_step=None*, ***kwargs*)
Bases: *subui.validators.StatusCodeValidator*

Validator to check that the returned status code is OK - 200

expected_status_code = 200

class subui.validators.**StatusCodeRedirectValidator**(*test_step=None*, ***kwargs*)
Bases: *subui.validators.HeaderLocationValidator*, *subui.validators.StatusCodeValidator*

Validator to check that the server returns a redirect with the "Location" header defined.

expected_status_code = 302

test_header_value = False

class subui.validators.**StatusCodeValidator**(*test_step=None*, ***kwargs*)
Bases: *subui.validators.BaseValidator*

Validator which allows to verify the returned server status code such as "OK-200" or "Redirect-302", etc.

Variables ***StatusCodeValidator.expected_status_code*** (*int*) – Expected status code to be returned by the server

expected_attrs = (u'expected_status_code',)

expected_status_code = None

test(*test_step*)
Test that the response status code matched expected status code.

CHAPTER 2

Django SubUI Tests

Framework to make workflow server integration test suites

- Free software: MIT license
- GitHub: https://github.com/dealertrack/django-subui-tests
- Documentation: http://django-subui-tests.readthedocs.io/

CHAPTER 3

Installing

You can install `django-subui-tests` using pip:

```
$ pip install django-subui-tests
```

CHAPTER 4

Testing

To run the tests you need to install testing requirements first:

```
$ make install
```

Then to run tests, you can use `nosetests` or simply use Makefile command:

```
$ nosetests -sv
# or
$ make test
```

- genindex
- modindex
- search

Python Module Index

S

Index