Factory Djoy Documentation

Release 2.1.3

James Cooke

Jan 19, 2020

Contents

CHAPTER 1

Factory Djoy

Factories for Django, creating valid model instances every time.

Factory Djoy provides two simple classes, `UserFactory` and `CleanModelFactory`, which wrap Factory Boy. They call `full_clean()` when creating Django model instances to ensure that only valid data enters your Django database.

1.1 Compatibility

Factory Djoy is compatible with:

- Django 1.11, 2.2 and 3.0.
- Python 3 (3.5, 3.6, 3.7, 3.8)
- Factory Boy version 2.11 or greater.

1.2 Resources

- Documentation on ReadTheDocs
- Package on PyPI
- Source code on GitHub
- Licensed on MIT
- Changelog

1.3 Contents

1.3.1 Installation and Quick Start

Installation

Easiest is to install with `pip`:

```
pip install factory_djoy
```

Quick Start

Factory Djoy provides a `UserFactory` out of the box which will build valid instances of the default Django contrib auth `User` model.

```
from factory_djoy import UserFactory

user = UserFactory()
```

For any other Django models that you would like valid instances for, create your factory by inheriting from `CleanModelFactory`.

```
from factory_djoy import CleanModelFactory

from project.app.models import Item


class ItemFactory(CleanModelFactory):
    class Meta:
        model = Item
```

1.3.2 UserFactory

```
from factory_djoy import UserFactory
```

`UserFactory` provides valid instances of the default Django user at `django.contrib.auth.User`. By default it creates valid users with unique usernames, random email addresses and populated first and last names.

```
>>> user = UserFactory()
>>> user.username
'cmcmahon'
>>> user.email
'amandajones@example.com'
>>> user.first_name + ' ' + user.last_name
'Tiffany Hoffman'
```

Just as with `factory_boy`, the API can be used to create instances with specific data when required:

```
>>> user = UserFactory(first_name='René')
>>> user.first_name
'René'
```

Passwords

In order to be able to use the generated User's password, then pass one to the factory at generation time, otherwise the randomly generated password will be hashed and lost. The `UserFactory` correctly uses `set_password` to set the password for the created User instance.

```
>>> from django.test import Client
>>> user = UserFactory(password='test')
>>> assert Client().login(username=user.username, password='test')
```

If you want a User that can not log in, then pass `None` as the password.

```
>>> user = UserFactory(password=None)
>>> user.has_usable_password()
False
```

Unique usernames

The factory will try up to 200 times per instance to generate a unique username.

As a result of the limitations around saving Users to the database in Django, these unique usernames are generated with a 3 step process:

- Ask `faker` for a random username. Compare it to those generated already and use it if its unique.
- Ensure that the picked username is not already in the Django database.
- Create instance with planned unique username and check the generated instance with a call to `full_clean`. This means that if anything has gone wrong during generation, then a `ValidationError` will be raised. This also means that values that might be passed into the Factory are tested for validity:

```
>>> UserFactory(username='user name')
Traceback (most recent call last):
...
ValidationError: {'username': ['Enter a valid username. This value may contain␣
↪only letters, numbers and @/./+/-/_ characters.']}
```

Invalid data

If you need to create invalid data in your tests, then the `build` strategy is available:

```
>>> user = UserFactory.build(username='user name')
>>> user.id is None
True
>>> user.full_clean()
Traceback (most recent call last):
...
ValidationError: {'username': ['Enter a valid username. This value may contain only␣
↪letters, numbers and @/./+/-/_ characters.']}
>>> user.save()
```

1.3.3 CleanModelFactory

This is a generic wrapper for `DjangoModelFactory`. It provides all the functionality of `DjangoModelFactory` but extends `create` to call `full_clean` at post-generation time. This validation ensures that your test factories

only create instances that meet your models' field level and model level validation requirements - this leads to better tests.

Example

Given a very simple model called `Item` which has one name field that is required and has a max length of 5 characters:

```
class Item(Model):
    """
    Single Item with one required field 'name'
    """
    name = CharField(max_length=5, unique=True)
```

Then we can create a clean factory for it using `CleanModelFactory`:

```
from factory_djoy import CleanModelFactory

from yourapp.models import Item


class SimpleItemFactory(CleanModelFactory):
    class Meta:
        model = Item
```

Now we haven't defined any default value for the `name` field, so if we use this factory with no keyword arguments then `RuntimeError` is raised:

```
>>> SimpleItemFactory()
Traceback (most recent call last):
...
django.core.exceptions.ValidationError: {'name': ['This field cannot be blank.']}

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
...
RuntimeError: Error building <class 'yourapp.models.Item'> with SimpleItemFactory.
Bad values:
  name: ""
```

However, if we pass a valid name, then everything works OK:

```
>>> SimpleItemFactory(name='tap')
```

Automatically generating values

The point of using `CleanModelFactory` is not to make testing harder because lots of keyword arguments are needed for each factory call, instead it should be easier and more reliable. Really the work with `SimpleItemFactory` above is not complete.

Now we replace `SimpleItemFactory` with a new `ItemFactory` that generates the required `name` field by default. We're going to use `factory_boy`'s Fuzzy attributes to generate random default values each time the model is instantiated and because `full_clean` is called every time an instance is created, we will know that every instance passed validation.

```
from factory.fuzzy import FuzzyText
from factory_djoy import CleanModelFactory


class ItemFactory(CleanModelFactory):
    class Meta:
        model = Item

    name = FuzzyText(length=5)
```

Now we can happily generate multiple instances of `Item` leaving the factory to create random names for us.

```
>>> item = ItemFactory()
>>> item.name
'TcEBK'
```

Alternatively, if you wanted all your created `Item` instances to have the name value for `name` each time, you can just set that in the factory declaration.

```
class FixedItemFactory(CleanModelFactory):
    class Meta:
        model = Item

    name = 'thing'
```

However, in this instance, you will receive `RuntimeError` because `name` is expected to be unique.

```
>>> FixedItemFactory.create_batch(2)
Traceback (most recent call last):
...
django.core.exceptions.ValidationError: {'name': ['Item with this Name already exists.
↪']}

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
...
RuntimeError: Error building <class 'yourapp.models.Item'> with FixedItemFactory.
Bad values:
  name: "thing"
```

`full_clean` is triggered only with the `create` strategy. Therefore using `build` followed by `save` can provide a way to emulate "bad" data in your Django database if that's required. In this example, we can create an `Item` instance without a `name`.

```
>>> item = FixedItemFactory.build(name='')
>>> item.save()
>>> assert item.id
```

After saving successfully, if `full_clean` is called then the saved `Item` will fail validation because it does not have a `name`:

```
>>> item.full_clean()
Traceback (most recent call last):
...
django.core.exceptions.ValidationError: {'name': ['This field cannot be blank.']}
```

Side notes

- The `ItemFactory` example above is used in testing `factory_djoy`. The `models.py` can be found in `test_framework` and the tests can be found in the `tests` folder.
- `CleanModelFactory` does not provide any `get_or_create` behaviour.

1.3.4 Motivation

Testing first

Primarily the goal of factories is to provide "easy to generate" data at test time - however this data must be 100% reliable, otherwise it's too easy to create false positive and false negative test results. By calling `full_clean` on every Django instance that is built, this helps to create certainty in the data used by tests - the instances will be valid as if they were created through the default Django admin.

Therefore, since it's so important that each factory creates valid data, these wrappers are tested rigorously using Django projects configured in the `test_framework` folder.

See also

- django-factory_boy which implements more factories for Django's stock models, but doesn't validate generated instances and has less tests.
- Django Factory Audit which compares every Django factory library I could find with respect to how they create valid instances.
- Django's model save vs full_clean for an explanation of how Django can screw up your data when saving.

1.3.5 Contribution

- Please see GitHub Issues. There are a number of outstanding tasks.
- Please ensure that any provided code:
  - Has been developed with "test first" process.
  - Can be auto-merged in GitHub.
  - Passes testing on Circle CI.
- Check out the development documentation for info on how to build, test and upload.

1.3.6 Development and Testing

How to work on and test Factory Djoy, across different versions of Django.

Quick start

The test framework is already in place and `tox` is configured to make use of it. Therefore the quickest way to get going is to make use of it.

Clone the repository and drop in:

```
$ git clone git@github.com:jamescooke/factory_djoy.git
$ cd factory_djoy
```

Create a virtual environment. This uses a bare call to `virtualenv`, you might prefer to use `workon`:

```
$ make venv
```

Activate the virtual environment and install requirements:

```
$ . venv/bin/activate
$ make install
```

Run all tests using `tox` for all versions of Python and Django:

```
$ make test
```

Circle will also run linting before the main test run:

```
$ make lint
```

Testing with real Django projects

`factory_djoy` is asserted to work with different vanilla versions of Django. For each version of Django tested, a default project and app exist in the `test_framework` folder.

This structure has the following features:

- `test_framework` contains a folder for each version of Django. For example, the Django 1.11 project is in the `test_framework/django111` folder.
- Every project is created with the default `django-admin startproject` command.
- In every project, a test settings file contains all the default settings as installed by Django, but adds the `djoyapp` app to the list of `INSTALLED_APPS`.
- Every `djoyapp` contains a `models.py` and provides some models used for testing.
- Initial migrations for the models also exist, created using the matching version of Django using the default `./manage.py makemigrations` command.
- Every project has a `tests` folder wired into the Django project root. This contains the tests that assert that `factory_djoy` behaves as expected.
- Every project's tests are executed through the default `./manage.py test` command.

Versioning notes

- `tox` is used to manage versions of Python and Django installed at test time.
- The latest point release from each major Django version is used, excluding versions that are no longer supported.
- The current version of Django in use at testing is compiled into each `django*.txt` file.
- To bump all package versions, the `bump_reqs` recipe can be used.

```
$ . venv/bin/activate
$ make bump_reqs
```

Creating Django test projects for Django version

In order to add a version of Django to the test run:

- Install the new version of Django into the current virtual environment:

```
$ pip install -U django
```

- Ask the new version of Django to create projects and all `test_framework` structures:

```
$ cd test_framework
$ make build
```

  Please note that creating a Django test project will fail if the target folder already exists. All `django*` folders can be removed with `make clean` - they can be rebuilt again identically with the `build` recipe.

- Add a requirements file for the new version of Django. For version `1.11`:

```
$ cd test_framework/requirements
$ cat > django111.in
Django>=1.11,<2
^D
$ make all
```

- Add the new Django version to `tox.ini`. (There's probably a better DRYer way to complete this.)
- Remember to add the new Django version to the README and do a release.

Working locally

If there are multiple tests to run this can become inefficient with `tox`. Therefore, you can use the helper local environment configured inside `test_framework`. This installs Python 3.5 and latest Django.

Create a new virtual environment in the `test_framework` folder and install the requirements:

```
$ cd test_framework
$ make venv
$ . venv/bin/activate
$ make install
```

The test framework means that all the tests can be run on the test models and factories using the standard `manage.py` test command. So, if working with Django 1.10, after calling `make build` to create the app and folder structure for that Django version, then all tests can be run with:

```
$ make test
```

Release process

Decide the new version number. Semantic versioning is used and it will look like `1.2.3`.

- In a Pull Request for the release:
  - Update CHANGELOG with changes.
  - Set version number in `factory_djoy/__about__.py`
  - Ensure Pull Request is GREEN, then merge.

- With the newly merged master:
  - Run tests locally:

```
$ make lint test
```

  - Clean out any old distributions and make new ones:

```
$ make clean dist
```

  - Test upload with Test PyPI and follow it with an install direct from Test PyPI (might need to create a `~/.pypirc` file with settings for the test server:

```
$ make test-upload

$ deactive
$ virtualenv /tmp/tenv
$ . /tmp/tenv/bin/activate
$ make test-install
```

  - Tag release branch and push it:

```
$ git tag v1.2.3
$ git push origin --tags
```

  - Upload to PyPI:

```
$ make upload
```

All done.

Post release:

- Ensure that link in CHANGELOG to the new diff works OK on GitHub.
- Check new docs are built on RTD.

Helper recipes

The root `Makefile` has a couple of helper recipes (note this is different to the `Makefile` in `test_settings`):

- `dist`: Creates the distribution files.
- `upload`: Push generated distribution to PyPI.
- `bump_reqs`: Update all packages, commit updates to a new `auto/bump-requirements` branch and push it to origin.
- `clean`: Remove all compiled Python files, distributions, etc.