DJOAuth2 Documentation

Release 0.6.0

Peter Downs

Sep 27, 2017

Contents

CHAPTER 1

Important Links

- Source code: https://github.com/locu/djoauth2
- Documentation: http://djoauth2.readthedocs.org/
- Issue tracker: https://github.com/locu/djoauth2/issues
- Mailing list: https://groups.google.com/forum/#!forum/djoauth2

CHAPTER 2

What is DJOAuth2?

DJOAuth2 is an implementation of a *sane* subset of the OAuth 2 specification, which is described by the OAuth Website as

> An open protocol to allow secure authorization in a simple and standard method from web, mobile and desktop applications.

The goal of this implementation is to provide a well-structured Django application that can be easily installed to add OAuth 2.0 provider capability to existing projects. The official specification is broad, and allows for many different ways for clients and servers to interact with each other. This implementation is a secure subset of these interactions in order to make it as easy as possible to reap the benefits of OAuth without having to struggle with the more difficult parts of the spec.

OAuth, and this implementation, are best suited to solving the following problems:

- Allowing for fine-grained API control — you want your users to choose which applications have access to their data.
- Acting as an authentication server, allowing other sites to "Log in with <your app>".

CHAPTER 3

Why use DJOAuth2?

In the fall of 2012, when this project began, we read an article by Daniel Greenfield (better known as pydanny) criticizing the dearth of high-quality, open-source OAuth 2.0 provider implementations in Python. The article contains a wishlist of features for any OAuth implementation:

- Near turnkey solution
- Working code (duplicates above bullet but I'm making a point)
- Working tutorials
- Documentation
- Commented code
- Linted code
- Test coverage > 80%

This project aims to meet all of these goals, and in particular strives to be:

- Easy to add to existing Django projects, with few dependencies or requirements.
- Easy to understand, by virtue of high-quality documentation and examples.
- Functionally compliant with the official specification.
- Sane and secure by default — the specification allows for insecure behavior, which has been exploited in many existing implementations by programmers such as Egor Homakov.
- Well-documented and commented, in order to make it easy to understand how the implementation complies with the specification.
- Well-tested (see the coverage details on the first page of these docs!)

CHAPTER 4

What is implemented?

In order to best describe this implementation, we must first describe a few common terms used in the *OAuth specification*:

OAuth defines four roles:

resource owner An entity capable of granting access to a protected resource. When the resource owner is a person, it is referred to as an end-user.

resource server The server hosting the protected resources, capable of accepting and responding to protected resource requests using access tokens.

client An application making protected resource requests on behalf of the resource owner and with its authorization. The term "client" does not imply any particular implementation characteristics (e.g., whether the application executes on a server, a desktop, or other devices).

authorization server The server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization.

This implementation allows your application to act as a "resource server" and as an "authorization server". Your application's users are the "resource owners", and other applications which would like access to your users' data are the "clients".

The specification describes two types of clients, "confidential" and "public":

OAuth defines two client types, based on their ability to authenticate securely with the authorization server (i.e., ability to maintain the confidentiality of their client credentials):

confidential Clients capable of maintaining the confidentiality of their credentials (e.g., client implemented on a secure server with restricted access to the client credentials), or capable of secure client authentication using other means.

public Clients incapable of maintaining the confidentiality of their credentials (e.g., clients executing on the device used by the resource owner, such as an installed native application or a web browser-based application), and incapable of secure client authentication via any other means.

> The client type designation is based on the authorization server's definition of secure authentication and its acceptable exposure levels of client credentials. The authorization server SHOULD NOT make assumptions about the client type.

This implementation only supports "confidential" clients. Any web, mobile, or desktop application that acts as a client must also use some sort of secured server in order to protect its client credentials. Apps that are entirely native, or built entirely on the "client-side" of the web, are not supported.

The decisions that are most important to the security of your application are:

- The authorization endpoint will only return authorization codes, which can later be exchanged for access tokens.
- Password credentials grants, implicit grants, client credentials grants, and all extension grants are not supported.
- Public clients are not supported.
- Every client is required to register its `redirect_uri`.
- All authorization, token, and API requests are required to use TLS encryption in order to prevent credentials from being leaked to a third-party. In addition, the registered `redirect_uri` must also be secured with TLS.
- Clients are required to CSRF-protect their redirection endpoints.

These decisions have been made in an attempt to decrease the attack surface-area of the implementation. The specification has a great overview of security considerations that contains reasoning for many of these decisions.

In addition, we only support Bearer tokens in an effort to make interacting with the implementation as simple as possible for clients. This means no fiddling with MAC-signing or hashing!

CHAPTER 5

Quickstart Guide

This guide will help you get set up to use `djoauth2` with an existing Django project. The code repository also includes a finished example for comparison; that page includes instructions for setting it up.

Requirements

DJOAuth2 has been tested and developed with the following:

- Python 2.7
- Django 1.4+
- Django AppConf 0.6

DJOAuth2 uses South for migrations. For the Django 1.4.X series we support South version 0.7.6; for Django 1.5.X and 1.6.X we support South version 0.8.4.

Installation

1. Install the project with `pip`:

```
pip install djoauth2
```

Adding `djoauth2` to an existing application

First, add `djoauth2` to the `INSTALLED_APPS` list in your project's `settings.py`:

```
INSTALLED_APPS = [
  'django.contrib.auth',
  'django.contrib.contenttypes',
  'django.contrib.sessions',
  'django.contrib.sites',
  'django.contrib.messages',
  'django.contrib.staticfiles',
  'django.contrib.admin',
  'south',
  # ...
  # ... your other custom apps
  # ...
  'djoauth2',
]
```

If you're not already using *South*, make sure to also add `south` to the list of `INSTALLED_APPS`.

Optionally, for developing without SSL (**NOT for production code**), add the following setting to turn off `djoauth2`'s SSL-enforcement:

```
DJOAUTH2_SSL_ONLY = False
```

Do not set this to `False` in production code: SSL is mandated by the specification. This value is only designed to make it easier to *develop* with OAuth.

Install the models:

```
python manage.py syncdb
python manage.py migrate djoauth2
```

In Django 1.5+, `djoauth2` will respect **'your custom User model'_** if you have one configured (with the **'AUTH_USER_MODEL'_** setting.) In Django 1.4, or 1.5+ if you're not using a custom User model, the `djoauth2` models will link to the `django.contrib.auth.models.User` object.

Run the tests — they should all pass!

```
python manage.py test djoauth2
```

Now that we know that `djoauth2` works, it's time to set up the URL endpoints so that clients can make requests. Although the library handles all of the logic for us, we will have to set up some endpoints — to do so, we'll update our project's `urls.py` file and add an application to hold the endpoints. For the purposes of this demo we're going to call it `oauth2server`, but you could name it anything you'd like.

Here's what the `urls.py` file from our project should look like:

```
# coding: utf-8
from django.conf.urls import patterns, include, url
from django.contrib import admin


admin.autodiscover()

urlpatterns = patterns('',
    # Admin, for creating new Client and Scope objects. You can also create
    # these from the command line but it's easiest from the Admin.
    url(r'^admin/', include(admin.site.urls)),

    # The endpoint for creating and exchanging access tokens and refresh
    # tokens is handled entirely by the djoauth2 library.
    (r'^oauth2/token/$', 'djoauth2.views.access_token_endpoint'),

    # The authorization endpoint, a page where each "resource owner" will
    # be shown the details of the permissions being requested by the
    # "client".
    (r'^oauth2/authorization/$', 'oauth2server.views.authorization_endpoint'),

    # The page to show when Client redirection URIs are misconfigured or
    # invalid. This should be a nice, simple error page.
    (r'^oauth2/missing_redirect_uri/$', 'oauth2server.views.missing_redirect_uri'),

    # An access-protected API endpoint, which we'll define later.
    (r'^api/user_info/$', 'api.views.user_info'),
)
```

As you can see, it references an endpoint defined by `djoauth2` (the `access_token_endpoint`) and two others (`authorization_endpoint` and `missing_redirect_uri`) that we say exist in our `oauth2server` application. The `oauth2server` application only exists to define those two views — here's what the `views.py` file should look like:

```
# coding: utf-8
from django.shortcuts import render
from django.http import HttpResponse
from django.forms import Form

from djoauth2.authorization import make_authorization_endpoint


def missing_redirect_uri(request):
  """ Display an error message when an authorization request fails and has no
  valid redirect URI.

  The Authorization flow depends on recognizing the Client that is requesting
  certain permissions and redirecting the user back to an endpoint associated
  with the Client.  If no Client can be recognized from the request, or the
  endpoint is invalid for some reason, we redirect the user to a page
  describing that an error has occurred.
  """
  return HttpResponse(content="Missing redirect URI!")

authorization_endpoint = make_authorization_endpoint(
  # The URI of a page to show when a "client" makes a malformed or insecure
  # request and their registered redirect URI cannot be shown.  In general, it
  # should simply show a nice message describing that an error has occurred;
  # see the view definition above for more information.
  missing_redirect_uri='/oauth2/missing_redirect_uri/',

  # This endpoint is being dynamically constructed, but it also needs to know
  # the URI at which it is set up so that it can create forms and handle
  # redirects, so we explicitly pass it the URI.
  authorization_endpoint_uri='/oauth2/authorization/',

  # The name of the template to render to show the "resource owner" the details
  # of the "client's" request. See the documentation for more details on the
  # context used to render this template.
  authorization_template_name='oauth2server/authorization_page.html')
```

The template passed to the `make_authorization_endpoint` helper will be rendered with the following context:

- `form`: a Django `Form` that may hold data internal to the `djoauth2` application.

- `client`: The `djoauth2.models.Client` requesting access to the user's scopes.
- `scopes`: A list of `djoauth2.models.Scope`, one for each of the scopes requested by the client.
- `form_action`: The URI to which the form should be submitted – use this value in the `action=""` attribute on a `<form>` element.

The template in our example application is included below. Please note that it is important to include the `{{form}}` context — `djoauth2` may use this to hold information across authorization requests. Currently, the `user_action` values must be `"Accept"` and `"Decline"`.

```
{% if client.image_url %}
  <img src="{{client.image_url}}">
{% endif %}

<p>{{client.name}} is requesting access to the following scopes:</p>

<ul>
  {% for scope in scopes %}
  <li> <b>{{scope.name}}</b>: {{scope.description}} </li>
  {% endfor %}
</ul>


<form action="{{form_action}}" method="POST">
  {% csrf_token %}
  <div style="display: none;"> {{form}} </div>
  <input type="submit" name="user_action" value="Decline"/>
  <input type="submit" name="user_action" value="Accept"/>
</form>
```

And with that, all of the OAuth routes are implemented! All that's left is to set up an API endpoint that requires clients to have been authorized via OAuth — we referenced it in the URL conf by the name `api.views.user_info`. We're going to create a new application, `api`, to hold this view. In your own app, there's no need to create a new application, and you can simply use existing API views.

The `api/views.py` file:

```
# coding: utf-8
import json

from django.http import HttpResponse
from django.views.decorators.csrf import csrf_exempt

from djoauth2.decorators import oauth_scope


@csrf_exempt
@oauth_scope('user_info')
def user_info(access_token, request):
  """ Return basic information about a user.

  Limited to OAuth clients that have received authorization to the 'user_info'
  scope.
  """
  user = access_token.user
  data = {
      'username': user.username,
      'first_name': user.first_name,
      'last_name': user.last_name,
      'email': user.email}

  return HttpResponse(content=json.dumps(data),
                      content_type='application/json',
                      status=200)
```

(Any existing endpoint can be easily protected by our `@oauth_scope` decorator; just modify the signature so that it expects a `djoauth2.models.AccessToken` as the first argument. For more information, see the `djoauth2.decorators.oauth_scope` documentation.)

With our code all set up, we're ready to set up the DB and start the webserver:

```
python manage.py syncdb
python manage.py migrate
      python manage.py runserver 8080
```

Now, log in to the admin page and create a `Client` and a `Scope`. Set up the client so that the `redirect_uri` field is a valid URI under your control. While testing we often use URIs like `http://localhost:1111` that don't point to any server. The scope's `name` should be the same as that used to protect the `api.views.user_info` endpoint — in this case, `user_info`.

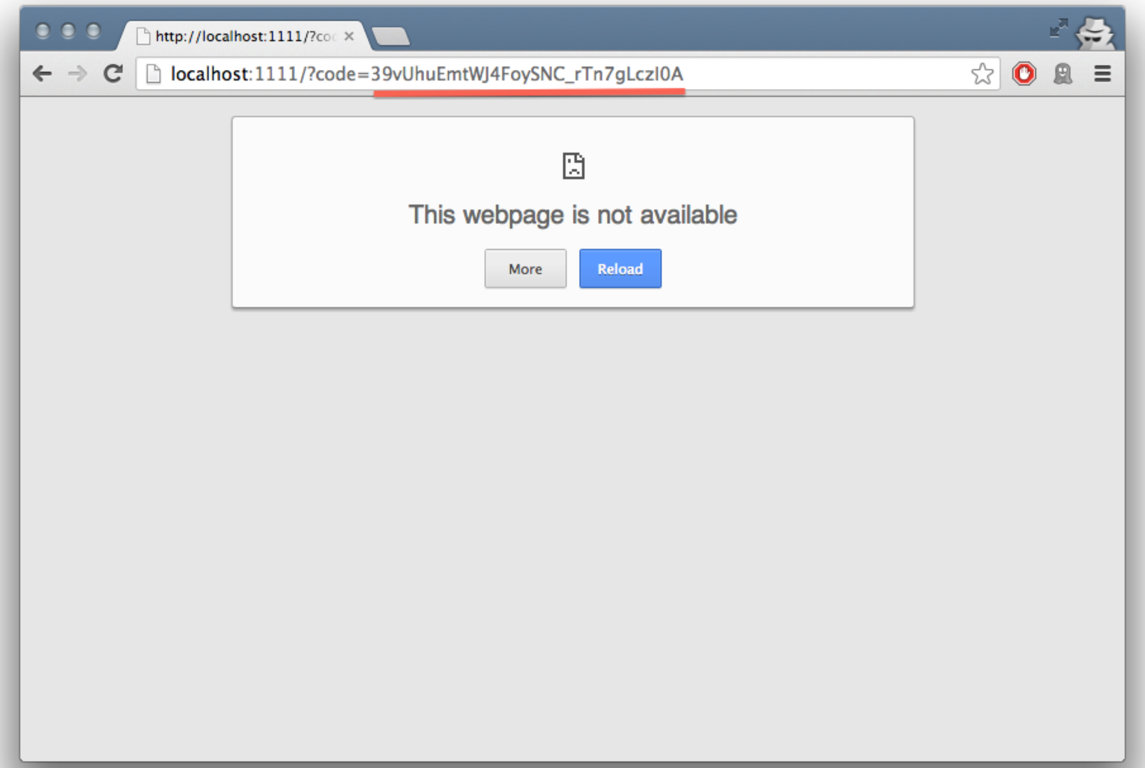
Interacting as a Client

We're ready to begin making requests as a client! In this example, we'll grant our client access to a scope, exchange the resulting authorization code for an access token, and then make an API request. This is adapted from our example project's `client_demo.py` script, which you can edit and run yourself. Go and check it out!

The first step is to grant our client authorization. Open a browser and visit the following URL:

```
http://localhost:8080/oauth2/authorization/?
  scope={the name of the scope you created}&
  client_id={the 'key' value from the Client you created}&
  response_type=code
```

If it worked, you should see the results of rendering your authorization template. If you confirm the request, you should be redirected to the registered client's `redirect_uri`. If you use a value like `http://localhost:1111`, your browser will show a "could not load this page" message. This is unimportant — what really matters is the `code` GET parameter in the URl. This is the value of the authorization code that was created by the server.


http://localhost:1111/?co
localhost:1111/?code=39vUhuEmtWJ4FoySNC_rTn7gLczl0A
This webpage is not available
More
Reload

We must now exchange this code for an access token. We do this by making a `POST` request like so:

```
POST http://localhost:8080/oauth2/token/ HTTP/1.1
Authorization: Basic {b64encode(client_id + ':' + client_secret)}

code={authorization code value}&grant_type=authorization_code
```

The `Authorization` header is used to identify us as the client that was granted the authorization code that we just received. The value should be the result of joining the client ID, a `:`, and the client secret, and encoding the resulting string with base 64. In Python, this might look like:

```
import requests
from base64 import b64encode
token_response = requests.post(
  'http://localhost:8080/oauth2/token/',
  data={
    'code': 'Xl4ryuwLJ6h2cTkW5K09aUpBQegmf8',
    'grant_type': 'authorization_code',
  },
  headers={
    'Authorization': 'Basic {}'.format(
        b64encode('{}:{}'.format(client_key, client_secret))),
  })
assert token_response.status_code == 200
```

This will return a JSON dictionary with the access token, access token lifetime, and (if available) a refresh token. Continuing the example from above:

```
import json

token_data = json.loads(token_response.content)
access_token = token_data['access_token']
refresh_token = token_data.get('refresh_token', None)
access_token_lifetime_seconds = token_data['expires_in']
```

With this access token, we can now make API requests on behalf of the user who granted us access! Again, continuing from above:

```
api_response = requests.post(
  'http://localhost:8080/api/user_info/',
  headers={
    'Authorization': 'Bearer {}'.format(token_data['access_token'])
  },
  data={})
assert api_response.status_code == 200
print api_response.content
# {"username": "exampleuser",
#  "first_name": "Example",
#  "last_name": "User",
#  "email": "exampleuser@locu.com"}
```

While the access token has not expired, you will be able to continue making API requests. Once it has expired, any API request will return an `HTTP 401 Unauthorized`. At that point, if you have a refresh token, you can exchange it for a new access token like so:

```
token_response = requests.post(
  'http://localhost:8080/oauth2/token/',
  data={
    'refresh_token': 'h9EY74_58aueZqHskUwVmMiTngcW3I',
    'grant_type': 'refresh_token',
  },
  headers={
    'Authorization': 'Basic {}'.format(
        b64encode('{}:{}'.format(client_key, client_secret))),
  })

assert token_response.status_code == 200

new_token_data = json.loads(token_response.content)
new_access_token = new_token_data['access_token']
new_refresh_token = new_token_data.get('refresh_token', None)
new_access_token_lifetime_seconds = new_token_data['expires_in']
```

As long as you have a refresh token, you can continue to exchange them for new access tokens. If your access token expires and you have lost the refresh token value, the refresh request fails, or you were never issued a refresh token, then you must begin again by redirecting the user to the authorization page.

CHAPTER 6

The `djoauth2` Code

`access_token` Module

`authorization` Module

`conf` Module

`decorators` Module

`errors` Module

exception `djoauth2.errors.`**`DJOAuthError`**

Base class for all OAuth-related errors.

`error_name` **= 'invalid_request'**

`status_code` **= 400**

`djoauth2.errors.`**`get_error_details`**`(`*error*`)`

Return details about an OAuth error.

Returns a mapping with two keys, `'error'` and `'error_description'`, that are used in all error responses described by the OAuth 2.0 specification. Read more at:

- http://tools.ietf.org/html/rfc6749
- http://tools.ietf.org/html/rfc6750

`models` Module

`views` Module

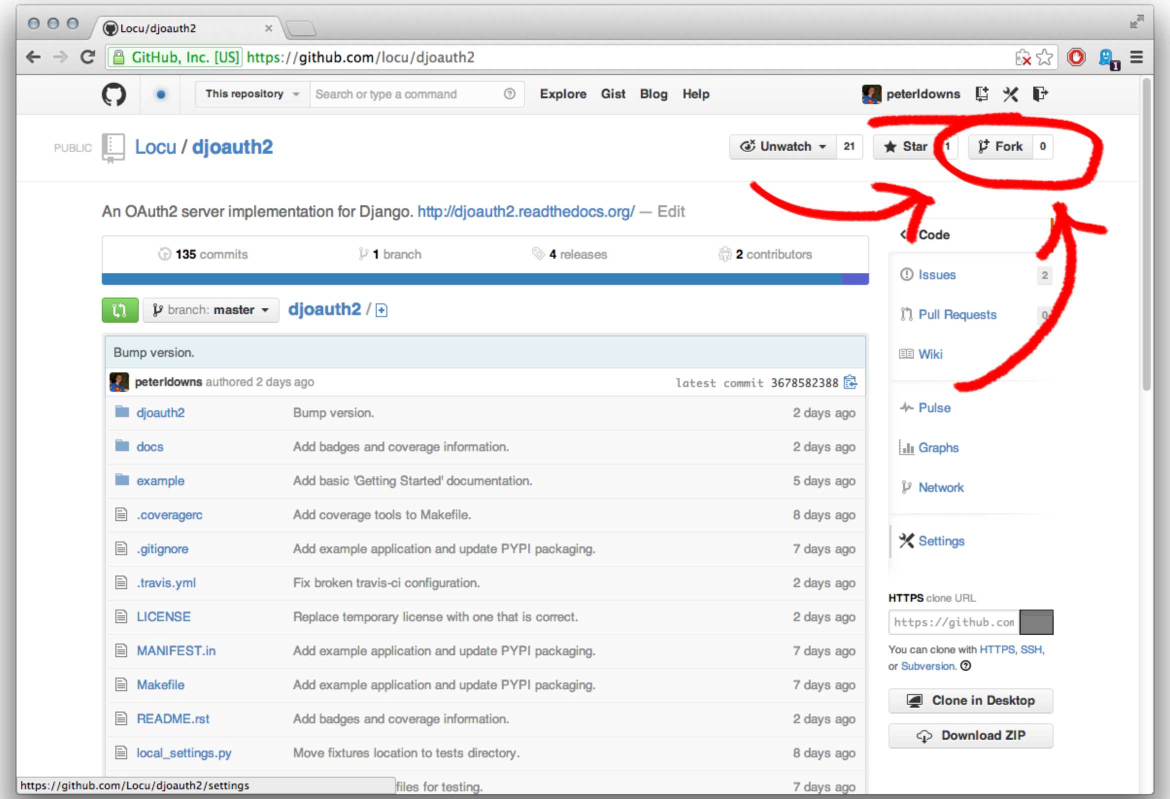
CHAPTER 7

Contributing

We <3 contributions; please feel free to check out the code! In general, this is a quick overview of how to contribute to DJOAuth2 using the standard Github pull-request flow. For more information, Github has a nice overview here.

Fork and clone the repository

The first step of contributing is creating your own copy ("fork") of the main DJOAuth2 repository. Do this through the Github web interface:



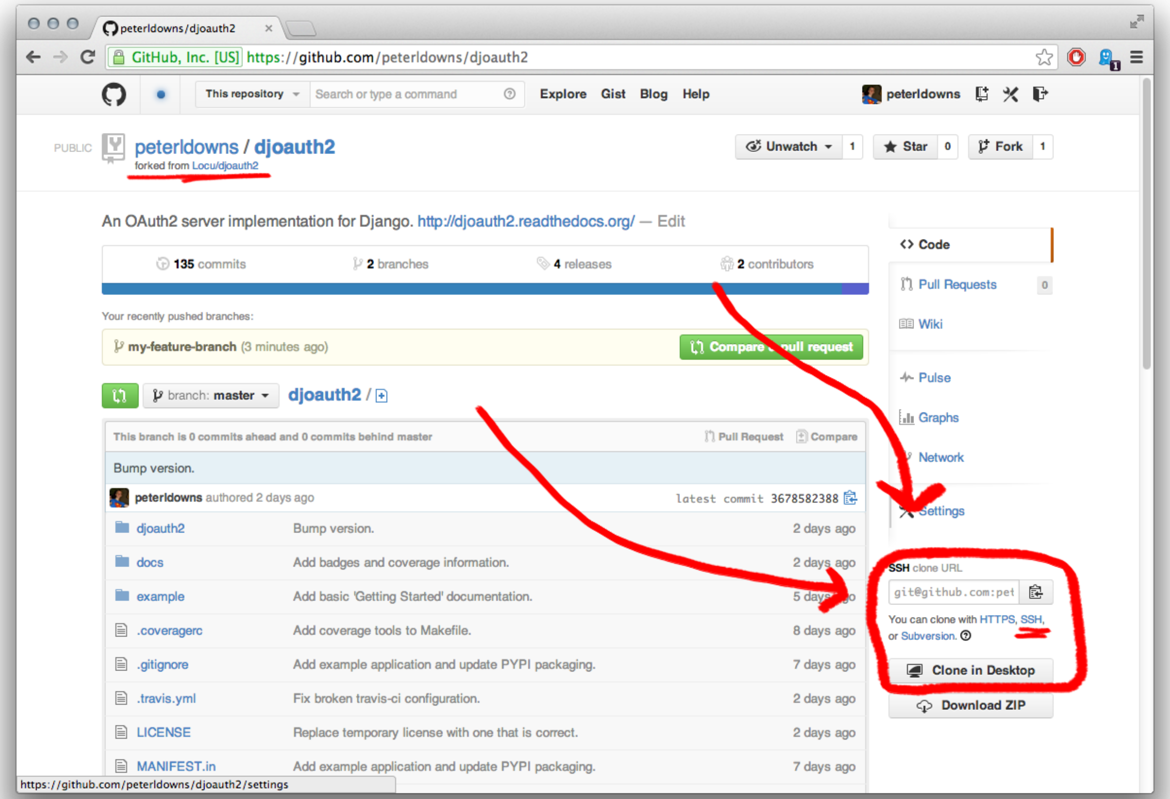
An OAuth2 server implementation for Django. http://djoauth2.readthedocs.org/ — Edit

Bump version.

peterldowns authored 2 days ago — latest commit 3678582388

djoauth2	Bump version.	2 days ago
docs	Add badges and coverage information.	2 days ago
example	Add basic 'Getting Started' documentation.	5 days ago
.coveragerc	Add coverage tools to Makefile.	8 days ago
.gitignore	Add example application and update PYPI packaging.	7 days ago
.travis.yml	Fix broken travis-ci configuration.	2 days ago
LICENSE	Replace temporary license with one that is correct.	2 days ago
MANIFEST.in	Add example application and update PYPI packaging.	7 days ago
Makefile	Add example application and update PYPI packaging.	7 days ago
README.rst	Add badges and coverage information.	2 days ago
local_settings.py	Move fixtures location to tests directory.	8 days ago
	files for testing.	7 days ago

Now that you have a copy, copy the “SSH clone URL” from the right-most column:



An OAuth2 server implementation for Django. http://djoauth2.readthedocs.org/ — Edit

Bump version.
peterldowns authored 2 days ago — latest commit 3678582388

djoauth2	Bump version.	2 days ago
docs	Add badges and coverage information.	2 days ago
example	Add basic 'Getting Started' documentation.	5 days ago
.coveragerc	Add coverage tools to Makefile.	8 days ago
.gitignore	Add example application and update PYPI packaging.	7 days ago
.travis.yml	Fix broken travis-ci configuration.	2 days ago
LICENSE	Replace temporary license with one that is correct.	2 days ago
MANIFEST.in	Add example application and update PYPI packaging.	7 days ago

and run the following commands from a local terminal:

```
cd ~

# The git@github.com URL is the "SSH clone URL" that you copied.
git clone git@github.com:<YOUR_USER_NAME>/djoauth2.git
cd djoauth2
```

Install dependencies

We rely on *virtualenv*_ for managing dependencies in order to make it as easy as possible to start contributing. Before contributing, run the following commands:

```
# Install development dependencies inside a new virtualenv
make dev-env

# Activate the virtualenv so that you have access to the dependencies that
# were installed.
. dev-env/bin/activate
```

Making changes

After setting up your virtualenv, check out a new branch locally:

```
git checkout -b 'my-feature-branch'
```

Now make your changes. Don't forget to update the tests! Please follow our style guide:

- 2-space indents
- All indents are spaces, not tabs.
- Wrap lines at 80 characters.

```
vim djoauth2/...
vim djoauth2/tests/...
```

Schema Migrations

If your changes touched the `models.py` file, you must attempt to generate a South migration in case the schema has changed.

It's important that for backwards-compatibility reasons you use South version 0.7.6 and Django 1.4.3 to generate migration files. After entering the `dev-env` virtualenv, run the following commands:

```
pip install Django==1.4.3
pip install South==0.7.6
```

Then, generate the migrations with the included script:

```
./generate_migrations.py

# Now, test to see that they apply without an error.
./generate_migrations.py --test-migrations
```

Testing

DJOAuth2 is a standalone Django application, which can be hard to test. To obviate a need for installing and re-installing inside of a test project, we provide a script (`runtests.py`) that sets up a minimal Django environment To use it, enter your shell and run:

```
# Run all of the tests
./runtests.py
# or
make tests

# Run a group of tests
./runtests.py djoauth2.tests.TestAuthorizationCodeEndpoint

# Run an individual test
./runtests.py djoauth2.tests.TestAuthorizationCodeEndpoint.test_get_requests_succeed
```

Coverage

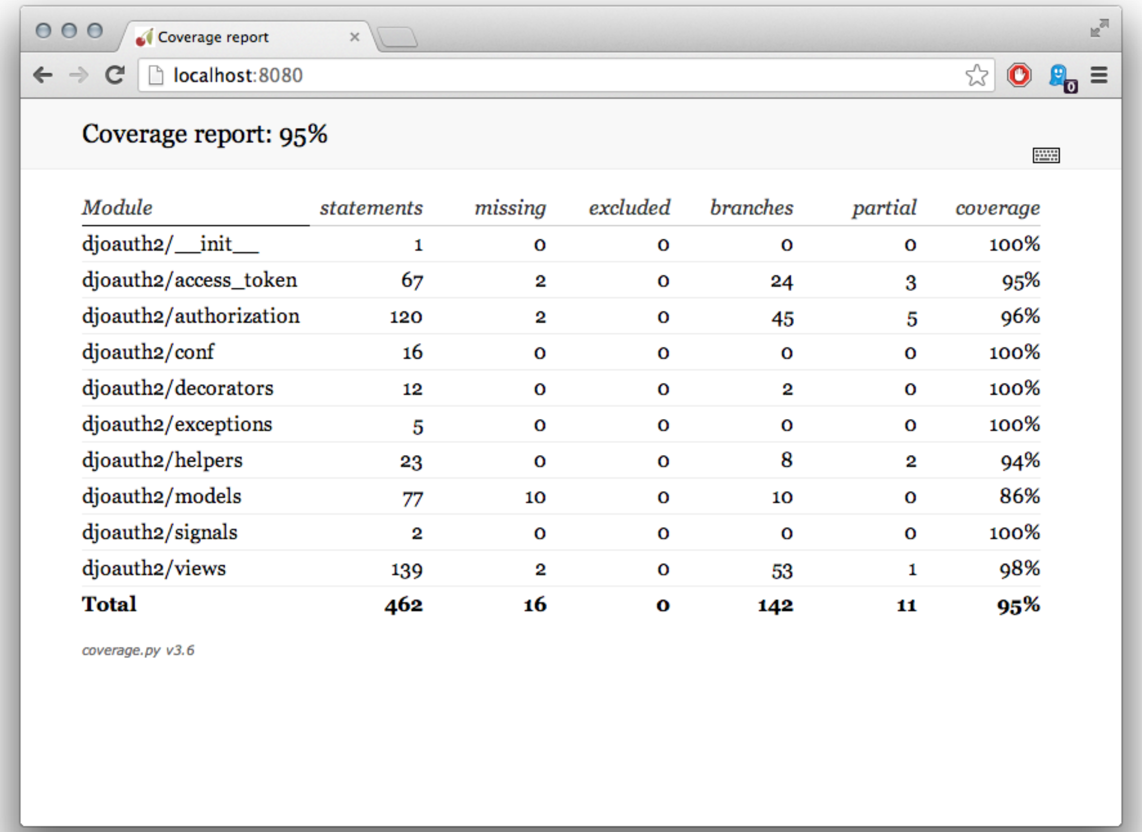
While we don't fetishize 100% coverage, it can be useful to double check that testing actually exercised the code that you added.

To get a coverage report, run `make coverage`. This will output a brief summary report in the terminal and also generate an interactive HTML version of the report. The interactive version will display the code line-by-line and highlight any code that was not covered by the tests.

```
# Generate the coverage report
make coverage

# Fire up a webserver to view the interactive HTML version
cd docs/coverage/
python -m SimpleHTTPServer 8080

# Now navigate to localhost:8080 in a browser
```



Coverage report: 95%

Module	statements	missing	excluded	branches	partial	coverage
djoauth2/__init__	1	0	0	0	0	100%
djoauth2/access_token	67	2	0	24	3	95%
djoauth2/authorization	120	2	0	45	5	96%
djoauth2/conf	16	0	0	0	0	100%
djoauth2/decorators	12	0	0	2	0	100%
djoauth2/exceptions	5	0	0	0	0	100%
djoauth2/helpers	23	0	0	8	2	94%
djoauth2/models	77	10	0	10	0	86%
djoauth2/signals	2	0	0	0	0	100%
djoauth2/views	139	2	0	53	1	98%
Total	**462**	**16**	**0**	**142**	**11**	**95%**

coverage.py v3.6

Updating Documentation

Made changes that require documentation (hint: probably)? Rebuild the docs:

```
make docs
```

And view them in your browser locally:

```
cd docs/_build/html
python -m SimpleHTTPServer 8080

Now navigate to localhost:8080 in a browser
```

By the way, if you have any questions, concerns, or complaints about the current documentation, **please** let us know and/or submit a pull request! We're committed to making the docs as easy to use as possible, so if something is not working we'd love to hear it.

Committing

Once your changes are finished (including tests and documentation) it's time to commit them:

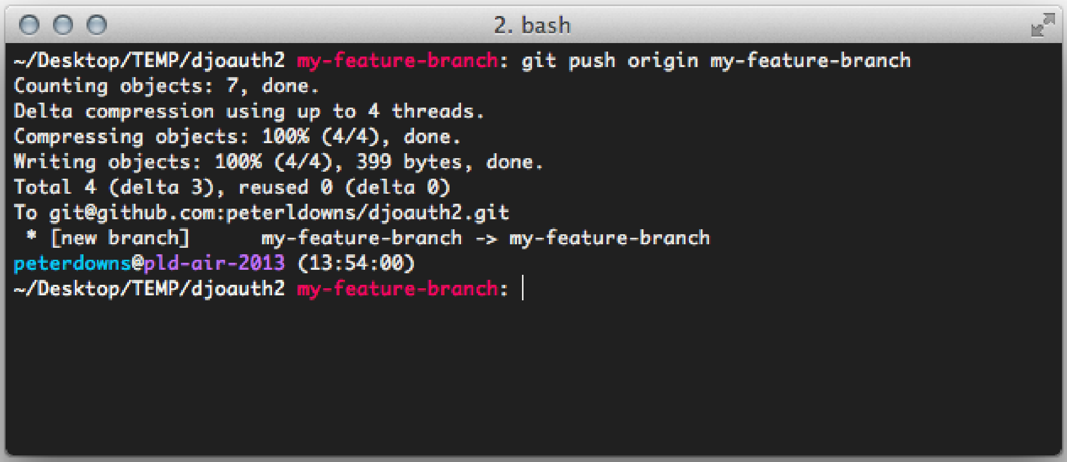
```
git commit -a -m "Add my new feature."
```

Submitting a pull request

Once your changes are locally committed and tested, it's time to submit a pull request to get your changes reviewed and merged upstream. Again, Github has a nice overview here.

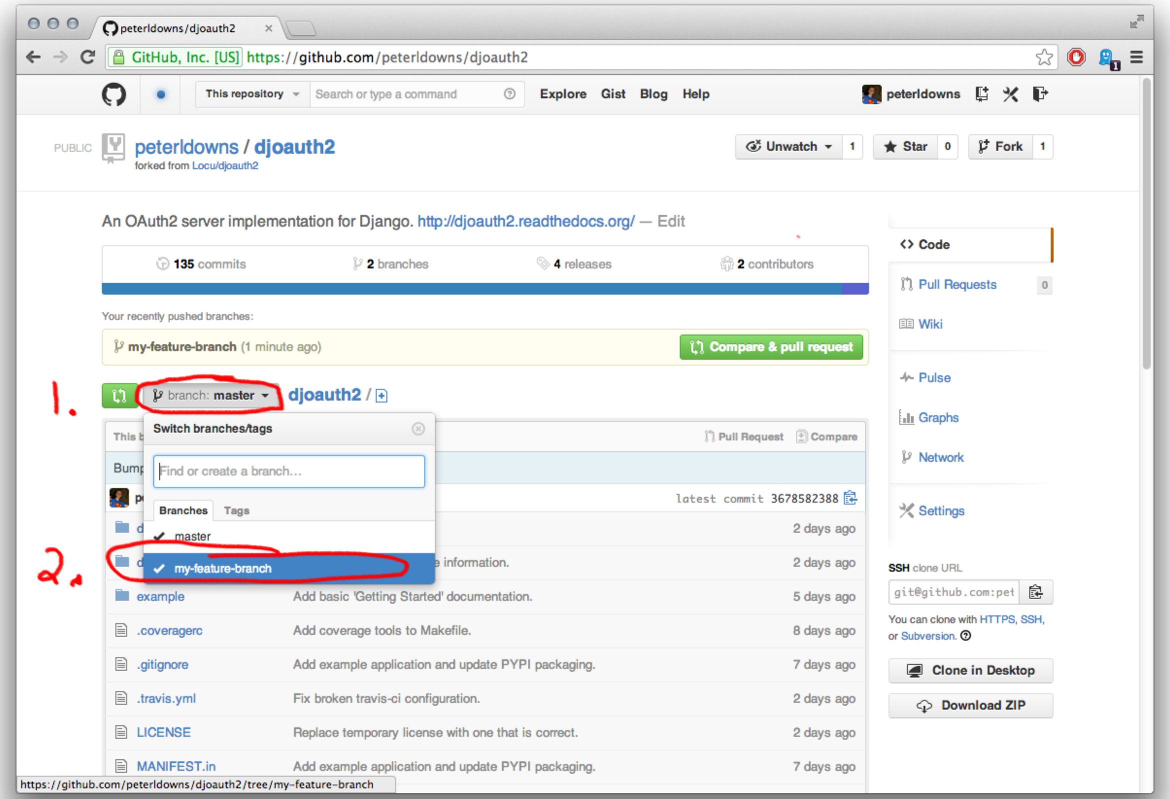
- Push your changes to your github repository:

```
git push origin my-feature-branch
```

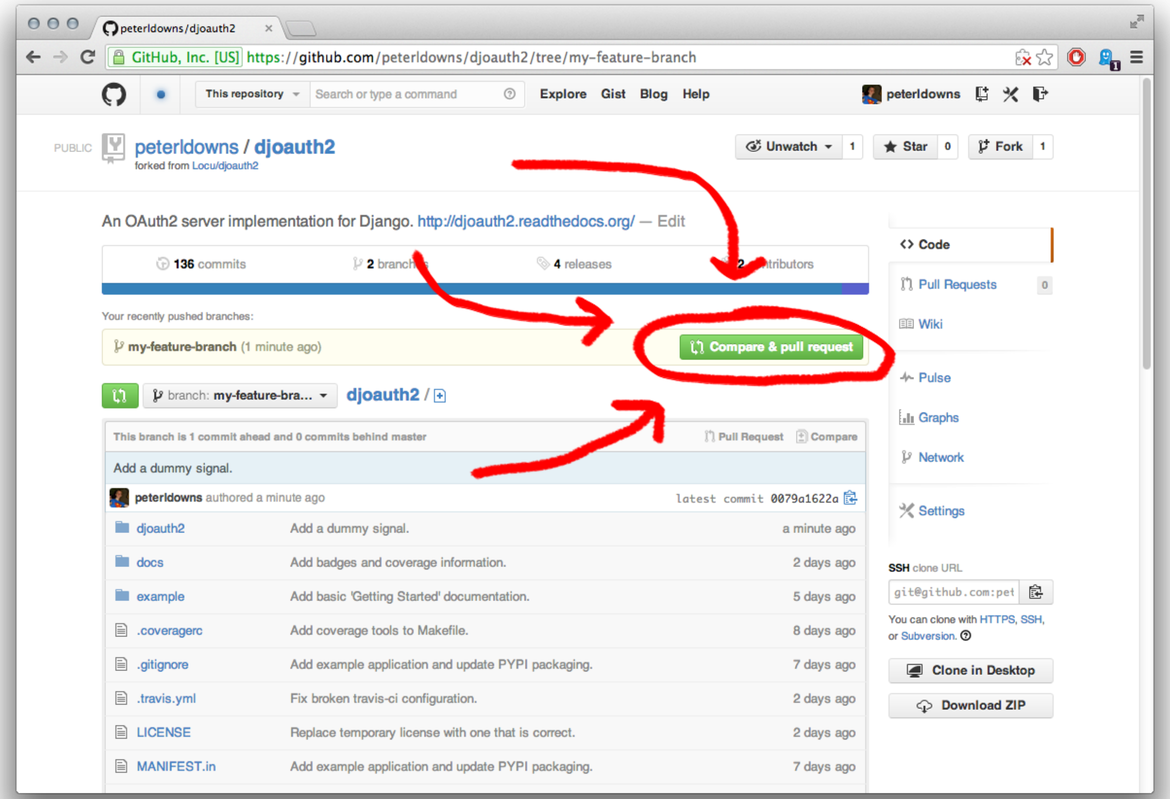


```
~/Desktop/TEMP/djoauth2 my-feature-branch: git push origin my-feature-branch
Counting objects: 7, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 399 bytes, done.
Total 4 (delta 3), reused 0 (delta 0)
To git@github.com:peterldowns/djoauth2.git
 * [new branch]      my-feature-branch -> my-feature-branch
peterdowns@pld-air-2013 (13:54:00)
~/Desktop/TEMP/djoauth2 my-feature-branch: |
```

- In Github, switch to `my-feature-branch`



- Click on the large green "compare & pull request" button:



An OAuth2 server implementation for Django. http://djoauth2.readthedocs.org/ — Edit

Your recently pushed branches:

my-feature-branch (1 minute ago)

This branch is 1 commit ahead and 0 commits behind master

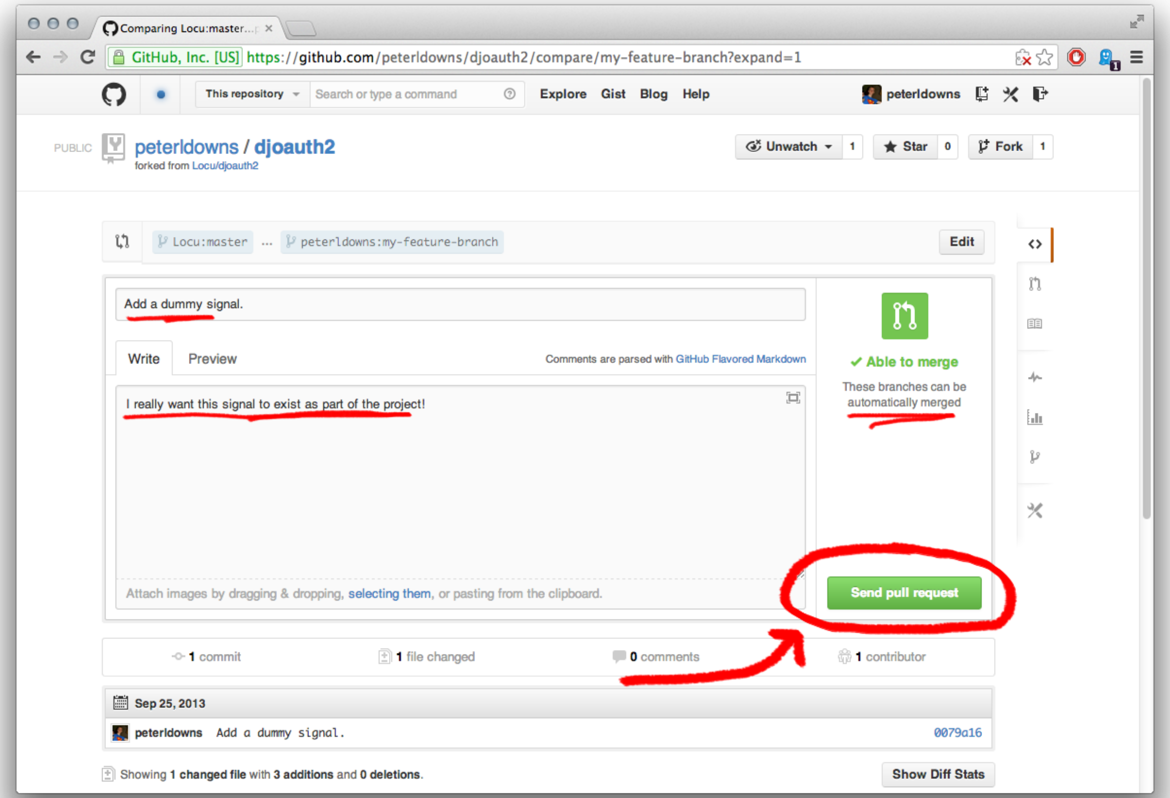
Add a dummy signal.

peterldowns authored a minute ago — latest commit 0079a1622a

djoauth2	Add a dummy signal.	a minute ago
docs	Add badges and coverage information.	2 days ago
example	Add basic 'Getting Started' documentation.	5 days ago
.coveragerc	Add coverage tools to Makefile.	8 days ago
.gitignore	Add example application and update PYPI packaging.	7 days ago
.travis.yml	Fix broken travis-ci configuration.	2 days ago
LICENSE	Replace temporary license with one that is correct.	2 days ago
MANIFEST.in	Add example application and update PYPI packaging.	7 days ago

You can clone with HTTPS, SSH, or Subversion.

- Write up a nice explanation of your changes and fire it off!



https://github.com/peterldowns/djoauth2/compare/my-feature-branch?expand=1

Add a dummy signal.

Comments are parsed with GitHub Flavored Markdown

I really want this signal to exist as part of the project!

Attach images by dragging & dropping, selecting them, or pasting from the clipboard.

✓ Able to merge

These branches can be automatically merged

peterldowns Add a dummy signal. 0079a16

Showing 1 changed file with 3 additions and 0 deletions.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index