Django

Documentation

Writing your first patch for Django

Introduction

Interested in giving back to the community a little? Maybe you've found a bug in Django that you'd like to see fixed, or maybe there's a small feature you want added.

Contributing back to Django itself is the best way to see your own concerns addressed. This may seem daunting at first, but it's really pretty simple. We'll walk you through the entire process, so you can learn by example.

Who's this tutorial for?

See also

If you are looking for a reference on how to submit patches, see the Submitting patches documentation.

For this tutorial, we expect that you have at least a basic understanding of how Django works. This means you should be comfortable going through the existing tutorials on writing your first Django app. In addition, you should have a good understanding of Python itself. But if you don't, Dive Into Python is a fantastic (and free) online book for beginning Python programmers.

Those of you who are unfamiliar with version control systems and Trac will find that this tutorial and its links include just enough information to get started. However, you'll probably want to read some more about these different tools if you plan on contributing to Django regularly.

For the most part though, this tutorial tries to explain as much as possible, so that it can be of use to the widest audience.

Where to get help:

If you're having trouble going through this tutorial, please post a message to django-developers or drop by #django-dev on irc.freenode.net to chat with other Django users who might be able to help.

What does this tutorial cover?

We'll be walking you through contributing a patch to Django for the first time. By the end of this tutorial, you should have a basic understanding of both the tools and the processes involved. Specifically, we'll be covering the following:

- Installing Git.
- How to download a development copy of Django.
- Running Django's test suite.
- Writing a test for your patch.
- Writing the code for your patch.
- Testing your patch.
- Generating a patch file for your changes.
- Where to look for more information.

Once you're done with the tutorial, you can look through the rest of Django's documentation on contributing. It contains lots of great information and is a must read for anyone who'd like to become a regular contributor to Django. If you've got questions, it's probably got the answers.

Python 3 required!

This tutorial assumes you are using Python 3. Get the latest version at Python's download page or with your operating system's package manager.

For Windows users

When installing Python on Windows, make sure you check the option “Add python.exe to Path”, so that it is always available on the command line.

Code of Conduct

As a contributor, you can help us keep the Django community open and inclusive. Please read and follow our Code of Conduct.

Installing Git

For this tutorial, you’ll need Git installed to download the current development version of Django and to generate patch files for the changes you make.

To check whether or not you have Git installed, enter **`git`** into the command line. If you get messages saying that this command could not be found, you’ll have to download and install it, see Git’s download page.

For Windows users

When installing Git on Windows, it is recommended that you pick the “Git Bash” option so that Git runs in its own shell. This tutorial assumes that’s how you have installed it.

If you’re not that familiar with Git, you can always find out more about its commands (once it’s installed) by typing **`git help`** into the command line.

Getting a copy of Django’s development version

The first step to contributing to Django is to get a copy of the source code. From the command line, use the **`cd`** command to navigate to the directory where you’ll want your local copy of Django to live.

Download the Django source code repository using the following command:

```
$ git clone https://github.com/django/django.git
```

Now that you have a local copy of Django, you can install it just like you would install any package using **`pip`**. The most convenient way to do so is by using a *virtual environment* (or virtualenv) which is a feature built into Python that allows you to keep a separate directory of installed packages for each of your projects so that they don’t interfere with each other.

It’s a good idea to keep all your virtualenvs in one place, for example in **`.virtualenvs/`** in your home directory. Create it if it doesn’t exist yet:

```
$ mkdir ~/.virtualenvs
```

Now create a new virtualenv by running:

```
$ python3 -m venv ~/.virtualenvs/djangodev
```


The path is where the new environment will be saved on your computer.

For Windows users

Using the built-in **`venv`** module will not work if you are also using the Git Bash shell on Windows, since activation scripts are only created for the system shell (**`.bat`**) and PowerShell (**`.ps1`**). Use the **`virtualenv`** package instead:

```
$ pip install virtualenv
$ virtualenv ~/.virtualenvs/djangodev
```

For Ubuntu users

On some versions of Ubuntu the above command might fail. Use the **`virtualenv`** package instead, first making sure you have **`pip3`**:

```
$ sudo apt-get install python3-pip
$ # Prefix the next command with sudo if it gives a permission denied error
$ pip3 install virtualenv
$ virtualenv --python=`which python3` ~/.virtualenvs/djangodev
```

The final step in setting up your virtualenv is to activate it:

```
$ source ~/.virtualenvs/djangodev/bin/activate
```

If the **`source`** command is not available, you can try using a dot instead:

```
$ . ~/.virtualenvs/djangodev/bin/activate
```

For Windows users

To activate your virtualenv on Windows, run:

```
$ source ~/virtualenvs/djangodev/Scripts/activate
```

You have to activate the virtualenv whenever you open a new terminal window. virtualenvwrapper is a useful tool for making this more convenient.

Anything you install through **`pip`** from now on will be installed in your new virtualenv, isolated from other environments and system-wide packages. Also, the name of the currently activated virtualenv is displayed on the command line to help you keep track of which one you are using. Go ahead and install the previously cloned copy of Django:

```
$ pip install -e /path/to/your/local/clone/django/
```

The installed version of Django is now pointing at your local copy. You will immediately see any changes you make to it, which is of great help when writing your first patch.

For this tutorial, we'll be using ticket #24788 as a case study, so we'll rewind Django's version history in git to before that ticket's patch was applied. This will allow us to go through all of the steps involved in writing that patch from scratch, including running Django's test suite.

Keep in mind that while we'll be using an older revision of Django's trunk for the purposes of the tutorial below, you should always use the current development revision of Django when working on your own patch for a ticket!

Note

The patch for this ticket was written by Paweł Marczewski, and it was applied to Django as commit 4df7e8483b2679fc1cba3410f08960bac6f51115. Consequently, we'll be using the revision of Django just prior to that, commit 4ccfc4439a7add24f8db4ef3960d02ef8ae09887.

Navigate into Django's root directory (that's the one that contains **`django`**, **`docs`**, **`tests`**, **`AUTHORS`**, etc.). You can then check out the older revision of Django that we'll be using in the tutorial below:

```
$ git checkout 4ccfc4439a7add24f8db4ef3960d02ef8ae09887
```

Running Django's test suite for the first time

When contributing to Django it's very important that your code changes don't introduce bugs into other areas of Django. One way to check that Django still works after you make your changes is by running Django's test suite. If all the tests still pass, then you can be reasonably sure that your changes haven't completely broken Django. If you've never run Django's test suite before, it's a good idea to run it once beforehand just to get familiar with what its output is supposed to look like.

Before running the test suite, install its dependencies by first **`cd`**-ing into the Django **`tests/`** directory and then running:

```
$ pip install -r requirements/py3.txt
```

If you encounter an error during the installation, your system might be missing a dependency for one or more of the Python packages. Consult the failing package's documentation or search the Web with the error message that you encounter.

Now we are ready to run the test suite. If you're using GNU/Linux, Mac OS X or some other flavor of Unix, run:

```
$ ./runtests.py
```

Now sit back and relax. Django's entire test suite has over 9,600 different tests, so it can take anywhere from 5 to 15 minutes to run, depending on the speed of your computer.

While Django's test suite is running, you'll see a stream of characters representing the status of each test as it's run. **E** indicates that an error was raised during a test, and **F** indicates that a test's assertions failed. Both of these are considered to be test failures. Meanwhile, **x** and **s** indicated expected failures and skipped tests, respectively. Dots indicate passing tests.

Skipped tests are typically due to missing external libraries required to run the test; see Running all the tests for a list of dependencies and be sure to install any for tests related to the changes you are making (we won't need any for this tutorial). Some tests are specific to a particular database backend and will be skipped if not testing with that backend. SQLite is the database backend for the default settings. To run the tests using a different backend, see Using another settings module.

Once the tests complete, you should be greeted with a message informing you whether the test suite passed or failed. Since you haven't yet made any changes to Django's code, the entire test suite **should** pass. If you get failures or errors make sure you've followed all of the previous steps properly. See Running the unit tests for more information. If you're using Python 3.5+, there will be a couple failures related to deprecation warnings that you can ignore. These failures have since been fixed in Django.

Note that the latest Django trunk may not always be stable. When developing against trunk, you can check Django's continuous integration builds to determine if the failures are specific to your machine or if they are also present in Django's official builds. If you click to view a particular build, you can view the "Configuration Matrix" which shows failures broken down by Python version and database backend.

Note

For this tutorial and the ticket we're working on, testing against SQLite is sufficient, however, it's possible (and sometimes necessary) to run the tests using a different database.

Writing some tests for your ticket

In most cases, for a patch to be accepted into Django it has to include tests. For bug fix patches, this means writing a regression test to ensure that the bug is never reintroduced into Django later on. A regression test should be written in such a way that it will fail while the bug still exists and pass once the bug has been fixed. For patches containing new features, you'll need to include tests which ensure that the new features are working correctly. They too should fail when the new feature is not present, and then pass once it has been implemented.

A good way to do this is to write your new tests first, before making any changes to the code. This style of development is called test-driven development and can be applied to both entire projects and single patches. After writing your tests, you then run them to make sure that they do indeed fail (since you haven't fixed that bug or added that feature yet). If your new tests don't fail, you'll need to fix them so that they do. After all, a regression test that passes regardless of whether a bug is present is not very helpful at preventing that bug from reoccurring down the road.

Now for our hands-on example.

Writing some tests for ticket #24788

Ticket #24788 proposes a small feature addition: the ability to specify the class level attribute **`prefix`** on Form classes, so that:

```
[…] forms which ship with apps could effectively namespace themselves such
that N overlapping form fields could be POSTed at once and resolved to the
correct form.
```

In order to resolve this ticket, we'll add a **`prefix`** attribute to the **`BaseForm`** class. When creating instances of this class, passing a prefix to the **`__init__()`** method will still set that prefix on the created instance. But not passing a prefix (or passing **`None`**) will use the class-level prefix. Before we make those changes though, we're going to write a couple tests to verify that our modification functions correctly and continues to function correctly in the future.

Navigate to Django's **`tests/forms_tests/tests/`** folder and open the **`test_forms.py`** file. Add the following code on line 1674 right before the **`test_forms_with_null_boolean`** function:

```
def test_class_prefix(self):
    # Prefix can be also specified at the class level.
    class Person(Form):
        first_name = CharField()
        prefix = 'foo'

    p = Person()
    self.assertEqual(p.prefix, 'foo')

    p = Person(prefix='bar')
    self.assertEqual(p.prefix, 'bar')
```

This new test checks that setting a class level prefix works as expected, and that passing a **`prefix`** parameter when creating an instance still works too.

But this testing thing looks kinda hard...

If you've never had to deal with tests before, they can look a little hard to write at first glance. Fortunately, testing is a *very* big subject in computer programming, so there's lots of information out there:

- A good first look at writing tests for Django can be found in the documentation on Writing and running tests.
- Dive Into Python (a free online book for beginning Python developers) includes a great introduction to Unit Testing.
- After reading those, if you want something a little meatier to sink your teeth into, there's always the Python **`unittest`** documentation.

Running your new test

Remember that we haven't actually made any modifications to **`BaseForm`** yet, so our tests are going to fail. Let's run all the tests in the **`forms_tests`** folder to make sure that's really what happens. From the command line, **`cd`** into the Django **`tests/`** directory and run:

```
$ ./runtests.py forms_tests
```

If the tests ran correctly, you should see one failure corresponding to the test method we added. If all of the tests passed, then you'll want to make sure that you added the new test shown above to the appropriate folder and class.

Writing the code for your ticket

Next we'll be adding the functionality described in ticket #24788 to Django.

Writing the code for ticket #24788

Navigate to the **`django/django/forms/`** folder and open the **`forms.py`** file. Find the **`BaseForm`** class on line 72 and add the **`prefix`** class attribute right after the **`field_order`** attribute:

```
class BaseForm(object):
    # This is the main implementation of all the Form logic. Note that this
    # class is different than Form. See the comments by the Form class for
    # more information. Any improvements to the form API should be made to
    # *this* class, not to the Form class.
    field_order = None
    prefix = None
```

Verifying your test now passes

Once you're done modifying Django, we need to make sure that the tests we wrote earlier pass, so we can see whether the code we wrote above is working correctly. To run the tests in the **`forms_tests`** folder, **`cd`** into the Django **`tests/`** directory and run:

```
$ ./runtests.py forms_tests
```

Oops, good thing we wrote those tests! You should still see one failure with the following exception:

```
AssertionError: None != 'foo'
```

We forgot to add the conditional statement in the **`__init__`** method. Go ahead and change **`self.prefix = prefix`** that is now on line 87 of **`django/forms/forms.py`**, adding a conditional statement:

```
if prefix is not None:
    self.prefix = prefix
```

Re-run the tests and everything should pass. If it doesn't, make sure you correctly modified the **`BaseForm`** class as shown above and copied the new test correctly.

Running Django's test suite for the second time

Once you've verified that your patch and your test are working correctly, it's a good idea to run the entire Django test suite just to verify that your change hasn't introduced any bugs into other areas of Django. While successfully passing the entire test suite doesn't guarantee your code is bug free, it does help identify many bugs and regressions that might otherwise go unnoticed.

To run the entire Django test suite, **cd** into the Django **tests/** directory and run:

```
$ ./runtests.py
```

As long as you don't see any failures, you're good to go.

Writing Documentation

This is a new feature, so it should be documented. Add the following section on line 1068 (at the end of the file) of **django/docs/ref/forms/api.txt**:

```
The prefix can also be specified on the form class::

    >>> class PersonForm(forms.Form):
    ...     ...
    ...     prefix = 'person'

.. versionadded:: 1.9

    The ability to specify ``prefix`` on the form class was added.
```

Since this new feature will be in an upcoming release it is also added to the release notes for Django 1.9, on line 164 under the "Forms" section in the file **docs/releases/1.9.txt**:

```
* A form prefix can be specified inside a form class, not only when
  instantiating a form. See :ref:`form-prefix` for details.
```

For more information on writing documentation, including an explanation of what the **versionadded** bit is all about, see Writing documentation. That page also includes an explanation of how to build a copy of the documentation locally, so you can preview the HTML that will be generated.

Generating a patch for your changes

Now it's time to generate a patch file that can be uploaded to Trac or applied to another copy of Django. To get a look at the content of your patch, run the following command:

```
$ git diff
```

This will display the differences between your current copy of Django (with your changes) and the revision that you initially checked out earlier in the tutorial.

Once you're done looking at the patch, hit the **q** key to exit back to the command line. If the patch's content looked okay, you can run the following command to save the patch file to your current working directory:

```
$ git diff > 24788.diff
```

You should now have a file in the root Django directory called **24788.diff**. This patch file contains all your changes and should look this:

```
diff --git a/django/forms/forms.py b/django/forms/forms.py
index 509709f..d1370de 100644
--- a/django/forms/forms.py
+++ b/django/forms/forms.py
@@ -75,6 +75,7 @@ class BaseForm(object):
     # information. Any improvements to the form API should be made to *this*
     # class, not to the Form class.
     field_order = None
+    prefix = None

     def __init__(self, data=None, files=None, auto_id='id_%s', prefix=None,
                  initial=None, error_class=ErrorList, label_suffix=None,
@@ -83,7 +84,8 @@ class BaseForm(object):
         self.data = data or {}
         self.files = files or {}
         self.auto_id = auto_id
-        self.prefix = prefix
+        if prefix is not None:
+            self.prefix = prefix
         self.initial = initial or {}
         self.error_class = error_class
         # Translators: This is the default suffix added to form field labels
diff --git a/docs/ref/forms/api.txt b/docs/ref/forms/api.txt
index 3bc39cd..008170d 100644
--- a/docs/ref/forms/api.txt
+++ b/docs/ref/forms/api.txt
@@ -1065,3 +1065,13 @@ You can put several Django forms inside one ``<form>`` tag. To give each
     >>> print(father.as_ul())
     <li><label for="id_father-first_name">First name:</label> <input type="text" name="father-first_name" id="id_father-first_name"
/></li>
     <li><label for="id_father-last_name">Last name:</label> <input type="text" name="father-last_name" id="id_father-last_name" />
</li>
+
+The prefix can also be specified on the form class::
+
+    >>> class PersonForm(forms.Form):
+    ...     ...
+    ...     prefix = 'person'
+
+.. versionadded:: 1.9
+
+    The ability to specify ``prefix`` on the form class was added.
diff --git a/docs/releases/1.9.txt b/docs/releases/1.9.txt
index 5b58f79..f9bb9de 100644
--- a/docs/releases/1.9.txt
+++ b/docs/releases/1.9.txt
@@ -161,6 +161,9 @@ Forms
   :attr:`~django.forms.Form.field_order` attribute, the ``field_order``
   constructor argument , or the :meth:`~django.forms.Form.order_fields` method.

+* A form prefix can be specified inside a form class, not only when
+  instantiating a form. See :ref:`form-prefix` for details.
+
 Generic Views
 ^^^^^^^^^^^^^


diff --git a/tests/forms_tests/tests/test_forms.py b/tests/forms_tests/tests/test_forms.py
index 690f205..e07fae2 100644
--- a/tests/forms_tests/tests/test_forms.py
+++ b/tests/forms_tests/tests/test_forms.py
@@ -1671,6 +1671,18 @@ class FormsTestCase(SimpleTestCase):
         self.assertEqual(p.cleaned_data['last_name'], 'Lennon')
         self.assertEqual(p.cleaned_data['birthday'], datetime.date(1940, 10, 9))

+    def test_class_prefix(self):
+        # Prefix can be also specified at the class level.
+        class Person(Form):
+            first_name = CharField()
+            prefix = 'foo'
+
+        p = Person()
+        self.assertEqual(p.prefix, 'foo')
+
+        p = Person(prefix='bar')
+        self.assertEqual(p.prefix, 'bar')
+
     def test_forms_with_null_boolean(self):
```

```
        # NullBooleanField is a bit of a special case because its presentation (widget)
        # is different than its data. This is handled transparently, though.
```

So what do I do next?

Congratulations, you've generated your very first Django patch! Now that you've got that under your belt, you can put those skills to good use by helping to improve Django's codebase. Generating patches and attaching them to Trac tickets is useful, however, since we are using git - adopting a more git oriented workflow is recommended.

Since we never committed our changes locally, perform the following to get your git branch back to a good starting point:

```
$ git reset --hard HEAD
$ git checkout master
```

More information for new contributors

Before you get too into writing patches for Django, there's a little more information on contributing that you should probably take a look at:

- You should make sure to read Django's documentation on claiming tickets and submitting patches. It covers Trac etiquette, how to claim tickets for yourself, expected coding style for patches, and many other important details.
- First time contributors should also read Django's documentation for first time contributors. It has lots of good advice for those of us who are new to helping out with Django.
- After those, if you're still hungry for more information about contributing, you can always browse through the rest of Django's documentation on contributing. It contains a ton of useful information and should be your first source for answering any questions you might have.

Finding your first real ticket

Once you've looked through some of that information, you'll be ready to go out and find a ticket of your own to write a patch for. Pay special attention to tickets with the "easy pickings" criterion. These tickets are often much simpler in nature and are great for first time contributors. Once you're familiar with contributing to Django, you can move on to writing patches for more difficult and complicated tickets.

If you just want to get started already (and nobody would blame you!), try taking a look at the list of easy tickets that need patches and the easy tickets that have patches which need improvement. If you're familiar with writing tests, you can also look at the list of easy tickets that need tests. Just remember to follow the guidelines about claiming tickets that were mentioned in the link to Django's documentation on claiming tickets and submitting patches.

What's next?

After a ticket has a patch, it needs to be reviewed by a second set of eyes. After uploading a patch or submitting a pull request, be sure to update the ticket metadata by setting the flags on the ticket to say "has patch", "doesn't need tests", etc, so others can find it for review. Contributing doesn't necessarily always mean writing a patch from scratch. Reviewing existing patches is also a very helpful contribution. See Triaging tickets for details.

Getting Started with Django

Team Organization

Django Software Foundation

Code of Conduct

Diversity statement

Get Involved

Join a Group

Contribute to Django

Submit a Bug

Report a Security Issue

Follow Us

GitHub

Twitter

News RSS

Django Users Mailing List