djangowind Documentation

Release 1.1.0

Columbia University CTL

Nov 13, 2018

Contents

djangowind is a Django authentication backend for Columbia University's CAS and WIND servers, including associated helpers.

CHAPTER 1

Basic Usage

(you will of course need django's built in auth, sessions, and sites apps installed. That's done for you on a default install but if you've changed things, you might need to re-enable those and do a syncdb. The current version of djangowind works with Django 1.0)

In your django app, you'll need to do a few things to enable it.

First, add 'djangowind' to INSTALLED_APPS in settings.py. Then add:

```
AUTHENTICATION_BACKENDS = ('djangowind.auth.WindAuthBackend','django.contrib.auth.
↪backends.ModelBackend',)
WIND_BASE = "https://wind.columbia.edu/"
WIND_SERVICE = "cnmtl_full_np"
```

to settings.py. 'django.contrib.auth.backends.ModelBackend' is django's standard built-in auth backend that checks the username/password against the database. The example config leaves that in as the second in the AUTHENTICATION_BACKENDS. That will let you use both WIND authentication and standard django database accounts in the same app. If you want to restrict things to *only* WIND, just take out the ModelBackend entry (be sure to leave it as a tuple or list). WIND_BASE and WIND_SERVICE aren't strictly necessary as those are the defaults in the code.

Djangowind uses the django.core.context_processors.request template context processor, so that needs to be enabled. So add the following to your settings as well:

```
TEMPLATE_CONTEXT_PROCESSORS = (
    'django.contrib.auth.context_processors.auth',
    'django.template.context_processors.debug',
    'django.template.context_processors.request',
)
```

Now, in urls.py, add the mapping:

```
('^accounts/',include('djangowind.urls')),
```

to your urlpatterns. This will keep all the auth stuff under '/accounts/' the same as the standard django auth. You can, of course, override that behavior by using different mappings (but be careful to also change the relevant templates if you change that). The only other required step for basic usage is to override the default login template (since we

need to include a "login through wind" button). Add a 'registration/login.html' template to your app's templates with content something like the following:

```
{% extends "base.html" %}
{% block content %}
{% if form.has_errors %}
<p>Your username and password didn't match. Please try again.</p>
{% endif %}
<form method="get" action="{{ wind_base }}login">
<input type="hidden" name="service" value="{{ wind_service }}" />
<input type="hidden" name="destination"
value="http://{{ request.get_url }}/accounts/windlogin/?next={{ next }}" />
<p>If you have a Columbia UNI, you already have an account and can
login through WIND with it</p>
<input type="submit" value="Here" />
</form>
<p>otherwise: </p>
<form method="post" action=".">
<table>
<tr><td><label for="id_username">Username:</label></td><td>{{ form.username }}</td></
↪tr>
<tr><td><label for="id_password">Password:</label></td><td>{{ form.password }}</td></
↪tr>
</table>
<input type="submit" value="login" />
<input type="hidden" name="next" value="{{ next }}" />
</form>
{% endblock %}
```

Alternatively, if that exact template code is suitable for you, you can just make sure that djangowind's template directory is in your TEMPLATE_PATHS. But you'll probably want to customize your login page. You'll also want to make sure that the domain is set correctly in the Sites table for your site. At this point, everything should basically work the same as with regular Django Auth (with obvious exceptions of password related things) and you can refer to the documentation. Ie, you can use a @login_required decorator on a view and the user will have to go through that login screen and login via WIND to access the resource. If you have extra fields that you want on user objects, you will probably want to read about UserProfile objects.

CHAPTER 2

Admin App Integration

The Django admin app requires a little additional work to get it to function properly with WIND auth. Basically, admin wants to use its own login template instead of auth's login template. So you need to make an 'admin/login.html' template in your templates directory. Admin doesn't pass in a 'next' variable in the context, so you need to set that to '/admin/' yourself to have them redirected to the admin interface when they log in. Once you get the admin template properly overridden, you should be able to login through WIND and, if your user is marked as staff or superuser, you'll be able to get into the admin interface.

CHAPTER 3

Additional Behavioral Configuration

The configuration above is very minimal. When the user logs in through WIND, it checks their ticket and, if valid, logs in the user. If there isn't an auth_user entry for that UNI, it will create one automatically and set it's password to django's "not a valid password" special value. first_name, last_name, etc will all be left blank and wind affils will be ignored. Most of this is changeable though.

djangowind includes two different hooks for helpers (and includes a couple basic helpers). There are profile handlers which are expected to take care of filling in last_name, first_name, and email fields when the user is first created and Affil Handlers which get to look at the WIND affils and take action based on them.

3.1 Profile Handlers

djangowind includes a CDAP based profile handler that will fill in last_name, first_name, and email fields on the user object by talking to our CDAP service. To use it, add the following to your app's settings.py:

```
WIND_PROFILE_HANDLERS = ['djangowind.auth.CDAPProfileHandler']
```

CDAPProfileHandler requires restclient to be installed and 'cdap.ccnmtl.columbia.edu' to be properly set up in your /etc/hosts. If we ever run the CDAP server at a different URL, you can change the host by setting CDAP_BASE.

If you want to write your own ProfileHandler, it's just a class with a process(self,user) method on it. It will be passed the user object which probably only has the username field set and will be expected to set the other fields on the user and call .save() on it. It doesn't need to return anything. They just put it somewhere in your PYTHONPATH and add it to (or replace the CDAPProfileHandler) WIND_PROFILE_HANDLERS. If there is more than one handler in WIND_PROFILE_HANDLERS, djangowind will go through them all in order and give each a chance to process the user object. So keep that in mind if you're chaining them.

Eg, a trivial Profile Handler that gives everyone slack would look like:

```
class SlackProfileHandler:
    def process(self,user):
        user.first_name = "Bob"
        user.last_name = "Dobbs"
```

(continues on next page)

(continued from previous page)

```
        user.email = "bob@subgenius.org"
        user.save()
```

And remember that the Profile Handlers are only called up when a new user is added (ie, they login for the first time and an entry isn't found in auth_users that matches their UNI).

3.2 Affil Handlers

What to do with wind affils?

Django Auth includes Groups so the natural thing would be to map wind affils directly to groups. That's often a good idea, but Django Auth groups only have one 'name' field so that would have to be set to match the wind affil string, which is ugly. Also, frequently we use wind affils to map students into "classes" which is conceptually a bit different from django auth groups. So we need a bit more flexibility than just always doing a one to one mapping.

But that's the simplest case, so we'll start there. djangowind includes a built-in AffilGroupMapper that does the simple one-to-one mapping. To enable it, add:

```
WIND_AFFIL_HANDLERS = ['djangowind.auth.AffilGroupMapper']
```

to settings.py. Again, WIND_AFFIL_HANDLERS is a list of handler classes. djangowind will go through them in order, giving them each a chance to do their thing if there's more than one in the list. Unlike Profile Handlers, Affil Handlers are run each time the user logs in through WIND. So if their affils change over time, it will get picked up at the next login.

WIND also returns an affil for each user that matches their UNI. I'm not sure exactly why they do that, but you probably don't want to have a django auth Group for every UNI that logs in. AffilGroupMapper strips that UNI group out automatically. If, for some reason, you really want those UNI groups, you can tell it not to strip them out by setting WIND_AFFIL_GROUP_INCLUDE_UNI_GROUP to True in your settings.py.

AffilGroupMapper also creates an 'ALL_CU' Group and places every user that comes in through WIND into that group. That's useful if you allow both WIND logins and regular django db backed logins and want an easy way to tell the two kinds of accounts apart.

AffilGroupMapper will only ever *add* the user to Groups, it will not remove them from Groups that aren't in the WIND affils. It's done this way because otherwise if you have extra django Groups that don't map to WIND affils, it has no way really of telling them apart and removing the user from only the right groups. This means that it's not a foolproof security technique to use this mapper to check that someone is in a class. If they login once when they're in a class, it will add them to that Group. Then, if they drop the class and log back in, that class won't be in their WIND affils, but they'll still be in that Group as far as Django is concerned. If you need something more secure, you can write your own mapper (see below).

There are two other Affil Handlers that djangowind includes that are useful. Rather than Groups, the django admin app uses two boolean fields on the User objects to determine access, is_staff, and is_superuser. It's often quite useful to mark a particular WIND affil (or affils) as admins and automatically give them access to the admin interface. Eg, we often use tlcxml for that purpose. djangowind includes StaffMapper and SuperuserMapper Affil Handlers to do this. Add them to your WIND_AFFIL_HANDLERS list to use them.

StaffMapper will look for a WIND_STAFF_MAPPER_GROUPS setting, which should be a list of WIND affil strings. Any user coming in that has any of those affils will get their is_staff boolean field set to True (and thus gain access to the admin interface).

SuperuserMapper does the same thing using a WIND_SUPERUSER_MAPPER_GROUPS setting and sets the is_superuser field to True. Superusers have access to the admin interface including the parts that let you edit other user

accounts and permissions (django probably made you create one the first time you did a syncdb with the admin or auth apps installed).

It's useful to remember that both StaffMapper and SuperuserMapper *don't* throw out the UNI matching groups like AffilGroupMapper does, so you can put UNIs in the groups lists to give specific users admin rights. Eg, if your settings.py has:

```
WIND_STAFF_MAPPER_GROUPS = ['tlcxml.cunix.local:columbia.edu']
WIND_SUPERUSER_MAPPER_GROUPS = ['anp8','jb2410']
```

Then all CCNMTL staff will have basic staff access and Anders and Jonah will have superuser access.

If those included helpers don't do everything you need, you can write your own and include it in WIND_AFFIL_HANDLERS. An Affil Handler is just a class with a map(self,user,affils) method. 'user' is the Django Auth User object, 'affils' is a list of wind affil strings. It's expected to do whatever needs to be done in terms of adding or removing groups and setting stuff on the user object.

Eg, a stricter version of AffilGroupMapper that makes the user *only* belong to the Groups that match their WIND affils would look like:

```
class StrictAffilGroupMapper:
    def map(self,user,affils):
        groups = []
        for affil in affils:
            try:
                group = Group.objects.get(name=affil)
            except Group.DoesNotExist:
                group = Group(name=affil)
                group.save()
            groups.append(group)
        user.groups = groups
        user.save()
```