
djangocms-restapi Documentation

Release 1.0.0

Rolf Håvard Blindheim

October 16, 2015

1	Working environment	3
1.1	Installing virtualenv and virtualenvwrapper	3
1.2	Creating a virtual environment for the project	4
2	Project structure	5
2.1	Setting up the project	5
2.2	Write some code	9
3	Testing	11
3.1	The setup script	11
3.2	Setting up the test runner	12
3.3	Running the test suite	14
3.4	Writing a test case	14
3.5	Testing multiple environments	15
3.6	Continuous Integration	16
4	Documenting your library	19
4.1	Writing documentation	19
4.2	Compile the documentation	21
4.3	Read the Docs	22
5	Pack and deploy your masterpiece!	25
5.1	Push the library to PyPi	25
6	Conclusion	29
7	Indices and tables	31

Django has a vast ecosystem of 3rd party modules. This crash course is an attempt at writing a guide for writing, testing, documenting and deploying such modules to [Pypi](#) - the Python Package Index.

The concepts in this guide does not strictly apply just to Django, but can with minor adjustments be used as a guideline for writing all sorts of portable Python modules.

It is not however; a tutorial for Python nor Django. If you are new to Django, I suggest you complete the [Django tutorial](#) first, then come back and read the crash course.

Contents:

Working environment

When working on a new project it will probably have different requirements than your previous projects.

Some reasons might include:

1. Required libraries probably differs
2. Same libraries, but different or incompatible versions
3. You shouldn't pollute your system with libraries only required for a single project.
4. It's easy to fix in case you screw up something

There are many solutions to this, the most popular includes:

1. `virtualenv`
2. `pyenv`
3. Vagrant

Each of these have their own strength and weaknesses, but I usually goes for `virtualenv` with a wrapper library called `virtualenvwrapper`. I think it works quite good, so I'm going to use it in this guide as well.

1.1 Installing `virtualenv` and `virtualenvwrapper`

`Virtualenv` creates a new python “distribution” in its own directory. All required python libraries will be installed in this directory as well, instead of in the operating systems python environment.

Installing is easy. First you need `pip` installed. The `pip` command is available in most recent python installs by default. We'll install the `virtualenv` and `virtualenvwrapper` libraries in our system environment, because it need to have access to the system installed python distribution in order to make copies of it.

```
~$ sudo pip install virtualenvwrapper
```

The above command will install both `virtualenv` and the `virtualenvwrapper` which gives us an easy cli for managing our virtual environments.

The `virtualenvwrapper` uses an environment variable called `WORKON_HOME`, which defaults to `~/.virtualenvs`. All the virtual environments created are placed in this directory. You can of course override the `WORKON_HOME` variable if you choose to, but I think it's a good place to store my virtual environments.

1.2 Creating a virtual environment for the project

We are now ready to create a new and shiny environment for our project. The following statement is probably quite opinionated, but it's my guide, so here it comes ;)

All new python projects should be written using the most recent Python version (3.4 at the time of writing)!

Of course, there are exceptions to this rule, but I believe that it is better to write in the most recent version, and use the `six` compatibility layer to add backward support.

That being said, go ahead and install Python 3 (the most recent version) if you don't have it installed.

```
(Skip this step if you already have Python3 installed)
~$ sudo apt-get install python3
```

Let's create a new virtual environment called `django-foobar` for our project.

```
~$ mkvirtualenv -p `which python3` django-foobar
Running virtualenv with interpreter /usr/local/bin/python3
<snip>
Installing setuptools, pip...done.
(django-foobar)~$
```

You'll see the currently active environment in your prompt. In order to deactivate the environment, simply enter the command `deactivate`, and to activate it again you would type in `workon django-foobar`.

While the virtual environment is active, all packages you install will be installed in the `$WORKON_HOME/django-foobar/lib/python3.4/site-packages/`, the python interpreter will be `$WORKON_HOME/django-foobar/bin/python` and so on. So when you execute the `python` command, it will be the one in your active environment, not your system python.

Good stuff!

Project structure

Everybody tends to have a personal preference for the structure of their project, and I'm no different. After some trial and tweaking I've come up with a structure I'm quite happy with.

2.1 Setting up the project

First, activate the `django-foobar` environment if it's not already active. Next we need to install Django itself into the new environment.

```
~$ workon django-foobar
(django-foobar):~$ pip install Django
```

2.1.1 Creating the working directory

```
(django-foobar):~$ mkdir django-foobar
(django-foobar):~$ cd django-foobar
```

Note: Whenever referring to the “working directory” in this guide, I'm referring to this. In other words, the top level directory for the entire project.

2.1.2 Set up the testing environment

TDD is a good practice to follow when writing 3rd party Django modules. If you want your library to succeed and people to use and contribute to it, you'll need a good test suite! We are going to use the `django-admin` command to create a testing environment for our module. Inside the project directory enter the following command (make sure you include the `.` on the end!)

```
(django-foobar):~/django-foobar$ django-admin startproject tests .
```

Now, you might be wondering why we are using the `django-admin startproject` command to create the testing environment, and not the module we are going to develop. That's because we don't want to distribute an entire Django project. We only want to distribute the `django-foobar` library we're going to write, but we want to make sure it can run in an existing Django project. That's what we're going to use the `tests` project for!

2.1.3 Module source code

Next, we need to create a directory containing the source code for our `django-foobar` module. Python modules use underscores instead of dashes in their package names, so the python module will be called `django_foobar`. Either create a directory called `django_foobar` with an `__init__.py` file inside it, or just use the `django-admin` command to create a new app.

```
(django-foobar):~/django-foobar$ django-admin startapp django_foobar
```

2.1.4 Documentation

Obviously, we need to document our library. This can be done with various tools, but I like to use [Sphinx](#). That's the very same tool I've used to write this guide, so you know how the documentation will look like =>

```
(django-foobar):~/django-foobar$ pip install Sphinx
...
Lots of stuff going on
...
(django-foobar):~/django-foobar$ sphinx-quickstart docs
Welcome to the Sphinx 1.3.1 quickstart utility.

Please enter values for the following settings (just press Enter to
accept a default value, if one is given in brackets).

Selected root path: docs

You have two options for placing the build directory for Sphinx output.
Either, you use a directory "_build" within the root path, or you separate
"source" and "build" directories within the root path.
> Separate source and build directories (y/n) [n]:
```

Just hit <enter> for default values and write some sensible values for those which requires input (like project name, author, and so on).

2.1.5 Requirements

The `requirements.txt` file contains all the libraries required for *developing* the module. Requirements for *installing* the module will be in another file called `setup.py`. Don't pay any attention to that for now, we'll create it later in this guide.

Create the `requirements.txt` file in your project root.

```
(django-foobar):~/django-foobar$ touch requirements.txt
```

Open the file in a text editor and add a few dependencies. It's quite common to lock dependency versions when developing a library, but it's not strictly required. If no version lock is found, `pip` will install the most recent version.

```
Django==1.8.4
Sphinx==1.3.1
sphinx-rtd-theme==0.1.8
```

We'll be adding more to this file as we go along.

2.1.6 README.txt and LICENSE.txt

Create a README.txt and LICENSE.txt file. The README.txt file should contain a brief description of the module, as well as other useful information. In case you decide to upload your project to a source hosting service such as Github or Bitbucket, the content of this file will be displayed on the front page of your repository.

The LICENSE.txt file is not so important right now, but if you are going to distribute the source code, it is good practice to slap a license on it. There are many licenses to choose from, but I tend to choose a fairly liberal license such as the [MIT License](#) or the [Apache v2.0 License](#). There is no single license which match every project, so choosing the right one depends a lot on the project type and use case. Also, if you work for a company, they might have their own rules and restrictions for licensing source code, so make sure to pay attention to that as well ;)

You don't have to decide now, just leave the LICENSE.txt file there as a reminder. Check out [this link](#) for a list of popular open source licenses.

```
(django-foobar):~/django-foobar$ touch README.txt LICENSE.txt
```

2.1.7 Source control

This step is optional, but you probably want to add source control to your project. There is different source control tools, but I like Git, so let's initialize a git repository in the project root.

```
(django-foobar):~/django-foobar$ git init
Initialized empty Git repository in /path/to/your/project/.git/
```

Not all files in your project should go into source control. Add a .gitignore file in order to tell Git which files or directories to ignore.

```
(django-foobar):~/django-foobar$ touch .gitignore
```

Open the .gitignore file in your favorite text editor and paste the following lines:

```
__pycache__/  
*.py[cod]  
  
build/  
dist/  
sdist/  
.eggs/  
*.egg-info/  
  
# Unit test / coverage reports  
htmlcov/  
.tox/  
.coverage  
.cache  
nosetests.xml  
coverage.xml  
  
# Translations  
*.mo  
*.pot  
  
# Django stuff:
```

```
*.log
*.db

# Sphinx documentation
docs/_build/
```

You can add stuff to this file as you go along in case there's something we have missed.

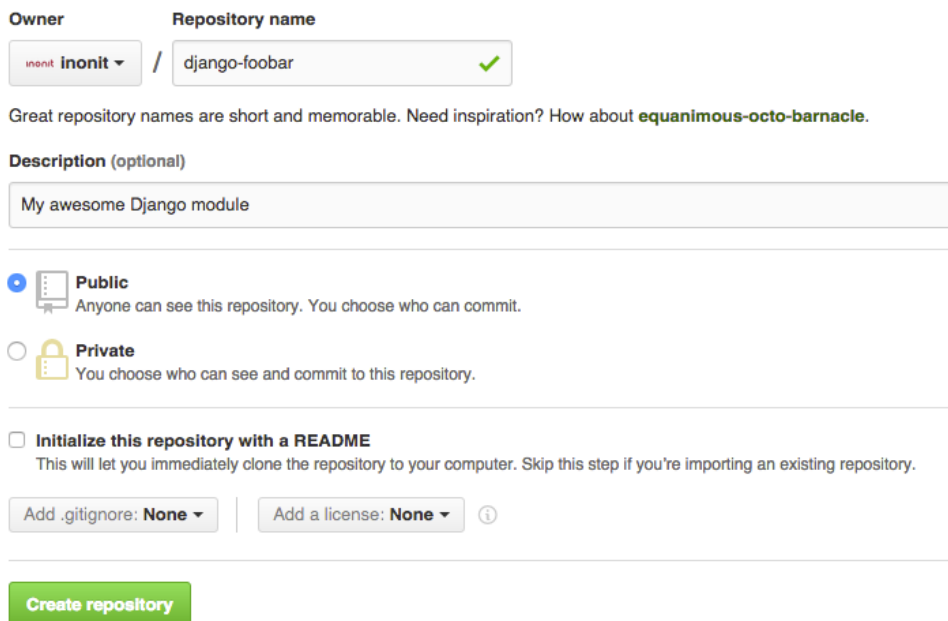
Now is a good time to commit your work!

```
(django-foobar):~/django-foobar$ git add *
(django-foobar):~/django-foobar$ git commit -m "initial commit"
```

Github

If you don't have a [Github](#) account, go ahead and create one now!

1. Create an empty repository for your code



Owner **Repository name**

inonit inonit / django-foobar ✓

Great repository names are short and memorable. Need inspiration? How about **equanimous-octo-barnacle**.

Description (optional)

My awesome Django module

☒ **Public**
Anyone can see this repository. You choose who can commit.

☐ **Private**
You choose who can see and commit to this repository.

☐ **Initialize this repository with a README**
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: **None** | Add a license: **None** ⓘ

Create repository

Fill in some details about your project, but make it public.

2. Push your code to the new Github repository.

```
(django-foobar):~/django-foobar$ git remote add origin https://github.com/<your-user>/django-foobar
(django-foobar):~/django-foobar$ git push -u origin master
```

Enter your credentials when asked.

There. We now have a pretty good project structure for developing the amazing `django-foobar` module. It should look something like this:

```
-- LICENSE.txt
-- README.rst
-- django_foobar
|  -- __init__.py
|  -- admin.py
```

```

|   -- migrations
|   |   -- __init__.py
|   -- models.py
|   -- tests.py
|   -- views.py
-- docs
|   -- Makefile
|   -- _build
|   -- _static
|   -- _templates
|   -- conf.py
|   -- index.rst
|   -- make.bat
-- manage.py
-- requirements.txt
-- .gitignore
-- tests
|   -- __init__.py
|   -- settings.py
|   -- urls.py
|   -- wsgi.py

```

2.2 Write some code

Now, this is not a guide for writing Django code, so we'll just write a small testable unit in our `django_foobar` module.

Open the `django_foobar/views.py` file in the and write in the following code.

```

# -*- coding: utf-8 -*-

from __future__ import absolute_import, unicode_literals

from django.http import HttpResponse
from django.views.generic import View

class DeepThoughtView(View):

    def get(self, request):
        return HttpResponse(42)

```

Next, open the `tests/settings.py` file and add your module to `INSTALLED_APPS`

```

INSTALLED_APPS = (
    <snip>

    'django_foobar',
)

```

Finally, hook up the view in the `tests/urls.py` file.

```

from django_foobar.views import DeepThoughtView

urlpatterns = [
    <snip>

```

```
    url(r'^deephought/$', view=DeepThoughtView.as_view(), name='django_foobar_deephought'),  
]
```

Testing

Django includes a test runner which can be run by issuing the `python manage.py test` command. However, when writing 3rd party modules I like to use a slightly different approach. It includes creating a proper `setup.py` file, the `nose` and `tox` testing frameworks.

3.1 The setup script

Let's start by creating a `setup.py` file. This file will be used to run test suite and later build, package and upload the module to `pypi`.

Make sure you create this file in the projects working directory (next to all the other files we created earlier).

```
(django-foobar):~/django-foobar$ touch setup.py
```

We'll base our setup script on [Setuptools](#). In order to make this as painless as possible, some nice people has created a bootstrap script for `setuptools` which we'll take advantage of. Download the `ez_setup.py` script and place it in the projects working directory.

```
(django-foobar):~/django-foobar$ curl -O https://bootstrap.pypa.io/ez_setup.py
```

Once downloaded, open the `setup.py` file in a text editor and enter the following code.

```
import re
import os

try:
    from setuptools import setup
except ImportError:
    from ez_setup import use_setuptools
    use_setuptools()
    from setuptools import setup

setup(
    name="django-foobar",
    version="1.0",
    description="Super awesome portable module for Django",
    long_description="Here you can elaborate a bit to explain the use case and purpose of the module",
    author="John Doe",
    author_email="john.doe@example.com",
    url="https://github.com/github-user/django-foobar",
    download_url="https://github.com/github-user/django-foobar.git",
    license="MIT License",
```

```

packages=[
    "django_foobar",
],
include_package_data=True,
install_requires=[
    "Django>=1.7.0",
],
tests_require=[
    "nose",
    "coverage",
],
zip_safe=False,
test_suite="tests.runtests.start",
classifiers=[
    "Operating System :: OS Independent",
    "Development Status :: 3 - Alpha",
    "Environment :: Web Environment",
    "Framework :: Django",
    "Intended Audience :: Developers",
    "License :: OSI Approved :: MIT License",
    "Programming Language :: Python :: 2",
    "Programming Language :: Python :: 3",
    "Topic :: Software Development :: Libraries :: Python Modules",
]
)

```

As you can see, this file contains all sorts of useful information about the author of the module, licensing stuff, development status, etc. More importantly for us now, it contains information about which libraries are required for *installing* the library, and which are required for *testing* the library. This means, that if you want to run the test suite, all the libraries listed in the `tests_require` list will be installed, but if somebody runs `pip install django-foobar`, only the libraries which we actually depends on, listed in `install_requires` will be installed (if not already installed).

The important thing here is to know what library versions are the *absolute minimum* for us to have installed for the library to work. In other words, you should never (or at least rarely) lock library versions in this file, but rather specify that we require version `>=` (larger than or equal to) some version number. So in the example above, we promise that our code will work with any Django version above v1.7.0.

Note: Keep in mind that you should consider the requirements you specify in this file to be a kind of contract between you as the maintainer and the people which are going to use your code.

You need to write tests in order to fulfill the contract!

Read more about `setup.py`.

3.2 Setting up the test runner

3.2.1 Running nose tests

In the `setup.py` file above, we have specified a `test_suite` attribute which points to `tests.runtests.start`. Create a new file called `runtests.py` in the `tests` project directory, and enter the following code:

```
#!/usr/bin/env python
```



```

from __future__ import absolute_import, print_function, unicode_literals

import os
import sys
import nose

def start(argv=None):
    sys.exitfunc = lambda: sys.stderr.write("Shutting down...\n")

    if argv is None:
        argv = [
            "nosetests", "--cover-branches", "--with-coverage",
            "--cover-erase", "--verbose",
            "--cover-package=django_foobar",
        ]

    nose.run_exit(argv=argv, defaultTest=os.path.abspath(os.path.dirname(__file__)))

if __name__ == "__main__":
    start(sys.argv)

```

3.2.2 Bootstrapping Django for the test runner

In order to let the test runner and setup script to be able to run Django tests, we need to initialize Django before running the tests. Enter the following code into `tests/__init__.py`.

```

from __future__ import absolute_import, unicode_literals
import os

test_runner = None
old_config = None

os.environ["DJANGO_SETTINGS_MODULE"] = "tests.settings"

import django
if hasattr(django, "setup"):
    django.setup()

def setup():
    global test_runner
    global old_config

    # If you want to support Django 1.5 and older, you need
    # this try-except block.
    try:
        from django.test.runner import DiscoverRunner
        test_runner = DiscoverRunner()
    except ImportError:
        from django.test.simple import DjangoTestSuiteRunner
        test_runner = DjangoTestSuiteRunner()

    test_runner.setup_test_environment()
    old_config = test_runner.setup_databases()

def teardown():
    test_runner.teardown_databases(old_config)
    test_runner.teardown_test_environment()

```

3.3 Running the test suite

Alright, now we have a good setup for running tests! We will no longer be writing the tests inside the `django_foobar` module, so you can delete the `django_foobar/tests.py` file.

Try running the test suite by running the following command from the working directory:

```
(django-foobar):~/django-foobar$ python setup.py test
running test
running egg_info
writing pbr to django_foobar.egg-info/pbr.json
writing django_foobar.egg-info/PKG-INFO
writing top-level names to django_foobar.egg-info/top_level.txt
writing dependency_links to django_foobar.egg-info/dependency_links.txt
writing requirements to django_foobar.egg-info/requirements.txt
reading manifest file 'django_foobar.egg-info/SOURCES.txt'
writing manifest file 'django_foobar.egg-info/SOURCES.txt'
running build_ext
```

Name	Stmts	Miss	Branch	BrMiss	Cover	Missing
django_foobar	0	0	0	0	100%	
django_foobar.admin	1	1	0	0	0%	1
django_foobar.migrations	0	0	0	0	100%	
django_foobar.models	1	1	0	0	0%	1
TOTAL	2	2	0	0	0%	

```
Ran 0 tests in 0.406s

OK
```

As you can see, the setup script builds an egg for us and runs the test suite against it. As we haven't written any actual test cases yet, it will print out the test coverage and report that zero tests were run.

3.4 Writing a test case

Start by creating a new file in the `tests` directory called `test_views.py`. The test runner will pick up any file prefixed with `test_` and run the test cases inside it.

We will write a test case for the `DeepThoughtView` we created earlier. Open the `tests/test_views.py` file and enter the following code inside:

```
# -*- coding: utf-8 -*-

from __future__ import absolute_import, unicode_literals

from django.core.urlresolvers import reverse
from django.test import TestCase

class DeepThoughtTestCase(TestCase):

    def test_deepthought_view(self):
        response = self.client.get(reverse("django_foobar_deepthought"))
        self.assertEqual(response.content, b"42")
```

Alright, lets try running the test suite again!

```
(django-foobar):~/django-foobar$ python setup.py test
running test
running egg_info
writing pbr to django_foobar.egg-info/pbr.json
writing django_foobar.egg-info/PKG-INFO
writing top-level names to django_foobar.egg-info/top_level.txt
writing requirements to django_foobar.egg-info/requirements.txt
writing dependency_links to django_foobar.egg-info/dependency_links.txt
reading manifest file 'django_foobar.egg-info/SOURCES.txt'
writing manifest file 'django_foobar.egg-info/SOURCES.txt'
running build_ext
test_deepthought_view (tests.test_views.DeepThoughtTestCase) ... ok
```

Name	Stmts	Miss	Branch	BrMiss	Cover	Missing
django_foobar	0	0	0	0	100%	
django_foobar.admin	1	1	0	0	0%	1
django_foobar.migrations	0	0	0	0	100%	
django_foobar.models	1	1	0	0	0%	1
django_foobar.views	6	0	0	0	100%	
TOTAL	8	2	0	0	75%	

```
Ran 1 test in 0.349s
```

Sweet, the test runner picked up our test case and reported success!

3.5 Testing multiple environments

Well, that's good and all, but at the current state, we only know that our code works as it should on the setup we're using for our development. What if somebody is running Python v2.7 with Django v1.6? Would it work as it should? We don't really know, so let's find out!

3.5.1 Tox to the rescue!

Tox is a tool for automate testing in Python. It does so by reading a `tox.ini` file where we specify the environments we want to test, and it will create a brand new virtual environment for that setup and run the test suite against it.

Let's start by installing `tox` and add it to our `requirements.txt` file.

```
(django-foobar):~/django-foobar$ pip install tox
(django-foobar):~/django-foobar$ echo "tox" >> requirements.txt
```

Create a new file `tox.ini` in the projects working directory and paste the content below.

```
[tox]
envlist =
    py27-django1.6,
    py34-django1.8

[django1.6]
deps =
    Django>=1.6,<1.7
```

```
[django1.8]
deps =
    Django>=1.8,<1.9

[testenv]
commands =
    python {toxiniidir}/setup.py test

[testenv:py27-django1.6]
basepython = python2.7
deps =
    {[django1.6]deps}

[testenv:py34-django1.8]
basepython = python3.4
deps =
    {[django1.8]deps}
```

As you can see we have a `[tox]` block in which we defines a list of environments we want to test. Next, we define some blocks for the different Django versions we want to test. The `[testenv]` block defines the command we want to run, and finally we combine the `[testenv]` with the different version blocks in order to fire up the test runner.

Note: You need to have all the python interpreters you want to test with installed on your system. If you don't have Python 2.7 installed, the Python 2.7 tests above will fail!

Now we have two different environments we can test: Python 2.7 with Django 1.6, and Python 3.4 with Django 1.8. Let's try to fire it up!

```
(django-foobar):~/django-foobar$ tox
<snip>
...
...
</snip>
py27-django1.6: commands succeeded
py34-django1.8: commands succeeded
congratulations :)
```

Great! We are now confident that our code works flawlessly with these configurations.

3.6 Continuous Integration

[Continuous Integration](#) is nice. Alright that was yet another opinionated statement from me, but it makes me happy to see my builds go green =)

[Travis](#) is my weapon of choice because all my code goes to Github, and Travis integrates very easily. Create a new file `.travis.yml` in your projects working directory and paste the following code.

```
language: python
python:
  - "2.7"

cache:
  directories:
    - pip_download_cache

before_install:
```

```

- mkdir -p $PIP_DOWNLOAD_CACHE

install:
- pip install tox
- pip install -e .

script:
- tox -e $TOX_ENV

env:
global:
- PIP_DOWNLOAD_CACHE="pip_download_cache"
matrix:
- TOX_ENV=py27-django1.6
- TOX_ENV=py34-django1.8

notifications:
email: false

```

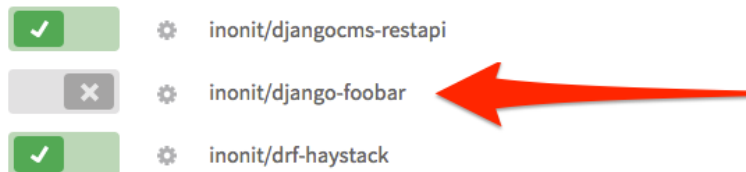
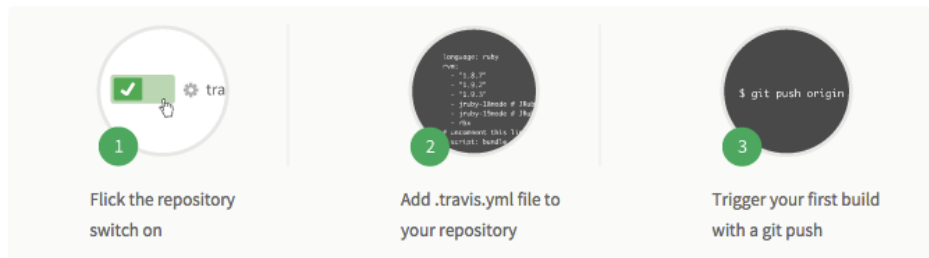
As you can read out from the configuration, we'll be using `tox` to run the test suite on Travis as well. You should be able to recognize the `tox` environments we created earlier in the `travis` configuration above.

Time to make the build!

1. Head over to [Travis](#) and hit the “Sign in with Github” button in the top right corner.
2. Navigate to your profile and locate the repository list. There should be a flip switch next to the repository name. Flip it to “On”.

Inonit AS Sync last synced 3 minutes ago

We're only showing your public repositories. You can find your private projects on [travis-ci.com](#).



Once the flip is on, Travis will start an automated build whenever a new commit is pushed to the Github repository.


3. Commit and push new stuff to Github.

```



(django-foobar):~/django-foobar$ git add *
(django-foobar):~/django-foobar$ git commit -m "a brand new commit"
(django-foobar):~/django-foobar$ git push -u origin master


```






4. Enjoy the build!

inonit / django-foobar  build passing





Current Branches Build History Pull Requests > Build #2 Settings

master a brand new commit
 Rolf Håvard Blindheim authored and committed

2 passed
 Commit f0a3299
 Compare e6cb5d1..f0a3299
 ran for 1 min 14 sec
 less than a minute ago 

Build Jobs

✓ # 2.1	 </> Python: 2.7	 TOX_ENV=py27-django1.6	🕒 29 sec
✓ # 2.2	 </> Python: 2.7	 TOX_ENV=py34-django1.8	🕒 45 sec

Holy cow, it worked!

Documenting your library

If you have followed this guide step-by-step, you should already have a `docs/` directory in the projects working directory containing a Sphinx project. If you skipped that step, now is a good time to go back and create a [Documentation](#) project.

Inside the `docs/` directory you'll find a file called `conf.py`. This is the configuration file which Sphinx use to build the documentation. The most important settings should already be added in there by the `sphinx-quickstart` command we used earlier, so we'll just leave that alone.

4.1 Writing documentation

Most of the time, the reason for writing 3rd party modules is that we believe we have found a use case for some kind of functionality that is so common and/or generic that we can put in some time and effort now, to save time in the long run. Chances are, that other people might find our module useful too, but we need to tell them (and our future selves) how to use it.

It is therefore crucial that we write good documentation and explain how to use the code.

4.1.1 Index

Inside the `docs/` directory you'll find a file called `index.rst`. Open it in a text editor and it will look something like this:

```
.. test documentation master file, created by
   sphinx-quickstart on Tue Sep  1 15:01:48 2015.
   You can adapt this file completely to your liking, but it should at least
   contain the root `toctree` directive.

Welcome to django-foobar documentation!
=====

Contents:

.. toctree::
   :maxdepth: 2

Indices and tables
=====

* :ref:`genindex`
```

```
* :ref:`modindex`
* :ref:`search`
```

Above the Content, start writing a short introduction for your library. It could be a few lines about the intended use case, target audience, a poem or something else.

As an exercise, let's document how we can run the test suite for our package. Create a new file called `01-testing.rst` next to the `index.rst` file and modify the `.. toctree::` block so it looks like this:

```
.. toctree::
   :maxdepth: 2

   01-testing
```

4.1.2 01-testing

Open the newly created `01-testing.rst` file, and write the following code:

```
Running the test suite
=====

In order to run the test suite for ``django-foobar`` run the following command from the
cli:

.. code-block:: none

    $ python setup.py test

You should see the tests complete without errors!
```

Let's go through some of the directives we have used above:

- **Headlines** Sphinx is pretty good at determining headline levels, but you **must** be consistent within the file. It is also pretty common to keep a consistent style for the entire documentation project.

```
=====
Headline 1
=====

Headline 2
=====

Headline 3
-----

Headline 4
.....
```

- `.. code-block:: none` This defines a new code block. The `none` argument provided indicates the syntax highlighting we want to use. When demonstrating commands to be run, `none` is usually a good choice, but Sphinx supports a lot of different syntax highlighting, like `python`, `ruby`, `xml` and so on.
- **Inline blocks** By using double backticks `“this is an inline block”`, you can create `inline` blocks. They are nice for highlighting important words, class names, and so on.

For a more in-depth guide for writing Sphinx docs, see [this guide](#) and the [Sphinx primer](#).

4.2 Compile the documentation

Okay, we have now written some documentation. Let's compile it to HTML. Run the following command from the docs/ directory.

```
$ make html
sphinx-build -b html -d _build/doctrees . _build/html
Running Sphinx v1.3.1
loading pickled environment... done
building [mo]: targets for 0 po files that are out of date
building [html]: targets for 1 source files that are out of date
updating environment: 0 added, 1 changed, 0 removed
reading sources... [100%] 04-documentation
looking for now-outdated files... none found
pickling environment... done
checking consistency... done
preparing documents... done
writing output... [100%] index
generating indices... genindex
writing additional pages... search
copying static files... done
copying extra files... done
dumping search index in English (code: en) ... done
dumping object inventory... done
build succeeded.
```

Open the docs/_build/html/index.html file in your web browser.

Table Of Contents

Welcome to [django-foobar's documentation!](#)

- [To The River](#)

[Indices and tables](#)

This Page

[Show Source](#)

Quick search

Go

Enter search terms or a module, class or function name.

Welcome to django-foobar's documentation!

To The River

Fair river! in thy bright, clear flow
Of crystal, wandering water,
Thou art an emblem of the glow
Of beauty – the unhidden heart – The playful mazineess of art
In old Alberto's daughter;
But when within thy wave she looks –
Which glistens then, and trembles –
Why, then, the prettiest of brooks
Her worshipper resembles;
For in my heart, as in thy stream,
Her image deeply lies –
The heart which trembles at the beam
Of her soul-searching eyes.

by Edgar Allan Poe

Contents:

- [Running the test suite](#)

Indices and tables

- [Index](#)
- [Module Index](#)
- [Search Page](#)

4.3 Read the Docs

[Read the Docs](#) is a service for hosting documentation. It's free for open source projects, so we'll set it up to host the documentation for us. It can be activated as a Github hook like the Travis CI hook, which means that every time we commit changes to the Github repository the documentation will be rebuilt.

1. Head over to *Read the Docs* <<https://readthedocs.org/>> and create an account
2. From the drop down menu in the top right corner, click "Add project"
3. Click the "Import from Github" button

Import a GitHub project

Showing your GitHub projects. [Sync your GitHub projects](#) to update them.

inonit/djangocms-snug

Create

inonit/django-foobar

Create

4. Fill in some details

Project Details

To import a project, start by entering a few details about your repository. More advanced project options can be configured if you select **Edit advanced project options**.

Name:

django-foobar

Repository URL:

https://github.com/inonit/djangi

Hosted documentation repository URL

Repository type:

Git

Edit advanced project options:

☐

Next

5. Authenticate with Github if asked, and accept adding the webhook

GitHub webhook activated

Search django-foobar


Versions

latest

Build a version
latest

Repository
<https://github.com/inonit/django-foobar>

Last Built
No builds yet

Owners


Done =)

Pack and deploy your masterpiece!

Finally, we're ready to release our little gem into the wild. In order to let users install the library using `pip`, we must upload it to the [Python Package Index](#). Head over there and create an account.

A screenshot of the PyPI registration form. It contains five input fields: 'Username:' with 'johndoe', 'Password:' with '.....', 'Confirm:' with '.....', 'Email Address:' with 'john.doe@example.com', and 'PGP Key ID (optional):' which is empty. To the right of the PGP field is the text '(This identifies a PGP or GPG key)'. Below the fields is a 'Register' button.

Username: johndoe

Password:

Confirm:

Email Address: john.doe@example.com

PGP Key ID (optional): (This identifies a PGP or GPG key)

Register

Next, create a new file called `.pypirc` in your `$HOME` directory. Put the following content in it. Replace the username and password with the ones you used when creating your account.

```
# this tells distutils what package indexes you can push to
[distutils]
index-servers =
    pypi

[pypi]
repository: https://pypi.python.org/pypi
username: johndoe
password: <your secret password>
```

5.1 Push the library to PyPi

Note: This is only meant as a guide line for how to push packages to PyPi. Anybody with a valid account can push content there, however; it might not be such a good idea to push all sorts of useless libraries out there.

The `django-foobar` library written in this guide is a pretty useless library, so it's no reason to actually push it! Use this as a reference when you have written a real library =)

5.1.1 The manifest file

The manifest file specifies what files we want to include in the source distribution we're going to upload to PyPi. Go ahead and create a new file called `MANIFEST.in` in your projects working directory and put the following content in it.

```
include README.rst
include LICENSE.txt
include requirements.txt
include tox.ini
recursive-include docs Makefile *.rst *.py *.bat
recursive-include tests *.py *.json *.txt
```

This is a pretty generic template but will be sufficient for most Django projects. You can read more about the [MANIFEST.in](#) file [here](#).

5.1.2 Registering your library

If you want to reserve the name for your library, but are not quite ready to distribute it yet you can register the name on PyPi. We're going to use the `setup.py` script we wrote earlier so make sure your current path is in the projects working directory.

```
(django-foobar):~/django-foobar$ python setup.py register
running register
running egg_info
writing pbr to django_foobar.egg-info/pbr.json
writing requirements to django_foobar.egg-info/requirements.txt
writing django_foobar.egg-info/PKG-INFO
writing top-level names to django_foobar.egg-info/top_level.txt
writing dependency_links to django_foobar.egg-info/dependency_links.txt
reading manifest file 'django_foobar.egg-info/SOURCES.txt'
writing manifest file 'django_foobar.egg-info/SOURCES.txt'
running check
Registering django-foobar to https://pypi.python.org/pypi
Server response (200): OK
```

Your library are now registered in the Python Package Index!

5.1.3 Uploading your library

In order to upload a source distribution, enter the following command:

```
(django-foobar):~/django-foobar$ python setup.py sdist upload
running sdist
running egg_info
writing pbr to django_foobar.egg-info/pbr.json
writing top-level names to django_foobar.egg-info/top_level.txt
writing django_foobar.egg-info/PKG-INFO
writing dependency_links to django_foobar.egg-info/dependency_links.txt
writing requirements to django_foobar.egg-info/requirements.txt
reading manifest file 'django_foobar.egg-info/SOURCES.txt'
writing manifest file 'django_foobar.egg-info/SOURCES.txt'
running check
creating django-foobar-1.0
creating django-foobar-1.0/django_foobar
creating django-foobar-1.0/django_foobar.egg-info
```

```
making hard links in django-foobar-1.0...
hard linking README.rst -> django-foobar-1.0
hard linking setup.py -> django-foobar-1.0
hard linking django_foobar/__init__.py -> django-foobar-1.0/django_foobar
hard linking django_foobar/admin.py -> django-foobar-1.0/django_foobar
hard linking django_foobar/models.py -> django-foobar-1.0/django_foobar
hard linking django_foobar/tests.py -> django-foobar-1.0/django_foobar
hard linking django_foobar/views.py -> django-foobar-1.0/django_foobar
hard linking django_foobar.egg-info/PKG-INFO -> django-foobar-1.0/django_foobar.egg-info
hard linking django_foobar.egg-info/SOURCES.txt -> django-foobar-1.0/django_foobar.egg-info
hard linking django_foobar.egg-info/dependency_links.txt -> django-foobar-1.0/django_foobar.egg-info
hard linking django_foobar.egg-info/not-zip-safe -> django-foobar-1.0/django_foobar.egg-info
hard linking django_foobar.egg-info/pbr.json -> django-foobar-1.0/django_foobar.egg-info
hard linking django_foobar.egg-info/requirements.txt -> django-foobar-1.0/django_foobar.egg-info
hard linking django_foobar.egg-info/top_level.txt -> django-foobar-1.0/django_foobar.egg-info
Writing django-foobar-1.0/setup.cfg
creating dist
Creating tar archive
removing 'django-foobar-1.0' (and everything under it)
running upload
Submitting dist/django-foobar-1.0.tar.gz to https://pypi.python.org/pypi
Server response (200): OK
```

Congratulations, you have just released a 3rd party Django module on PyPi. It may be a little while before the library is visible for pip.

Conclusion

During this guide, we have been through creating a working environment, how to structure your project, setting up a test environment suitable for doing Continuous Integration development , writing documentation and deploying the library to PyPi.

This is by no means a definite solution for all kind of projects, and a lot of topics covered here can be done in different ways. As you gain experience, you'll probably find things in this guide that doesn't suite your style of development or other things that work better for you or the project you are working on.

But by now you should be able to a project with a basic workflow that allows other people to contribute.

Have fun!

Indices and tables

- `genindex`
- `modindex`
- `search`