django-facetools Documentation

Release 0.2.0

Eric Palakovich Carr

April 04, 2013

CONTENTS

1 Introduction **1**

2 Table of Contents **3**

2.1 Installing Django Facetools 3

2.2 Writing your first Django app...for Facebook! 4

2.3 Integrating and Testing Facebook Open Graph 12

3 Indices and tables **17**

CHAPTER

ONE

INTRODUCTION

Django Facetools provides a set of features to ease development of Facecbook canvas apps in a Django project.

Features:

- Replacement `url` template tag as well as the `reverse` function that convert an internal url into it's facebook canvas equivalent (ex: http://my_app.mycompany.com/canvas/my_view to https://apps.facebook.com/my_app/my_view)
- Ability to define facebook test users, their permissions, and their initial friends per app. The management command `sync_facebook_test_users` lets you recreate your test users in your facebook app with one call.
- `FacebookTestCase` can be used in place of Django's `TestCase`. Just specify a test user's name, much like a fixture, and the test client will mock Facebook requests to your canvas app, complete with a valid signed request for the specified test user.
- *New in version 0.2:* `FacebookTestCase` can also have the test client's signed request

set manually with the new `set_client_signed_request` method. * Integration with other facebook django packages, supporting the following (with more to come):

- Fandjango (https://github.com/jgorset/fandjango)

CHAPTER

TWO

TABLE OF CONTENTS

2.1 Installing Django Facetools

2.1.1 Install Dependencies

Django Facetools has been tested on Python 2.6 and 2.7. It's been ran using the following packages:

- Django >= 1.3.1
- south >= 0.7.3
- requests >= 0.7.3

To install these dependencies, you can use `pip`:

```
$ pip install django
$ pip install south
$ pip install requests
```

2.1.2 Install Django Facetools

For the latest stable version (recommended), use `pip` or `easy_install`:

```
$ pip install django-facetools
```

Alternatively, you can also download the latest development version from http://github.com/bigsassy/django-facetools and run the installation script:

```
$ python setup.py install
```

or use `pip`:

```
$ pip install -e git://github.com/bigsassy/django-facetools#egg=django-facetools
```

2.1.3 Configure Django

- In your project settings, add `facetools` to the `INSTALLED_APPS`:

```
INSTALLED_APPS = (
    # ... your other apps here
    'facetools',
)
```

- Also set the FACEBOOK_APPLICATION_ID, FACEBOOK_APPLICATION_SECRET_KEY, FACEBOOK_CANVAS_URL. and FACEBOOK_CANVAS_PAGE settings:

```
FACEBOOK_APPLICATION_ID = '301572769893123'
FACEBOOK_APPLICATION_SECRET_KEY = '[insert your secret key]'
FACEBOOK_CANVAS_URL = 'https://myapp.mycompany.com/canvas/'
FACEBOOK_CANVAS_PAGE = 'http://apps.facebook.com/myapp'
```

2.2 Writing your first Django app...for Facebook!

Converting existing Django apps to work as a Facebook app can be very simple. We're going to see how simple by converting the official Django tutorial app to work on facebook. You can view the tutorial at:

https://docs.djangoproject.com/en/1.3/intro/tutorial01/

Here's how we're going to modify the poll app:

- Add tests for the original views
- Convert the app so it works as a Facebook canvas app
- Let users post their vote to their walls
- Let users invite their friends to vote in the poll
- Add more tests to cover the new features

We'll be using *django-facetools* along with *fandjango* to make this happen.

2.2.1 Get the example app

Download the app and it's dependencies

First things first. Clone this example app's starting point from github:

```
$ git clone git://github.com/bigsassy/facetools-example-start.git
$ cd facetools-example-start
```

Optionally, you can create a virtualenv for this app:

```
$ virtualenv --no-site-packages virenv
$ source virenv/bin/activate
```

Next, you'll want to install the requirements for facetools:

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

Now let's create the database:

```
$ cd mysite
$ python manage.py syncdb
```

And finally, let's start the development server:

```
$ python manage.py runserver
```

2.2.2 Setup the example app

Add some data to the application

Let's create some test data.

1. Go to http://localhost:8000/admin/
2. Click the *+Add* button for Polls
3. Set the question to *What's up?*
4. Click the *Show* link next to Date Information. Click the *Today* and *Now* link for *Date published*.
5. Enter the choices *Not much*, and *The sky*. Set each choice's *Votes* field to 0.
6. Click *Save and add another*
7. Repeat, but set the question to *What's going down?*, with the choices *Not much*, and *My cholesterol*, once again with 0 votes each.
8. Click the *Save* button.

Add names to our urls

To keep things DRY, we're going to add names to all of our urls. Change *polls/urls.py* to look like the following:

```
from django.conf.urls.defaults import patterns, include, url
from django.views.generic import DetailView, ListView
from polls.models import Poll

urlpatterns = patterns('',
    url(r'^$',
        ListView.as_view(
            queryset=Poll.objects.order_by('-pub_date')[:5],
            context_object_name='latest_poll_list',
            template_name='polls/index.html'
        ), name='poll_index'),
    url(r'^(?P<pk>\d+)/$',
        DetailView.as_view(
            model=Poll,
            template_name='polls/detail.html'
        ), name='poll_detail'),
    url(r'^(?P<pk>\d+)/results/$',
        DetailView.as_view(
            model=Poll,
            template_name='polls/results.html'
        ), name='poll_results'),
    url(r'^(?P<poll_id>\d+)/vote/$', 'polls.views.vote', name="poll_vote"),
)
```

Now we can use the *reverse* function and *url* template tag to get a url without hardcoding paths. Let's update our templates now, replacing the lines with hardcocded urls with the following:

```
# In index.html
<a href="{% url poll_detail poll.id %}">{{ poll.question }}</a>

# In results.html
<a href="{% url poll_detail poll.id %}">Vote again?</a>
```

```
# In detail.html
<form action="{% url poll_vote poll_id=poll.id %}" method="post">
```

Manually test the application

Ok, let's run the app through it's paces. It's a simple polling app where you can view a list of polls, vote in any of the polls, and view the results of the poll.

Check the list page first by going to http://localhost:8000/polls/. You should see a bulleted list with a two polls, called *What's up?* and *What's going down?*.

Click on the *What's up?* poll to see its detail page. We should see the poll's title again, two radio buttons with the options *Not much* and *The sky*, and finally a vote button. Chooese *The sky* and click vote.

This should take you to a result page, once again showing the poll's title, followed by a bulleted list of the two choices *Not much* and *The sky* with their votes, and a link to vote again.

2.2.3 Add tests for the original views

So with everything working, we're going to write some tests using Django's test client for everything we just manually did. We'll be revisiting these tests when we use facetools to inject facebook test users into the Django test client.

Fist, let's create a data fixture to run out tests against. Stop the *runserver* command and run the following:

```
$ python manage.py dumpdata polls --indent=4 > polls/fixtures/polls.json
```

Open the new file, *polls/fixtures/polls.json*, change the number of votes for poll choice "The sky" from 1 to 0, and save it. Now we have a nice set of test data.

Now open *polls/tests.py* and make make it look like this:

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

from polls.models import Poll

class ServerSideTests(TestCase):
    fixtures = ['polls.json']

    def test_index(self):
        pass

    def test_detail(self):
        pass

    def test_voting(self):
        pass

    def test_results(self):
        pass
```

We're going to write some tests to ensure the website is functioning correctly on the server. Let's get some of the simple ones out of the way, only checking for templates and valid context variables:

```
def test_index(self):
    # The view should return a valid page with the correct template
    response = self.client.get(reverse("poll_index"))
```

```
        self.assertEquals(200, response.status_code)
        self.assertTemplateUsed(response, "polls/index.html")
        self.assertIn('latest_poll_list', response.context)

        # The template should get all the polls in the database
        expected_polls = [p.pk for p in response.context['latest_poll_list']]
        actual_polls = [p.pk for p in Poll.objects.all()]
        self.assertEquals(set(expected_polls), set(actual_polls))

    def test_detail(self):
        expected_poll = Poll.objects.get(pk=1)

        # The view should return a valid page with the correct template
        response = self.client.get(reverse("poll_detail", args=[expected_poll.pk]))
        self.assertEquals(200, response.status_code)
        self.assertTemplateUsed(response, "polls/detail.html")
        self.assertIn('poll', response.context)

        # The poll should be the correct poll
        actual_poll = response.context['poll']
        self.assertEquals(expected_poll.pk, actual_poll.pk)

    def test_results(self):
        expected_poll = Poll.objects.get(pk=1)

        # The view should return a valid page with the correct template
        response = self.client.get(reverse("poll_detail", args=[expected_poll.pk]))
        self.assertEquals(200, response.status_code)
        self.assertTemplateUsed(response, "polls/detail.html")
        self.assertIn('poll', response.context)

        # The poll should be the correct poll
        actual_poll = response.context['poll']
        self.assertEquals(expected_poll.pk, actual_poll.pk)
```

Next we'll write a test to put the voting feature through its paces:

```
    def test_voting(self):
        poll = Poll.objects.get(pk=1)

        # Test initial data assumptions
        self.assertEquals(0, poll.choice_set.get(pk=1).votes)
        self.assertEquals(0, poll.choice_set.get(pk=2).votes)

        # Test voting a bunch of times
        self.vote_and_assert(poll, 1, {1: 1, 2: 0})
        self.vote_and_assert(poll, 1, {1: 2, 2: 0})
        self.vote_and_assert(poll, 2, {1: 2, 2: 1})
        self.vote_and_assert(poll, 1, {1: 3, 2: 1})
        self.vote_and_assert(poll, 2, {1: 3, 2: 2})

    def vote_and_assert(self, poll, choice_pk, expected_choice_votes):
        response = self.client.post(reverse("poll_vote",
            kwargs={'poll_id': poll.pk}),
            {
                'poll_id': poll.pk,
                'choice': choice_pk
            }
```

```
    )

    # Make sure after voting the user is redirected to the results page
    expected_redirect_url = reverse("poll_results", args=[poll.pk])
    self.assertEquals(302, response.status_code)
    self.assertTrue(response['Location'].endswith(expected_redirect_url))

    # Make sure that the votes in the database reflect the new vote
    for choice_pk,expected_votes in expected_choice_votes.items():
        choice = poll.choice_set.get(pk=choice_pk)
        self.assertEquals(expected_votes, choice.votes)
```

Time to make sure our tests are working. Assuming your still in the *mysite* directory on the command line, do the following:

```
$ python manage.py test polls
```

And with that we have pretty good coverage of our views (front-end is another story). Now, let's get to the fun stuff.

2.2.4 Convert the app into a Facebook canvas app

Create the facebook app

With that, it's time to start using Facebook. So let's transform this Django app into a Facebook app.

Before we do anything, you should familiarize yourself with Facebook canvas apps: http://developers.facebook.com/docs/guides/canvas/.

Next, go the the tutorial at http://developers.facebook.com/docs/appsonfacebook/tutorial/ and complete the sections *Creating your App* and *Configuring your App*, using the following values for your app settings:

- App Display Name: Whatever you want
- App name space: Whatever you want
- Contact e-mail: Your e-mail address
- App Domain: Leave this blank for this tutorial
- Category: Leave it on Other

In the *Select how your app integrates with Facebook* section, click the checkmark next to *App on Facebook*. Next enter *https://localhost:8000/canvas/* for the *Canvas URL*. Leave the secure canvas canvas url blank, as we'll be running this site as a development site.

Next, click the *Save changes* button to create your app! You'll get a warning saying Secure Canvas URL will be required. To fix this requirement, we're going to turn on sandbox mode. Sandbox mode makes the application invisible to facebook users not explicitly added to the app.

To setup sandbox mode, click on *Advanced* under *Settings* in the left navigation, click the *Enabled* radio for *Sandbox Mode*, and click the *Save changes* button at the bottom of the page.

Seperate your canvas app from the admin

Next, we want to make sure the admin section of our site isn't availalble from the facebook app. We're going to modify the root *urls.py* in the *mysite* directory so the polls app is reached from /canvas/ (e.g. http:localhost:8000/canvas/polls/poll/1/)/ We're going to change one line from this:

```
url(r'^polls/', include('polls.urls')),
```

To this:

```
url(r'^canvas/polls/', include('polls.urls')),
```

Now, let's run out tests to make sure everything is still working. Close the *runserver* command if it's still running and do the following:

```
$ python manage.py test polls
```

Sure enough, all out tests still pass even after changing our url structure. This is because we used the *reverse* function in our tests to get each view's url by name, instead of hardcoding them. That's how we keep things DRY in Django.

Try out your Facebook app!

Ok, go to your app url. First, bring your server back up:

```
$ python manage.py runserver
```

Then open polls via your facebook canvas app in your browser. The url will be something like https://apps.facebook.com/your-app-namespace/polls/. You should be greeted with a CSRF token error page. This happens because facebook sends a POST to our app with the signed request you read about earlier in the facebook docs.

This causes Django to complain because we have the *CsrfViewMiddleware* installed, which looks for a CSRF token in any post request to prevent cross-site request forgery attacks. Time to bring out Fandjango.

Installing and configuring Fandjango

Assuming you installed the requirments file, Fandjango should already available in your virtualenv.

Setting up Fandjango is easy. In *settings.py*:

1. Add *fandjango* to your *INSTALLED_APPS*

2. Add *fandjango.middleware.FacebookMiddleware* to your *MIDDLEWARE_CLASSES*, before the CSRF middleware. *MIDDLEWARE_CLASSES* should end up looking like this:

```
MIDDLEWARE_CLASSES = (
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'fandjango.middleware.FacebookMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
)
```

3. Add the following settings at the bottom of the file. You can find your values at https://developers.facebook.com/apps/:

```
FACEBOOK_APPLICATION_ID = "Your App ID / API Key here"
FACEBOOK_APPLICATION_SECRET_KEY = "Your App Secret here"
FACEBOOK_APPLICATION_NAMESPACE = "your-app-namespace"
```

4. Finally, run *syncdb* again to add the Fandjango tables:

```
$ python manage.py syncdb
```

Let's bring your server back up:

```
$ python manage.py runserver
```

And let's visit your page again. You should see the poll page in all it's glory. Now visit http://localhost:8000/admin. Your admin page is also available and seperate from the facebook page.

2.2.5 Using Facetools to fix iframe problems

Ok, so now we have our Django app running as a Facebook canvas app. But there are a few problems that persist.

1. The links for each poll read like http://localhost:8000/canvas/polls/1 instead of https://apps.facebook.com/facetools-example/polls/1.
2. When you click on a poll it goes to the page, but the browsers address bar doesn't update.
3. When you vote in the poll you get an error, yet the admin shows the vote is getting counted. This is because the vote gets handled in our *polls.views.vote* view, but then fails when the view tries to redirect in an iframe.

We're going to solve all these problems using Facetools. Do the following:

1. Add 'facetools' to your *INSTALLED_APPS* in the *settings.py* file.
2. Add the following settings at the bottom of the file. You can find your values at https://developers.facebook.com/apps/:

```
# existing settings you've already entered, and are required by facetools
FACEBOOK_APPLICATION_ID = "Your App ID / API Key here"
FACEBOOK_APPLICATION_SECRET_KEY = "Your App Secret here"

# New settings you're adding now
FACEBOOK_CANVAS_PAGE = "Your canvas page here"
FACEBOOK_CANVAS_URL = "The value from Secure Canvas URL here"
```

3. Add *{% load facetools_tags %}* to the top of all three template *.html files.
4. Rename *url* to *facebook_url* Add a target of *_top* to each anchor tag in our templates:

```
# In index.html
<a href="{% facebook_url poll_detail poll.id %}" target="_top">{{ poll.question }}</a>

# In results.html
<a href="{% facebook_url poll_detail poll.id %}" target="_top">Vote again?</a>
```

5. Change the *vote* view in *polls/views.py* so *HttpResponseRedirect* is now *facebook_redirect*, and that is imported from *facetools.url*. It should look like this:

```
# ... other imports ...
from facetools.url import facebook_redirect

def vote(request, poll_id):
    p = get_object_or_404(Poll, pk=poll_id)
    try:
        selected_choice = p.choice_set.get(pk=request.POST['choice'])
    except (KeyError, Choice.DoesNotExist):
        # Redisplay the poll voting form.
        return render_to_response('polls/detail.html', {
            'poll': p,
```

```
            'error_message': "You didn't select a choice.",
            }, context_instance=RequestContext(request))
    else:
        selected_choice.votes += 1
        selected_choice.save()
        return facebook_redirect(reverse('poll_results', args=(p.id,)))
```

Save your changes, do *runserver* again if it's not running, and go to the index page again in your browser. Now the url for each poll points to the the page in facebook. And when you submit your vote in a poll, you'll get redirected to the results page.

How did Facetools help?

The *facebook_url* tags automatically translate any url path that falls in the FACEBOOK_CANVAS_URL and translates it to it's facebook equivalent.

The *facebook_redirect* function applies the same logic, acting as a substitute for *HttpResponseRedirect* and *django.urlresolvers.reverse*. It replaces the redirect resonse object with a regular html result. The html consists of a redirect via javascript. It'll look something like this:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
        "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
    <script type="text/javascript">
        top.location.href="https://apps.facebook.com/your-app-namespace/polls/1/results/";
    </script>
</head>
<body>

</body>
</html>
```

Check out tests

Once last thing, we should check that our tests still pass. Go back to the *mysite* directory on the command line and run your tests:

```
$ python manage.py test polls
```

You should get one AssertionError stating 302 != 200. This is where we used to check that POSTing a vote would result in a http status code for redirects. Since we're now forced to use javascript to redirect the client, we're getting a regular 200 status code instead.

Update the second code block in the *vote_and_assert* method of the *ServerSideTests* class in the *polls/tests.py* file from this:

```
# Make sure after voting the user is redirected to the results page
expected_redirect_url = reverse("poll_results", args=[poll.pk])
self.assertEquals(302, response.status_code)
self.assertIn(expected_redirect_url, response.content)
```

to this:

```
# Make sure after voting the user is redirected to the results page
expected_redirect_url = facebook_reverse("poll_results", args=[poll.pk])
```

```
self.assertEquals(200, response.status_code)
self.assertIn(expected_redirect_url, response.content)
```

and add the following import to *polls/tests.py*:

```
from facetools.url import facebook_reverse
```

Now when you run the tests again they all should pass.

2.3 Integrating and Testing Facebook Open Graph

2.3.1 Force Facebook users to install app and grant permissions

Now let's add a feature that actually leverages Facebook's Open Graph. We're going to add a welcome message to the poll index page. To get access to the user's name, we'll need Facebook users to install the app and grant permissions to us.

To do this with Fandjango is easy. We need to add a decorator on each of our view functions. We can also add permissions our app requires in our *settings.py*.

First, add this with the other Facebook settings in the *settings.py* file:

```
FACEBOOK_APPLICATION_INITIAL_PERMISSIONS = [
    'read_stream',
    'user_birthday',
]
```

This will make Fandjango ask users their permission to read from their stream and get their birthday (a.k.a. their age). Next we add the decorator to each view function. Change *polls/views.py* like so:

```
# ... other imports ...
from fandjango.decorators import facebook_authorization_required

@facebook_authorization_required
def vote(request, poll_id):
    # ... the function body ...
```

And change *polls/urls.py* to look like this:

```
# ... other imports ...
from fandjango.decorators import facebook_authorization_required

urlpatterns = patterns('',
    url(r'^$',
        facebook_authorization_required(
            ListView.as_view(
                queryset=Poll.objects.order_by('-pub_date')[:5],
                context_object_name='latest_poll_list',
                template_name='polls/index.html')
        ), name='poll_index'),
    url(r'^(?P<pk>\d+)/$',
        facebook_authorization_required(
            DetailView.as_view(
                model=Poll,
                template_name='polls/detail.html')
        ), name='poll_detail'),
    url(r'^(?P<pk>\d+)/results/$',
```

```
        facebook_authorization_required(
            DetailView.as_view(
                model=Poll,
                template_name='polls/results.html')
        ), name='poll_results'),
    url(r'^(?P<poll_id>\d+)/vote/$', 'polls.views.vote', name="poll_vote"),
)
```

Now each view has the *facebook_authorization_required* decorator, which will look for a signed request either in POST data or in the user's cookies. If it's missing, it'll redirect the user to an authorization page to install your app and grant it the permissions you specify.

Finally, Facetools requires you add *facetools.middleware.FandjangoIntegrationMiddleware* to MIDDLEWARE_CLASSES, which should look this this when you're done:

```
MIDDLEWARE_CLASSES = (
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'fandjango.middleware.FacebookMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'facetools.middleware.FandjangoIntegrationMiddleware',
)
```

2.3.2 Adding Facebook open graph data to a template

Change the template under *polls/templates/polls/index.html* so it looks like this:

```
{% load facetools_tags %}

<h1>Hello, {{ request.facebook.user.full_name }}!</h1>

{% if latest_poll_list %}
<ul>
    {% for poll in latest_poll_list %}
    <li><a href="{% facebook_url poll_detail poll.id %}" target="_top">{{ poll.question }}</a></li>
    {% endfor %}
</ul>
{% else %}
<p>No polls are available.</p>
{% endif %}
```

And we'll need to add a template context processor so we can access the request object in our templates. Add this to the bottom of your *settings.py* file:

```
TEMPLATE_CONTEXT_PROCESSORS = (
    "django.contrib.auth.context_processors.auth",
    "django.core.context_processors.debug",
    "django.core.context_processors.i18n",
    "django.core.context_processors.media",
    "django.core.context_processors.static",
    "django.contrib.messages.context_processors.messages",
    "django.core.context_processors.request",
)
```

Now when you go to the index page, you should be greeted by name. Fandjango attaches the facebook object to every request. Assuming a valid signed request was found, the facebook object will have a two member variables,

signed_request and *user*.

The *signed_request* variable is a dict with the signed request data. The *user* variable is a Fandjango *User* instance, containing useful data, like *user.full_name*, *user.gender*, and *user.email*, along with a property called *graph*. The *graph* property is an instance of *Facepy.GraphAPI*, which gives you an API for this user's open graph data.

2.3.3 Testing Facebook open graph data

Facebook provides a mechanism for defining test users for an app without creating fake accounts in facebook. You can read up about it here: http://developers.facebook.com/docs/test_users/

Facetools provides a means of managing your test users so that they can be created and used automatically in your tests across one or environments (development vs staging).

We're going to update our tests to ensure the open graph data is working correctly on our site, with a little help from Facetools

Setup Facebook Test Users in Facetools

First we'll deine our facebook test user. Create the file *polls/facebook_test_users.py* with the following content:

```
facebook_test_users = [
    {
        'name': 'Sam Samson',
        'installed': True,
        'permissions': [
            'read_stream',
            'user_birthday',
        ]
    }
]
```

This is how we define test users in Facetools. Each Django app can provide its own set of users in a *facebook_test_users.py* file. The file either needs to define a list named *facebook_test_users*, consisting of dicts following the above format, or a callable of the same name that also returns a list of those dicts.

What's nice is that we defined this test user once, and now we can recreate him anywhere with a management command we'll see in a bit. This is particularly nice if you have a facebook app for each of your environemnts (e.g. myfacebookapp-dev, myfacebookapp-staging).

Next, we'll create the test user on facebook using the *sync_facebook_test_users* management command. From the command line in the *mysite* directory, run:

```
$ python manage.py sync_facebook_test_users polls
```

Once this finishes running, you'll have a test user defined on facebook, and a test fixture with the TestUser data at *polls/fixtures/facetools_test_users.json*. This test fixture is created or re-created everytime the command is run, which is particularly useful for updating the fixture's access token when they go stale.

Extra - We won't use this for the tutorial, but you can also define a test users friends among other test users. It works like this:

```
facebook_test_users = [
    {
        'name': 'Sam Samson',
        'installed': True,
        'permissions': [
            'read_stream',
```

```
            'user_birthday',
        ]
    },
    {
        'name': 'Laura Ensminger',
        'installed': True,
        'permissions': [
            'read_stream',
            'user_birthday',
        ],
        'friends': ['Sam Samson']
    }
]
```

If you ran *sync_facebook_test_users* now, you would get two test users that are friends with each other on Facebook.

Update unit tests to test graph data

We're going to update our test for index now. Update *polls/tests.py* so it look like this:

```
# ... other imports ...#
from facetools.test import FacebookTestCase

class ServerSideTests(FacebookTestCase):
    fixtures = ['polls.json']
    facebook_test_user = "Sam Samson"

    def test_index(self):
        # The view should return a valid page with the correct template
        response = self.client.get(reverse("poll_index"))
        self.assertEquals(200, response.status_code)
        self.assertTemplateUsed(response, "polls/index.html")
        self.assertIn('latest_poll_list', response.context)

        # The template should get all the polls in the database
        expected_polls = [p.pk for p in response.context['latest_poll_list']]
        actual_polls = [p.pk for p in Poll.objects.all()]
        self.assertEquals(set(expected_polls), se't(actual_polls))

        # The response content should have our teset user's name
        assertIn(self.test_user.name, response.content)

    # ... rest of file ...#
```

We've done a few things here. First, we've imported a FacebookTestCase, and then changed the parent class of ServerSideTests from TestCase to FacebookTestCase. Using this class will make the Django test client mock a request as if made from the facebook canvas page, giving you access to a signed request of the specified test user, in this case "Sam Samson". It'll also supply us with *self.test_user*, the *TestUser* object of "Sam Samson".

Integrate Fandjango into the tests

Next we'll need to hook into Facetools' signals. One is for syncing any of your user data models with the up-to-date (thanks to *sync_facebook_test_users*) test user data (in particular their access tokens). THe second is to update the test client to include the signed request, e.g. via a cookie.

If you are using Fandjango then we can use functions provided by Facetools. Add the following code at the top of *polls/models.py*:

```
# ... other imports ...#

from facetools.signals import sync_facebook_test_user, setup_facebook_test_client
from facetools.integrations import fandjango
sync_facebook_test_user.connect(fandjango.sync_facebook_test_user)
setup_facebook_test_client(fandjango.setup_facebook_test_client)

# ... rest of file ...#
```

With this, we'll have a Fandjango User record created for our test user before each test is ran, complete with the proper acesss token. And we'll also have a signed request for the test user added to a cookie that Fandjango sets when a user logins on the real Facebook canvas site.

Now if you go ahead and run the tests again everything should pass.

Manually manage Facebook Test Users in test cases

New in 0.2

Currently, our *ServerSideTests* test case should look like this:

> **class ServerSideTests(FacebookTestCase):** fixtures = ['polls.json'] facebook_test_user = "Sam Samson"
>
> # ... rest of test case ...#

In *0.2* you can now manually set the signed request for a *FacebookTestCase* test client, instead of using a *facebook_test_users.py* file, the *facebook_test_user* attribute, and the *sync_facebook_test_users* management command. Instead you can use the new *set_client_signed_request* method to set the test client's signed request.

The *set_client_signed_request* accepts a facebook user ID and an access token. This will cause the *setup_facebook_test_client* For example, you could modify the *ServerSideTests* test case to look like this:

> class ServerSideTests(FacebookTestCase):
>
> **def setUp(self):** facebook_id, oauth_token = your_function_for_getting_a_users_facebook_id_and_oauth_token()
> self.set_client_signed_request(facebook_id, oauth_token)
>
> # ... rest of test case ...#

This way you can manage creation and deletion of test users manually. This also works well if you don't need to automatically sync your test fixture data with real and currently active test user facebook ids and access tokens. This makes sense if your test coverage doesn't touch any open graph calls, or if you use a library to mock away open graph calls.

2.3.4 Wrap Up

And that's how you use Facetools and Fandjango together. Here are the key takeaways:

- Facetools will let you translate URLs for Facebook canvas pages with little effort
- Facetools makes it easy to create and keep Facebook test users in sync across multiple facebook apps (e.g. apps.facebook.com/myapp-dev, apps.facebook.com/myapp-staging)
- Facetools give you a test client that mocks Facebook's communication with your canvas app.
- Facetools provides a signal to sync your internal User models with your app's Facebook test users
- Facetools also provides a signal to view, modify, and use the signed request before each of your tests.
- And finally, Facetools comes with an integration package to work hand-in-hand

with Fandjango. Support for more Facebook packages coming soon.

CHAPTER

THREE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*