django-all-access Documentation

Release 0.9.0

Mark Lavin

Mar 06, 2017

Contents

django-all-access is a reusable application for user registration and authentication from OAuth 1.0 and OAuth 2.0 providers such as Twitter and Facebook.

The goal of this project is to make it easy to create your own workflows for authenticating with these remote APIs. django-all-access will provide the simple views with sane defaults along with hooks to override the default behavior. You can find a basic demo application running at http://django-all-access.mlavin.org/

CHAPTER 1

Features

- Sane and secure defaults for OAuth authentication
- Easy customization through class-based views
- Built on the amazing requests library

CHAPTER 2

Installation

It is easiest to install django-all-access from PyPi using pip:

```
pip install django-all-access
```

django-all-access requires Python 2.7 or 3.3+ along with the following Python packages:

```
django>=1.8
pycrypto>=2.4
requests>=2.0
requests_oauthlib>=0.4.2
oauthlib>=0.6.2
```

CHAPTER 3

Documentation

Additional documentation on using django-all-access is available on Read The Docs.

CHAPTER 4

License

django-all-access is released under the BSD License. See the LICENSE file for more details.

CHAPTER 5

Contributing

If you have questions about using django-all-access or want to follow updates about the project you can join the mailing list through Google Groups.

If you think you've found a bug or are interested in contributing to this project check out django-all-access on Github.

CHAPTER 6

Contents

Getting Started

Below are the basic steps need to get django-all-access integrated into your Django project.

Configure Settings

You need to add `allaccess` to your installed apps as well as include an additional authentication backend in your project settings. django-all-access requires `django.contrib.auth`, `django.contrib.sessions` and `django.contrib.messages` which are enabled in Django by default. `django.contrib.admin` is recommended for managing the set of providers, but is not required.

```
INSTALLED_APPS = (
    # Required contrib apps
    'django.contrib.auth',
    'django.contrib.sessions',
    'django.contrib.messages',
    # Optional
    'django.contrib.admin',
    # Other installed apps would go here
    'allaccess',
)

AUTHENTICATION_BACKENDS = (
    # Default backend
    'django.contrib.auth.backends.ModelBackend',
    # Additional backend
    'allaccess.backends.AuthorizedServiceBackend',
)
```

Note that `AUTHENTICATION_BACKENDS` is not included in the default settings created by `startproject`. If you want to continue to use the default username/password based authentication, you should be sure to include `django.contrib.auth.backends.ModelBackend` in this setting.

By default, django-all-access uses the built-in Django settings `LOGIN_URL` and `LOGIN_REDIRECT_URL`. You should be sure that these are set to valid URLs for your site.

Configure Urls

To use the default redirect and callback views, you should include them in your root URL configuration.

```
from django.conf.urls import include


urlpatterns = [
    # Other URL patterns would go here
    url(r'^accounts/', include('allaccess.urls')),
]
```

This makes the login URL for a particular provider `/accounts/login/<provider>/`, such as `/accounts/login/twitter/` or `/accounts/login/facebook/`. Once the user has authenticated with the remote provider, they will be sent back to `/accounts/callback/<provider>/`, such as `/accounts/callback/twitter/` or `/accounts/callback/facebook/`.

Create Database Tables

You'll need to create the necessary database tables for storing OAuth providers and user associations with those providers. This is done with the `migrate` management command built into Django:

```
python manage.py migrate allaccess
```

Next Steps

At this point your project is configured to use the default django-all-access authentication, but no providers have been added. Continue reading to learn how to add providers for your project.

Configuring Providers

django-all-access configures and stores the set of OAuth providers in the database. To enable your users to authenticate with a particular provider, you will need to add the OAuth API URLs as well as your application's consumer key and consumer secret. The process of registering your application with each provider will vary and you should refer to the provider's API documentation for more information.

Note: While the consumer key/secret pairs are stored in the database as opposed to putting them in the settings file, they are encrypted using the AES specification. Since this is a symmetric-key encryption the key/secret pairs can still be read if the encryption key is compromised. In this case django-all-access uses a key based on the standard `SECRET_KEY` setting. You should take care to keep this setting secret as its name would imply.

Common Providers

To get you started, there is an initial fixture of commonly used providers. This includes the URLs needed for Facebook, Twitter, Google, Microsoft Live, Github and Bitbucket. Once you've added `allaccess` to your `INSTALLED_APP`

and created the tables with `migrate`, you can load this fixture via:

```
python manage.py loaddata common_providers.json
```

This does not include the consumer id/key or secret which will still need to be added to the records. The below examples will help you understand what these values mean and how they would be populated for additional providers you might want to use.

OAuth 1.0 Providers

OAuth 1.0 Protocol is defined by RFC 5849. It is sometimes referred to as 3-Legged OAuth due to the number of requests between the provider and consumer.

To enable an OAuth provider, you should add a `Provider` record with the necessary `request_token_url`, `authorization_url` and `access_token_url` as defined by the protocol. The provider's API documentation should detail these for you. You will also need to define a `profile_url` which is the API endpoint for requesting the currently authenticated user's profile information. You will also need to register for a key/secret pair from the provider.

This protocol is implemented by a number of providers. These providers include Twitter, Netflix, Yahoo, Linkedin, Flickr, Bitbucket, and Dropbox. Additional providers can be found on the OAuth.net Wiki.

Twitter Example

Twitter is a popular social website which provides a REST API with OAuth 1.0 authentication. If you wanted to enable Twitter authentication on your website using django-all-access, you would create the following `Provider` record:

```
name: twitter
request_token_url: https://api.twitter.com/oauth/request_token
authorization_url: https://api.twitter.com/oauth/authenticate
access_token_url: https://api.twitter.com/oauth/access_token
profile_url: https://api.twitter.com/1.1/account/verify_credentials.json
```

After adding your consumer key and secret to this record you should now be able to authenticate with Twitter by visiting `/accounts/login/twitter/`. You can find more information on the Twitter API on their developer site.

OAuth 2.0 Providers

Unlike OAuth 1.0, OAuth 2.0 is only a working draft and not an official standard. In many ways it is much simpler than its predecessor. It is often referred to as 2-Legged OAuth because it removes the need for the request token step.

To enable an OAuth provider, you should add a `Provider` record with the necessary `authorization_url` and `access_token_url` as defined by the protocol. The provider's API documentation should detail these for you. You will also need to define a `profile_url` which is the API endpoint for requesting the currently authenticated user's profile information. You will also need to register for a key/secret pair from the provider.

Providers which implement the OAuth 2.0 protocol include Facebook, Google, FourSquare, Meetup, Github, and Yammer.

Facebook Example

Facebook is a large social network which provides a REST API with OAuth 2.0 authentication. The below `Provider` record will enable Facebook authentication:

```
name: facebook
authorization_url: https://www.facebook.com/v2.8/dialog/oauth
access_token_url: https://graph.facebook.com/v2.8/oauth/access_token
profile_url: https://graph.facebook.com/v2.8/me
```

As you can see, the `request_token_url` is not included because it is not needed. After adding your consumer key and secret to this record you should now be able to authenticate with Facebook by visiting `/accounts/login/facebook/`. Facebook also has developer docs for additional information on using their API.

Note: Facebook began using the version number in the URL as part of their 2.0 API. Since then very little has changed with regard to the OAuth flow but the version number is now required. The latest version of the API might not match the documentation here. For the most up to date info on the Facebook API you should consult their API docs.

Customizing Redirects and Callbacks

django-all-access provides default views/urls for authentication. These are built from Django's class based views making them easy to extend or override the default behavior in your project.

OAuthRedirect View

The initial step for authenticating with any OAuth provider is redirecting the user to the provider's website. The *OAuthRedirect* view extends from the RedirectView By default it is mapped to the `allaccess-login` URL name. This view takes one keyword argument from the URL pattern `provider` which corresponds to the `Provider.name` for an enabled provider. If no enabled provider is found for the name, this view will return a 404.

class `OAuthRedirect`

`client_class`
Used to change the *BaseOAuthClient* used by the view. See *OAuthRedirect.get_client()* for more details.

New in version 0.8.

`params`
Used to pass additional parameters to the authorization redirect (i.e. `scope` requests). See *OAuthRedirect.get_additional_parameters()* for more details.

`get_client`(*provider*)
Here you can override the OAuth client class which is used to generate the redirect URL. Another use case is to disable the enforcement of the OAuth 2.0 `state` parameter for providers which don't support it. If you are using the view for a single provider, it would be easiest to set the *OAuthRedirect.client_class* attribute on the class instead.

You should be sure to use the same client class for the callback view as well.

`get_redirect_url`(***kwargs*)
This method is originally defined by the RedirectView. The redirect URL is constructed from the `Provider.authorization_url` along with the necessary parameters to match the OAuth specifications. You should not need to override this method in your application.

`get_additional_parameters`(*provider*)
Here you can return additional parameters for the authorization request. By default this returns `{}`. A common usage for overriding this method is to request additional permissions for the authorization. There

is no standard for additional permissions in the OAuth 1.0 specification. For an OAuth 2.0 provider this is done with the `scope` parameter.

get_callback_url (*provider*)
This returns the URL which the remote provider should return the user after authentication. It is called by *OAuthRedirect.get_redirect_url()* to construct the appropriate redirect URL. By default the reverses the `allaccess-callback` URL name with the passed provider name.

You may want to override this method in your application if you wish to have a custom callback for a given provider, a different callback for login vs registration, or a different callback for an authenticated user associating a new provider with their account.

OAuthCallback View

After the user has authenticated with the remote provider or denied access to your application request, they are returned to the callback specifed in the initial redirect. *OAuthCallback* defines the default behaviour on this callback. This view extends from the base View class. By default it is mapped to the `allaccess-callback` URL name. Similar to the *OAuthRedirect* view, this view takes one keyword argument `provider` which corresponds to the `Provider.name` for an enabled provider. If no enabled provider is found for the name, this view will return a 404.

class OAuthCallback

client_class
Used to change the *BaseOAuthClient* used by the view. See *OAuthCallback.get_client()* for more details.

New in version 0.8.

provider_id
Used to customize how the user identifier is found from the user profile response from the provider. If the provider response includes a nested response then this value can include a dotted path to the id value.

For example if the response is *{'result': {'user': {'id': 'XXX'}}}* then you can set this attribute to *result.user.id* to access the value. See *OAuthCallback.get_user_id()* for more details.

get_callback_url (*provider*)
This returns the callback URL specified in the initial redirect if it is different than the current `request.path`. By default the callback URL will be the same and this view will return `None`. You will most likely not need to change this in your project.

get_client (*provider*)
Here you can override the OAuth client class which is used to fetch the access token and user information. Another use case is to disable the enforcement of the OAuth 2.0 `state` parameter for providers which don't support it. If you are using the view for a single provider, it would be easiest to set the *OAuthCallback.client_class* attribute on the class instead.

You should be sure to use the same client class for the redirect view as well.

get_error_redirect (*provider*, *reason*)
Returns the URL to send the user in the case of an authentication failure. The `reason` is a brief text description of the problem. By default this will return the user to the original login URL as defined by the `LOGIN_URL` setting.

get_login_redirect (*provider*, *user*, *access*, *new=False*)
You can use this to customize the URL to send the user on a successful authentication. By default this will be the `LOGIN_REDIRECT_URL` setting. The `new` parameter is there to indicate if this was a newly created or a previously existing user.

get_or_create_user(*provider*, *access*, *info*)
This method is used by *OAuthCallback.handle_new_user()* to construct a new user with a random username, no email and an unusable password. You may want to override this user to complete more of their infomation or attempt to match them to an existing user by either their username or email.

OAuthCallback.handle_new_user() will connect the user to the `access` record and does not need to be handled here.

Note If you are using Django 1.5 support for a custom User model, you should override this method to ensure the user is created correctly.

get_user_id(*provider*, *info*)
This method should return the unique identifier from the profile information. If the id cannot be determined, this should return `None`. The `info` parameter will be the parsed JSON response from the user's profile. If the response wasn't JSON, it will be the plain text response. By default this looks for a key `id` in the JSON dictionary. This will work for a number of providers, but will need to be changed to fit more complex response structures.

You can customize how this lookup is done by setting the *OAuthCallback.provider_id*. This can be done either in the class definition or when calling *.as_view*.

handle_existing_user(*provider*, *user*, *access*, *info*)
At this point the `user` has been authenticated via their `access` model with this provider, but they have not been logged in. This method will login the user and redirect them to the URL returned by *OAuthCallback.get_login_redirect()* with `new=False`.

The user's profile info is passed to this method to allow for updating their data from their provider profile, but this is not done by default.

handle_login_failure(*provider*, *reason*)
In the case of a failure to fetch the user's access token or remote profile information or determine their id from that info, this method will be called. It attachs a brief error message to the request via `contrib.messages` and redirects the user to the result of the *OAuthCallback.get_error_redirect()* method. You should override this function to add any additional logging or handling.

handle_new_user(*provider*, *access*, *info*)
If the user could not be matched to an existing `AccountAccess` record for this provider or that record did not contain a user, this method will be called. At this point the `access` record has already been saved but is not tied to a user. This will call *OAuthCallback.get_or_create_user()* to construct a new user record. The user is then logged in and redirected to the result of the *OAuthCallback.get_login_redirect()* call with `new=True`

You may want to override this user to complete more of their infomation or attempt to match them to an existing user by either their username or email. You may want to override this to redirect them without creating a new user in order to have them complete another registration form (i.e. pick a username or provide an email if not returned by the provider).

Customization in URLs

For some minor customizations to the redirects and callbacks, it's possible to handle that in the URL inclusion rather than by creating a subclass of the view. The most common customizations are adding additional scope on the redirect and changing how the provider identifier is found on the callback. Below is an example `urls.py` which handles both of these cases.

```
from django.conf.urls import include, url

from allaccess.views import OAuthRedirect, OAuthCallback
```

```
urlpatterns = [
    # Customize Facebook redirect to request additional scope
    url(r'^accounts/login/(?P<provider>facebook)/$',
        OAuthRedirect.as_view(params={'scope': 'email'})),
    # Customize Foursqaure callback to handle nested response
    url(r'^accounts/callback/(?P<provider>foursquare)/$',
        OAuthCallback.as_view(provider_id='response.user.id')),
    # All other provider cases are handled by the defaults
    url(r'^accounts/', include('allaccess.urls')),
]
```

Additional Scope Example

As noted above, the default *OAuthRedirect* redirect does not request any additional permissions from the provider. It is recommended by most providers that you limit the number of additional permissions that you request. The user will see the list of permissions you are requesting and if they see a long list of permissions they may decline the authorization. The below example shows how you can request additional parameters for various providers.

```
from allaccess.views import OAuthRedirect

class AdditionalPermissionsRedirect(OAuthRedirect):

    def get_additional_parameters(self, provider):
        if provider.name == 'facebook':
            # Request permission to see user's email
            return {'scope': 'email'}
        if provider.name == 'google':
            # Request permission to see user's profile and email
            perms = ['userinfo.email', 'userinfo.profile']
            scope = ' '.join(['https://www.googleapis.com/auth/' + p for p in perms])
            return {'scope': scope}
        return super(AdditionalPermissionsRedirect, self).get_additional_
↪parameters(provider)
```

This would be used instead of the default *OAuthRedirect* for the `allaccess-login` URL. Remember that this logic can be based on the provider or even the current request. That would allow your project to A/B test requesting more or less permissions to see its impact on user registrations.

Additional Accounts Example

You may want to allow a user to associate their account on your website with multiple providers. This example will show a basic outline of how you can customize these views for that purpose.

First we will define a new callback which will associate the provider with the current user rather than creating a new user. This view will also have to handle the case that another user is associated with the new provider. For this the view will just return an error.

```
from allaccess.views import OAuthCallback

class AssociateCallback(OAuthCallback):

    def get_or_create_user(self, provider, access, info):
        return self.request.user

    def handle_existing_user(self, provider, user, access, info):
```

```
        if user != self.request.user:
            return self.handle_login_failure(provider, "Another user is associated␣
↪with this account")
        # User was already associated with this account
        return super(AssociateCallback, self).handle_existing_user(provider, user,␣
↪access, info)
```

This view will require authentication which is handled in the URL pattern. There are multiple methods for decorating class based views which are detailed in the Django docs.

Next we will need a redirect view to send the user to this callback. This view will also require that the user already be authenticated which can be handled in the URL pattern.

```
from django.core.urlresolvers import reverse
from allaccess.views import OAuthRedirect

class AssociateRedirect(OAuthRedirect):

    def get_callback_url(self, provider):
        return reverse('associate-callback', kwargs={'provider': provider.name})
```

This assumes that we named the pattern for the above callback `associate-callback`. An example set of URL patterns is given below.

```
from django.contrib.auth.decorators import login_required

from .views import AssociateRedirect, AssociateCallback

urlpatterns = [
    url(r'^associate/(?P<provider>(\w|-)+)/$', login_required(AssociateRedirect.as_
↪view()), name='associate'),
    url(r'^associate-callback/(?P<provider>(\w|-)+)/$', login_
↪required(AssociateCallback.as_view()), name='associate-callback'),
]
```

That is the basic outline of how you would allow multiple account associations. This could be further customized using the hooks described earlier.

Additional API Calls

django-all-access requests the user's access token and fetches their profile information during the authentication process. If you want to make additional API calls on behalf of the user, it is easy to do and you have the full power of the python-requests library.

Getting the API

You can access the API client through the `AccountAccess.api_client` property. This will return either a *`OAuthClient`* or *`OAuth2Client`* based on the provider. API requests can be made using either the *`BaseOAuthClient.request()`* method. This takes the HTTP method as the first parameter and the URL as the second. An example for the Twitter API is given below:

```
from allaccess.views import OAuthCallback

class NewTweetCallback(OAuthCallback):
```

```
    def get_login_redirect(self, provider, user, access, new=False):
        "Send a tweet for new Twitter users."
        if new and provider.name == 'twitter':
            api = access.api_client
            url = 'https://api.twitter.com/1/statuses/update.json'
            data = {'status': 'I just joined an awesome new site!'}
            response = api.request('post', url, data=data)
            # Check for errors in the response?
        return super(NewTweetCallback, self).get_login_redirect(provider, user,␣
↪access, new)
```

This assumes that you have requested sufficient permissions to tweet on behalf of the user. While this example is done in the callback, you can access the API client at any time by querying the `AccountAccess` table. There is a catch in that the access token from the provider might have been revoked by the user or expired. You should refer to the provider's API documentation for information regarding available endpoints and the access token expiration.

The *`BaseOAuthClient.request()`* method is a thin wrapper around the underlying `python-requests` library which sets up the appropriate authenication for OAuth 1.0 or OAuth 2.0. For more information on additional hooks available, you should refer to the python-requests documentation.

API Client

The *`OAuthClient`* or *`OAuth2Client`* classes define methods centered around OAuth specifications and the authentication and registration workflow. The common methods are defined in a *`BaseOAuthClient`*. If you are going to extend the client for a particular provider, it is recommended that you extend the appropriate OAuth 1.0 or 2.0 client rather than the *`BaseOAuthClient`*.

class BaseOAuthClient

__init__(*provider*, *token=''*)
The client classes are created with an associated provider model record. The provider is used to provide the necessary URL (request token, access token, profile URL) information to the client.

get_access_token(*request*, *callback=None*)
Used to fetch the access token from the callback URL. Unless you are familiar with the OAuth specifications, it is not recommended that you override this method.

get_profile_info(*raw_token*)
Fetches and parses the profile information from the provider's profile URL. This assumes that the response is JSON. If not, you may need to override this method.

get_redirect_args(*request*, *callback*)
Builds the necessary query string parameters for the initial redirect based on the OAuth specification. Additional parameters are better added using *`OAuthRedirect.get_additional_parameters()`*. Unless you are familiar with the OAuth specifications, it is not recommended that you override this method.

get_redirect_url(*request*, *callback*)
Builds the appropriate OAuth callback URL based on the provider information and the result of *`BaseOAuthClient.get_redirect_args()`*. Unless you are familiar with the OAuth specifications, it is not recommended that you override this method.

parse_raw_token(*raw_token*)
Parses the token (key, secret) information from the raw token response.

request (*method*, *url*, ***kwargs*)
A thin wrapper around `python-requests`, this also sets up the appropriate authentication headers/parameters.

session_key
Returns a key for storing information in the user's session. For OAuth 1.0 this would be used to store the request token information. For OAuth 2.0 this is used for enforcing the `state` parameter.

Beyond the methods above, the *OAuthClient* also defines the below methods.

class OAuthClient

get_request_token (*request*, *callback*)
Retrieves the request token prior to the initial redirect to the provider. This is stored in the session using the *BaseOAuthClient.session_key* which is unique per provider. Unless you are familiar with the OAuth 1.0 specification, it is not recommended that you override this method.

OAuth2Client extends *BaseOAuthClient* to include these additional methods.

class OAuth2Client

check_application_state (*request*, *callback*)
On the callback this method is called to enforce the use of the `state` parameter. The use of `state` is optional in the OAuth 2.0 spec but it is recommended and enforced by default by django-all-access. If you do not want to enforce the use of `state`, you should override *OAuth2Client.get_application_state()* and leave this method alone.

get_application_state (*request*, *callback*)
Prior to the redirect, this method is used to generate a random `state` parameter which is stored in the session based on the *BaseOAuthClient.session_key*. By default it generates a secure random 32 character string. If you wish to make it longer you can override this method. If you do not want to enforce the `state` parameter or the provider you are using does not allow it, you can override this to return `None`.

Contributing Guide

There are a number of ways to contribute to django-all-access. If you are interested in making django-all-access better then this guide will help you find a way to contribute.

Ways to Contribute

Not all contributions are source code related. You can contribute to the project by writing a blog post on using django-all-access and sharing with the mailing list. You can also submit bug reports, feature requests or documentation updates through the Github issues.

Getting the Source

You can clone the repository from Github:

```
git clone git://github.com/mlavin/django-all-access.git
```

However this checkout will be read only. If you want to contribute code you should create a fork and clone your fork. You can then add the main repository as a remote:

```
git clone git@github.com:<your-username>/django-all-access.git
git remote add upstream git://github.com/mlavin/django-all-access.git
git fetch upstream
```

Running the Tests

When making changes to the code, either fixing bugs or adding features, you'll want to run the tests to ensure that you have not broken any of the existing functionality. With the code checked out and Django installed you can run the tests via:

```
python setup.py test
```

or:

```
python runtests.py
```

Note that the tests require the mock library. To test against multiple versions of Django you can use install and use `tox>=1.4`. The `tox` command will run the tests against the currently supported Python and Django versions.

    # Build all environments tox # Build a single environment tox -e py27-django18-normal

Building all environments will also build the documentation. More on that in the next section.

Building the Documentation

This project aims to have a minimal core with hooks for customization. That makes documentation an important part of the project. Useful examples and notes on common use cases are a great way to contribute and improve the documentation.

The docs are written in ReST and built using Sphinx. As noted above, you can use tox to build the documentation or you can build them on their own via:

```
tox -e docs
```

or:

```
make html
```

from inside the `docs/` directory.

Coding Standards

Code contributions should follow the PEP8 and Django contributing style standards. Please note that these are only guidelines. Overall code consistency and readability are more important than strict adherence to these guides.

Submitting a Pull Request

The easiest way to contribute code or documentation changes is through a pull request. For information on submitting a pull request you can read the Github help page https://help.github.com/articles/using-pull-requests.

Pull requests are a place for the code to be reviewed before it is merged. This review will go over the coding style as well as if it solves the problem intended and fits in the scope of the project. It may be a long discussion or it might just be a simple thank you.

Not necessarily every request will be merged but you should not take it personally if your change is not accepted. If you want to increase the chances of your change being incorporated, here are some tips.

- Address a known issue. Preference is given to a request that fixes a currently open issue.
- Include documentation and tests when appropriate. New features should be tested and documented. Bugfixes should include tests which demonstrate the problem.
- Keep it simple. It's difficult to review a large block of code, so try to keep the scope of the change small.

If you aren't sure if a particular change is a good idea, or if it would be helpful to other users, just ask. You should also feel free to ask for help writing tests or writing documentation if you aren't sure how to go about it.

Release History

Release and change history for django-all-access

v0.9.0 (2016-11-12)

Encrypted fields for storing the provider configurations and access tokens now sign the values after encryption to dectect if the key is valid before attempting to decrypt. This was added thanks to Florian Demmer (@fdemmer).

Other small changes include:

- Added Django 1.10 and Python 3.5 to the test suite coverage.
- Updated documentation on Facebook version numbers.
- Update provider fixtures to include the latest version number for Facebook.

v0.8.0 (2016-01-23)

Minor clean up release which drops support for outdated versions of Django. As such it also removes the old South migrations and the commands related to django-social-auth.

- Added support for additional parameters in the redirect view.
- Added support for more complex id lookups in the callback view.
- Additional documentation examples for customizing the views.
- Added support for Django 1.9.
- Tracking code coverage reports with Codecov.io.

Backwards Incompatible Changes

- Python 3.2 is no longer officially supported or tested.
- Django < 1.8 is no longer officially supported or tested.
- requests_oauthlib < 0.4.2 is no longer officially supported.
- `migrate_social_accounts` and `migrate_social_accounts` commands have been removed.

v0.7.2 (2015-05-13)

- Model updates for Django 1.8 compatibility. Requires a non-DB altering migration.

v0.7.1 (2015-04-19)

- Fixed issue in `migrate_social_accounts` where output was overly verbose.
- Fixed issue in `migrate_social_accounts` with handling skipped providers.

v0.7.0 (2014-09-07)

This release adds support for 1.7 and the new style migrations. If you are using Django < 1.7 and South >= 1.0 this should continue to work without issue.

For those using Django < 1.7 and South < 1.0 you'll need to add the `SOUTH_MIGRATION_MODULES` setting to point to the old South migrations.

```
SOUTH_MIGRATION_MODULES = {
    'allaccess': 'allaccess.south_migrations',
}
```

No new migrations were added for this release, but this will be the new location for future migrations. If your DB tables are up to date from v0.6, upgrading to 1.7 and running:

```
python manage.py migrate allaccess
```

should automatically fake the initial migration using the new-style migrations.

Backwards Incompatible Changes

- Python 2.6 is no longer officially supported or tested.

v0.6.0 (2014-02-01)

This release adds a better migration path for moving from django-social-auth and includes changes to support running on the Google App Engine. There are two South migrations included with this release. To upgrade, you should run:

```
python manage.py migrate allaccess
```

More details for this change are noted under the "Backwards Incompatible Changes".

- Added `migrate_social_accounts` and `migrate_social_providers` management commands to help migrate data from django-social-auth.
- Updated `Provider` model for compatibility with running on the Google App Engine. Thanks to Marco Seguri for the report and fix.
- Increased the URL lengths for the fields on the `Provider` model. Thanks to Marco Seguri for the fix.
- Added support for serialization of `Provider` and `AccountAccess` records by natural keys.
- Included a fixture of common providers (Facebook, Twitter, Google, Microsoft Live, Github and Bitbucket). Thanks to Marco Seguri for the initial patch.

Backwards Incompatible Changes

- The `key` and `secret` columns on `Provider` were renamed to `consumer_key` and `consumer_secret`. `key` is a reserved property

name when using Google App Engine and `secret` was changed as well for consistency. A migration has been added for the change but if you were referencing the `key`/`secret` explicitly in your code those references need to be updated as well. - `ProviderManager.enabled` has been removed. This was a short-cut method for filtering out providers with key or secret values. However, it doesn't work on Google App Engine. It was only used in a few places internally so it was removed. The equivalent query is `Provider.objects.filter(consumer_secret__isnull=False, consumer_key__isnull=False)`

v0.5.1 (2013-08-16)

- Fix incompatibility with the existing South migrations and a customized User model. Thanks to Jharrod LaFon for the report and fix.

v0.5.0 (2013-03-18)

This release adds additional hooks for changing the OAuth client behaviors. It also adds support for Python 3.2+.

- New view hooks for customizing the OAuth client
- Fixed issue with including oauth_verifier in POST when fetching the access token
- Documented the API for *`OAuthClient`* and *`OAuth2Client`*
- Updated requirements to requests >= 1.0 and requests_oauthlib >= 0.3.0
- Updated requirement for PyCrypto >= 2.4

Backwards Incompatible Changes

- Dropped support for requests < 1.0
- Dropped support for Django < 1.4.2

v0.4.1 (2013-01-02)

There were incompatibilty issues with requests-oauthlib (0.2) and requests which required dropping requests 1.0 support. The requirement of oauthlib was also raised to 0.3.4 due to similar issues. For more detail see the below issues.

- https://github.com/requests/requests-oauthlib/issues/1
- https://github.com/requests/requests-oauthlib/pull/10

v0.4.0 (2012-12-19)

This release is largely to keep pace with features/changes to some of the dependencies. This also helps work toward Python 3.0 support.

- Updated for compatibility with Django 1.4 timezone support
- Updated for compatibility with Django 1.5 swappable `auth.User`
- **Updated for compatibility with Requests 1.0**

- Added requests_oauthlib requirement
- Updated requirement of oauthlib to 0.3 or higher

v0.3.0 (2012-07-13)

This release added some basic logging to django-all-access. To enable this logging in your project, you should update your `LOGGING` configuration to include `allaccess` in the `loggers` section. Below is an example:

```
LOGGING = {
    'handlers': {
        'console':{
            'level':'DEBUG',
            'class':'logging.StreamHandler',
        },
        'mail_admins': {
            'level': 'ERROR',
            'class': 'django.utils.log.AdminEmailHandler',
            'filters': ['special']
        }
    },
    'loggers': {
        'django.request': {
            'handlers': ['mail_admins', ],
            'level': 'ERROR',
            'propagate': True,
        },
        'allaccess': {
            'handlers': ['console', ],
            'level': 'INFO',
        }
    }
}
```

For more information on logging please see the Django documentation or the Python documentation.

Features

- Added access to simple API wrapper through the `AccountAccess` model
- Added state parameter for OAuth 2.0 by default
- Added basic error logging to OAuth clients and views
- Added contributing guide and mailing list info

v0.2.1 (2012-06-29)

Bug Fixes

- Fixes missing Content-Length header when requesting OAuth 2.0 access token

v0.2.0 (2012-06-24)

There are two South migrations included with this release. To upgrade you should run:

```
python manage.py migrate allaccess
```

If you are not using South, you will not need to change your database schema because the underlying field type did not change. However, you should re-save all existing `AccountAccess` instances to ensure that their access tokens go through the encryption step

```
from allaccess.models import AccountAccess

for access in AccountAccess.objects.all():
    access.save()
```

Features

- `OAuthRedirect` view can now specify a callback URL
- `OAuthRedirect` view can now specify additional permissions
- Context processor for adding enabled providers to the template context
- User access tokens are stored with AES encryption
- Documentation on customizing the view workflow behaviors
- Travis CI integration

Bug Fixes

- Fixed OAuth2Client to include `grant_type` paramater when requesting access token
- Fixed OAuth2Client to match current OAuth draft for access token response as well as legacy response from Facebook

Backwards Incompatible Changes

- Moving the construction on the callback from the client to the view changed the signature of the client `get_redirect_url`, `get_redirect_args`, `get_request_token` (OAuth 1.0 only) and `get_access_token` to include the callback. These are largely internal functions and likely will not impact existing applications.
- The `AccountAccess.access_token` field was changed from a plain text field to an encrypted field. See previous note on migrating this data.

v0.1.1 (2012-06-22)

- Fixed bug with passing incorrect callback parameter for OAuth 1.0
- Additional documentation on configuring `LOGIN_URL` and `LOGIN_REDIRECT_URL`
- Additional view tests
- Handled poor `LOGIN_URL` and `LOGIN_REDIRECT_URL` settings in view tests

v0.1.0 (2012-06-21)

- Initial public release.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Index