behave-django Documentation

Release 1.4.0

Mitchel Cabuloy

Jan 09, 2022

Contents

Behave BDD integration for Django

CHAPTER 1

Features

- Web Browser Automation ready
- Database transactions per scenario
- Use Django's testing client
- Use unittest + Django assert library
- Use behave's command line arguments
- Use behave's configuration file
- Fixture loading
- Page objects

CHAPTER 2

Contents

2.1 Installation

Install using pip

```
$ pip install behave-django
```

Add `behave_django` to your `INSTALLED_APPS`

```
INSTALLED_APPS += ['behave_django']
```

2.2 Getting Started

Create the features directory in your project's root directory. (Next to `manage.py`)

```
features/
    steps/
        your_steps.py
    environment.py
    your-feature.feature
```

Run `python manage.py behave`:

```
$ python manage.py behave
Creating test database for alias 'default'...
Feature: Running tests # features/running-tests.feature:1
  In order to prove that behave-django works
  As the Maintainer
  I want to test running behave against this features directory
  Scenario: The Test                        # features/running-tests.feature:6
    Given this step exists                  # features/steps/running_tests.py:4 0.000s
```

(continues on next page)

(continued from previous page)

```
    When I run "python manage.py behave"   # features/steps/running_tests.py:9 0.000s
    Then I should see the behave tests run # features/steps/running_tests.py:14 0.000s

1 features passed, 0 failed, 0 skipped
1 scenarios passed, 0 failed, 0 skipped
3 steps passed, 0 failed, 0 skipped, 0 undefined
Took.010s
Destroying test database for alias 'default'...
```

See the environment.py, running-tests.feature and steps/running_tests.py files in the `features` folder of the project repository for implementation details of this very example. See the folder also for more useful examples.

2.2.1 Alternative folder structure

For larger projects, specifically those that also have other types of tests, it's recommended to use a more sophisticated folder structure, e.g.

```
tests/
    acceptance/
        features/
            example.feature
        steps/
            given.py
            then.py
            when.py
        environment.py
```

Your *behave* configuration should then look something like this:

```
[behave]
paths = tests/acceptance
junit_directory = tests/reports
junit = yes
```

This way you'll be able to cleanly accommodate unit tests, integration tests, field tests, penetration tests, etc. and test reports in a single tests folder.

Note: The behave docs provide additional helpful information on using *behave* with Django and various test automation libraries.

2.3 Web Browser Automation

You can access the test HTTP server from your preferred web automation library via `context.base_url`. Alternatively, you can use `context.get_url()`, which is a helper function for absolute paths and reversing URLs in your Django project. It takes an absolute path, a view name, or a model as an argument, similar to django.shortcuts.redirect.

Examples:

```
# Using Splinter
@when(u'I visit "{page}"')
def visit(context, page):
    context.browser.visit(context.get_url(page))
```

```
# Get context.base_url
context.get_url()
# Get context.base_url + '/absolute/url/here'
context.get_url('/absolute/url/here')
# Get context.base_url + reverse('view-name')
context.get_url('view-name')
# Get context.base_url + reverse('view-name', 'with args', and='kwargs')
context.get_url('view-name', 'with args', and='kwargs')
# Get context.base_url + model_instance.get_absolute_url()
context.get_url(model_instance)
```

2.4 Django's Test Client

Internally, Django's TestCase is used to maintain the test environment. You can access the TestCase instance via `context.test`.

```
# Using Django's testing client
@when(u'I visit "{url}"')
def visit(context, url):
    # save response in context for next step
    context.response = context.test.client.get(url)
```

2.4.1 Simple testing

If you only use Django's test client then *behave* tests can run much quicker with the `--simple` command line option. In this case transaction rollback is used for test automation instead of flushing the database after each scenario, just like in Django's standard `TestCase`.

No HTTP server is started during the simple testing, so you can't use web browser automation. Accessing `context.base_url` or calling `context.get_url()` will raise an exception.

2.4.2 unittest + Django assert library

Additionally, you can utilize unittest and Django's assert library.

```
@then(u'I should see "{text}"')
def visit(context, text):
    # compare with response from ``when`` step
    response = context.response
    context.test.assertContains(response, text)
```

2.5 Test Isolation

2.5.1 Database transactions per scenario

Each scenario is run inside a database transaction, just like your regular TestCases. So you can do something like:

```
@given(u'user "{username}" exists')
def create_user(context, username):
    # This won't be here for the next scenario
    User.objects.create_user(username=username, password='correcthorsebatterystaple')
```

And you don't have to clean the database yourself.

2.6 Fixture Loading

behave-django can load your fixtures for you per feature/scenario. There are two approaches to this:

- loading the fixtures in `environment.py`, or
- using a decorator on your step function

2.6.1 Fixtures in environment.py

In `environment.py` we can load our context with the fixtures array.

```
def before_all(context):
    context.fixtures = ['user-data.json']
```

This fixture would then be loaded before every scenario.

If you wanted different fixtures for different scenarios:

```
def before_scenario(context, scenario):
    if scenario.name == 'User login with valid credentials':
        context.fixtures = ['user-data.json']
    elif scenario.name == 'Check out cart':
        context.fixtures = ['user-data.json', 'store.json', 'cart.json']
    else:
        # Resetting fixtures, otherwise previously set fixtures carry
        # over to subsequent scenarios.
        context.fixtures = []
```

You could also have fixtures per Feature too

```
def before_feature(context, feature):
    if feature.name == 'Login':
        context.fixtures = ['user-data.json']
        # This works because behave will use the same context for
        # everything below Feature. (Scenarios, Outlines, Backgrounds)
    else:
        # Resetting fixtures, otherwise previously set fixtures carry
        # over to subsequent features.
        context.fixtures = []
```

Of course, since `context.fixtures` is really just a list, you can mutate it however you want, it will only be processed upon leaving the `before_scenario()` function of your `environment.py` file. Just keep in mind that it does not reset between features or scenarios, unless explicitly done so (as shown in the examples above).

Note: If you provide initial data via Python code using the ORM you need to place these calls in `before_scenario()` even if the data is meant to be used on the whole feature. This is because Django's

`LiveServerTestCase` resets the test database after each scenario.

2.6.2 Fixtures using a decorator

You can define Django fixtures using a function decorator. The decorator will load the fixtures in the `before_scenario`, as documented above. It is merely a convenient way to keep fixtures close to your steps.

```
from behave_django.decorators import fixtures

@fixtures('users.json')
@when('someone does something')
def step_impl(context):
    pass
```

Note: Fixtures included with the decorator will apply to all other steps that they share a scenario with. This is because *behave-django* needs to provide them to the test environment before processing the particular scenario.

2.6.3 Support for multiple databases

By default, Django only loads fixtures into the `default` database.

Use `before_scenario` to load the fixtures in all of the databases you have configured if your tests rely on the fixtures being loaded in all of them.

```
def before_scenario(context, scenario):
    context.databases = '__all__'
```

You can read more about it in the Multiple database docs.

2.7 Using Page Objects

Warning: This is an *alpha* feature. It may be removed or its underlying implementation changed without a deprecation process! Please follow the discussions in related issues or on Gitter if you plan to use it.

With *behave-django* you can use the Page Object pattern and work on a natural abstraction layer for the content or behavior your web application produces. This is a popular approach to make your tests more stable and your code easier to read.

```
# FILE: steps/pageobjects/pages.py
from behave_django.pageobject import PageObject, Link

class Welcome(PageObject):
    page = 'home'  # view name, model or URL path
    elements = {
        'about': Link(css='footer a[role=about]'),
    }
```

(continues on next page)

(continued from previous page)

```
class About(PageObject):
    page = 'about'
```

```
# FILE: steps/welcome.py
from pageobjects.pages import About, Welcome

@given(u'I am on the Welcome page')
def step_impl(context):
    context.welcome_page = Welcome(context)
    assert context.welcome_page.response.status_code == 200

@when(u'I click on the "About" link')
def step_impl(context):
    context.target_page = \
        context.welcome_page.get_link('about').click()
    assert context.target_page.response.status_code == 200

@then(u'The About page is loaded')
def step_impl(context):
    assert About(context) == context.target_page
```

A `PageObject` instance automatically loads and parses the page you specify by its `page` attribute. You then have access to the following attributes:

`request` The HTTP request used by the Django test client to fetch the document. This is merely a convenient alias for `response.request`.

`response` The Django test client's HTTP response object. Use this to verify the actual HTTP response related to the retrieved document.

`document` The parsed content of the response. This is, technically speaking, a Beautiful Soup object. You *can* use this to access and verify any part of the document content, though it's recommended that you only access the elements you specify with the `elements` attribute, using the appropriate helper methods.

Helpers to access your page object's elements:

`get_link(name) -> Link` A subdocument representing a HTML anchor link, retrieved from `document` using the CSS selector specified in `elements[name]`. The returned `Link` object provides a `click()` method to trigger loading the link's URL, which again returns a `PageObject`.

Note: *behave-django*'s PageObject is a headless page object, meaning that it doesn't use Selenium to drive the user interface.

If you need a page object that encapsulates Selenium you may take a look at alternative libraries, such as page-object, page-objects or selenium-page-factory. But keep in mind that this is a different kind of testing:

- You'll be testing the Web browser, hence for Web browser compatibility.
- Preparing an environment for test automation will be laborious.
- Mocking objects in your tests will be difficult, if not impossible.
- Your tests will be *significantly* slower and potentially brittle.

Think twice if that is really what you need. In most cases you'll be better off testing your Django application code only. That's when you would use Django's test client and our headless page object.

2.8 Environment Setup

2.8.1 django_ready hook

You can add a `django_ready` function in your `environment.py` file in case you want to make per-scenario changes inside a transaction.

For example, if you have factories you want to instantiate on a per-scenario basis, you can initialize them in `environment.py` like this:

```
from myapp.main.tests.factories import UserFactory, RandomContentFactory


def django_ready(context, scenario):
    # This function is run inside the transaction
    UserFactory(username='user1')
    UserFactory(username='user2')
    RandomContentFactory()
```

Or maybe you want to modify the `test` instance:

```
from rest_framework.test import APIClient


def django_ready(context, scenario):
    context.test.client = APIClient()
```

2.9 Configuration

2.9.1 Command line options

You can use regular *behave* command line options with the `behave` management command.

```
$ python manage.py behave --tags @wip
```

Additional command line options provided by *behave-django*:

`--use-existing-database`

Don't create a test database, and use the database of your default runserver instead. USE AT YOUR OWN RISK! Only use this option for testing against a *copy* of your production database or other valuable data. Your tests may destroy your data irrecoverably.

`--keepdb`

Starting with Django 1.8, the `--keepdb` flag was added to `manage.py test` to facilitate faster testing by using the existing database instead of recreating it each time you run the test. This flag enables `manage.py behave --keepdb` to take advantage of that feature. More information about `--keepdb`.

--simple

Use Django's simple `TestCase` which rolls back the database transaction after each scenario instead of flushing the entire database. Tests run much quicker, however HTTP server is not started and therefore web browser automation is not available.

2.9.2 Behave configuration file

You can use *behave*'s configuration file. Just create a `behave.ini`, `.behaverc`, `setup.cfg` or `tox.ini` file in your project's root directory and behave will pick it up. You can read more about it in the behave docs.

For example, if you want to have your features directory somewhere else. In your `.behaverc` file, you can put

```
[behave]
paths=my_project/apps/accounts/features/
      my_project/apps/polls/features/
```

Behave should now look for your features in those folders.

2.10 Contributing

Want to help out with *behave-django*? Cool! Here's a quick guide to do just that.

2.10.1 Preparation

Fork, then clone the repo:

```
$ git clone git@github.com:your-username/behave-django.git
```

Ensure Tox is installed. We use it to run linters, run the tests and generate the docs:

```
$ pip install tox
```

2.10.2 Essentials

Make sure the tests pass. The `@failing` tag is used for tests that are supposed to fail. The `@requires-live-http` tag is used for tests that can't run with `--simple` flag. See the `[testenv]` section in `tox.ini` for details.

```
$ tox -lv               # show all Tox targets
$ tox -e py37-django22  # run just a single target
$ tox                   # run all linting and tests
```

2.10.3 Getting your hands dirty

Start your topic branch:

```
$ git checkout -b your-topic-branch
```

Make your changes. Add tests for your change. Make the tests pass:

```
$ tox -e behave-latest
```

Finally, make sure your tests pass on all the configurations *behave-django* supports. This is defined in `tox.ini`. The Python versions you test against need to be available in your PATH.

```
$ tox
```

You can choose not to run all tox tests and let the CI server take care about that. In this case make sure your tests pass when you push your changes and open the PR.

2.10.4 Documentation changes

If you make changes to the documentation generate it locally and take a look at the results. Sphinx builds the output in `docs/_build/`.

```
$ tox -e docs
$ python -m webbrowser -t docs/_build/html/index.html
```

2.10.5 Finally

Push to your fork and submit a pull request.

To clean up behind you, you can run:

```
$ tox -e clean
```

2.10.6 Other things to note

- Write tests.
- Your tests don't have to be *behave* tests. `:-)`
- We're using PEP8 as our code style guide (`flake8` will run over the code on the CI server).

Thank you!

CHAPTER 3

Version Support

behave-django is tested against the officially supported combinations of Python and Django (Django 2.2, 3.0 on Python 3.5, 3.6, 3.7, 3.8).

The version of behave is not tied to our integration (read: "independent"). We test against the latest release on PyPI, and run a sample against Behave's current `master` branch.

CHAPTER 4

Indices and tables

- search