django-windowsauth

Dan Yishai

Feb 20, 2021

INSTALLATION AND SETUP

Easy integration and deployment of Django projects into Windows Environments.

CHAPTER

ONE

FEATURES

- Deploy to Microsoft IIS quickly using wfastcgi and `createwebconfig` command
- Authenticate via IIS's Windows Authentication
- Authorize against Active Directory using ldap3 package
- Manage LDAP connections for easy integrations
- Debug using django-debug-toolbar
- **NEW** Create Task Schedulers for Django management commands

1.1 Quick Start

1. Install with `pip install django-windowsauth`
2. Run `py manage.py migrate windows_auth`
3. Add "fastcgi application" with `wfastcgi-enable`
4. Configure project settings:

```
INSTALLED_APPS = [
   "windows_auth",
]

MIDDLEWARE = [
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.auth.middleware.RemoteUserMiddleware',
    'windows_auth.middleware.UserSyncMiddleware',
]

AUTHENTICATION_BACKENDS = [
    "windows_auth.backends.WindowsAuthBackend",
    "django.contrib.auth.backends.ModelBackend",
]

WAUTH_DOMAINS = {
    "<your domain's NetBIOS Name> (EXAMPLE)": {
        "SERVER": "<domain FQDN> (example.local)",
        "SEARCH_BASE": "<search base> (DC=example,DC=local)",
        "USERNAME": "<bind account username>",
        "PASSWORD": "<bind account password>",
    }
}
```

(continues on next page)

(continued from previous page)

```
# optional
STATIC_URL = '/static/'
STATIC_ROOT = BASE_DIR / "static"

MEDIA_URL = '/media/'
MEDIA_ROOT = BASE_DIR / "media"
```

5. Generate **web.config** files with `py manage.py createwebconfig -s -m -w`.
6. Create new IIS Website from the project files

1.2 Installation and Setup

This is a **detailed** walk-through the *django-windowsauth* installation and setup process.
For easy and quick installation please refer to the *Quick Start* guide.

1.2.1 Install and Setup IIS

First, you may need to install IIS role. This can be done though the Control Panel > Add and Remove Programs > Install Features (`appwiz.cpl`) or via **Server Manager**.

Those are the features you should select:

1. Application / CGI
2. Security / Windows Authentication
3. (suggested) Performance Features / Dynamic Content Compression
4. (suggested) Health and Diagnostics / Request Monitor
5. (suggested) Health and Diagnostics / Tracing

Next you will need to unlock some configuration section to later use the `createwebconfig` management command.

To unlock configuration sections:

1. Open IIS Manager > Configuration Editor
2. Select section `system.webServer/handlers`
3. Click `Unlock section` on the right sidebar.
4. Repeat for sections `system.webServer/security/authentication/anonymousAuthentication` and `system.webServer/security/authentication/windowsAuthentication`.

Note: For more information visit the IIS Topic on Microsoft Docs: https://docs.microsoft.com/en-us/iis

1.2.2 Getting it

You can get django-windowsauth by using pip:

```
$ pip install django-windowsauth
```

If you want to install it from source, grab the git repository and run setup.py:

```
$ git clone https://github.com/danyi1212/django-windowsauth.git
$ python setup.py install
```

1.2.3 Installing

You will need to add the `windows_auth` application to the INSTALLED_APPS setting in you Django project settings file.

```
INSTALLED_APPS = [
    ...
    'windows_auth',
    ...
]
```

This will allow to execute the `createwebconfig` command, add the new model *LDAPUer* and register it's Django Admin page.

Next, you will need to run the `migrate` management command to create the new SQL table of the new models.:

```
$ py manage.py migrate windows_auth
```

Note: This will perform migrations only for **windows_auth** app. If you have other migrations pending, you may want to omit the **windows_auth** argument to perform all available migrations.

1.2.4 Configure

In order to receive correctly the authenticated user from the **IIS Windows Authentication**, you will need to add a middleware called `RemoteUserMiddleware`. This middleware must be after `AuthenticationMiddleware`, that is usually provided by default with Django's `startproject` template.

```
MIDDLEWARE = [
    ...
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.auth.middleware.RemoteUserMiddleware',
    'windows_auth.middleware.UserSyncMiddleware',
    ...
]
```

To process the information passed from the **IIS Windows Authentication** and translate it into a **Django User**, you will need to specify the `WindowsAuthBackend` authentication backend.

```
AUTHENTICATION_BACKENDS = [
    'windows_auth.backends.WindowsAuthBackend',
    'django.contrib.auth.backends.ModelBackend',
]
```

Note: Be aware, this configuration keeps the Django's default **ModelBackend** in order to allow for fallback to **Django Native Users**. It can be used to authenticate without IIS, when using the `runserver` management command for example.

This is usually not advised to configure for **Production** setups, but only for **Development**.

See also:

Django documentation about *Authenticating using REMOTE_USER* https://docs.djangoproject.com/en/3.1/howto/auth-remote-user/

Next you will need to configure the settings for your **Domain** to allow for LDAP integration with **Active Directory**.

```
WAUTH_DOMAINS = {
    "EXAMPLE": {  # this is your domain's NetBIOS Name, same as in "EXAMPLE\\username
↪" login scheme
        "SERVER": "example.local",  # the FQDN of the DC server, usually is the FQDN␣
↪of the domain itself
        "SEARCH_BASE": "DC=example,DC=local",  # the default Search Base to use when␣
↪searching
        "USERNAME": "EXAMPLE\\bind_account",  # username of the account used to␣
↪authenticate your Django project to Active Directory
        "PASSWORD": "<super secret>",  # password for the binding account
    }
}
```

See also:

About LDAP Search Base: https://docs.microsoft.com/en-us/windows/win32/ad/binding-to-a-search-start-point

(optionally) Configure **file path** and **url path** settings for your `static` and `media` files.

```
STATIC_URL = '/static/'
STATIC_ROOT = BASE_DIR / "static"

MEDIA_URL = '/media/'
MEDIA_ROOT = BASE_DIR / "media"
```

You may need to execute `$ py manage.py collectstatic` management command after modifying the `STATIC_ROOT` setting.

See also:

Full how-to guide to *Serve Static Files through IIS*

1.2.5 Setup Logging

Throughout this whole module, logging is done to logger named `wauth`. You may handle and configure this logger through Django's setting `LOGGING`.

This can be done by adding the logger like so:

```
'wauth': {
    'handlers': ['console', 'file', 'mail_admins'],
    'level': 'INFO',
    'propagate': False,
},
```

Additionally, you may want to configure logging for `ldap3`. You can add this logger:

```
'ldap3': {
    'handlers': ['console', 'ldap'],
    'level': 'DEBUG',
    'propagate': False,
}
```

And make sure to configure `ldap3` log type, like this:

```
from ldap3.utils.log import set_library_log_detail_level, BASIC
set_library_log_detail_level(BASIC)
```

The lines above can be added in your Django settings file, just after the `LOGGING` setting. Remember to document about that in your code!

See also:

More information of that on https://ldap3.readthedocs.io/en/latest/logging.html

For your convenience, those are the handles used in the examples above:

```
'handlers': {
    'console': {
        'class': 'logging.StreamHandler',
        'level': 'WARNING',
    },
    'file': {
        'level': 'INFO',
        'class': 'logging.handlers.RotatingFileHandler',
        'maxBytes': 2 ** 20 * 100,  # 100MB
        'backupCount': 10,
        'filename': BASE_DIR / 'logs' / 'debug.log',
    },
    'ldap': {
        'level': 'INFO',
        'class': 'logging.handlers.RotatingFileHandler',
        'maxBytes': 2 ** 20 * 100,  # 100MB
        'backupCount': 10,
        'filename': BASE_DIR / 'logs' / 'ldap.log',
    },
    'mail_admins': {
        'level': 'ERROR',
        'class': 'django.utils.log.AdminEmailHandler',
        'include_html': True,
    },
},
```

Note: You will need to configure settings for sending emails to use the `mail_admins` handler: https://docs.djangoproject.com/en/3.1/topics/email/

1.2.6 Publish to IIS

First, we will need to create the `web.config` files for the IIS Website configuration. This can be done simply by running the management command::

```
$ py manage.py createwebconfig -s -m -w
```

Notice the `-s` and `-m` switches, those are to add configurations for **Serving Static Files though IIS**. You may want to omit those switches if you are not planning to serve static files though IIS.

The `-w` parameter configures IIS's `Windows Authentication` and disables `Anonymous Authentication` in the `web.config` file. You may want to change those settings manually to avoid **unlocking those configuration sections**.

See also:

Reference for `createwebconfig` at *Management Commands*

Next you will need to create a new IIS Website for your Django Project.

1. Open **IIS Manager**
2. Right-click over **sites**
3. Click **Add website...**
4. Give a **name** for your site (should use the same as for your Django project)
5. Specify **Physical path** for the root of your Django project folder (where the `manage.py` is)
6. Provide **binding information** as needed (can be changed later)

Congratulation, now you should be able to browse to your new website!

Next are some things to setup and verify before publishing to production...

1.3 Deployment Checklist

Before deploying your site to production it is important to go over some best practices and make sure your site is the **most stable and secure**. Provided here are some best practices related to `django-windowsauth`, IIS and LDAP.

See also:

Check out Django's deployment checklist too.

1. **Turn DEBUG off**. Make sure to never get it active on a production setup.
2. Store your **secrets** in a secure location. Here is a tutorial about *Managing Secrets*.
3. Use a proper **cache backend**, and use `WAUATH_USE_CACHE` for better performance. More about Django's cache framework
4. Use a production ready **database backend**, not SQLite. django-mssql-backend is a great backend for Microsoft SQL Server.
5. Configure `ALLOWED_HOSTS` and `CSRF_TRUSTED_ORIGINS` to exactly same as your **IIS Bindings**.
6. Setup Django **logging** and Admin Error Reporting for your project. See more https://docs.djangoproject.com/en/3.1/topics/logging/.
7. Enable and configure **IIS Logging**.
8. Keep your site files on a **separate drive** from the OS. Consider doing the same for logs and media.

9. Minimize to bare minimum permissions for the `web.config` files throughout your site.
10. Configure **HTTPS bindings** for your website with a CA signed certificate.
11. Use **only HTTPS** for your site, and configure HTTPS redirection with IIR Rewrite. Check out the `--https` flag for the `createwebconfig` command.
12. Use only **IIS Windows Authentication** when possible.
13. Protect your Django view using `@login_required` decorator and other authorization logics.
14. Use SSL and NTLM or Kerberos authentication for your LDAP connection. See *Securing LDAP Connections*.
15. Minimize the `SESSION_COOKIE_AGE` time and enable `SESSION_EXPIRE_AT_BROWSER_CLOSE` when using Windows Authentication as SSO. We recommend using 86400, 1 day in seconds.
16. *Customize Error Pages* for a better user experience.
17. Configure recycling times for your Application Pool at the least used time of the day.
18. Consider increasing the Maximum Worker Processes in your Application Pool to accommodate for heavy loads.
19. Setup Request Filtering to your site to limit unintended file access. You should deny access to ".py" and ".config" file extensions.
20. Enable dynamic IP restrictions based of requests/ms.

See also:

Some more great best practices for IIS are available at https://techcommunity.microsoft.com/t5/core-infrastructure-and-security/iis-best-practices/ba-p/1241577

1.4 Migration

1.4.1 From existing project

This is a very quick how-to for integrating `django-windowsauth` into existing Django projects.

First of all you will need to check your Django User's username field. In case it already matches their Active Directory logon name, you are good to go.

If that is not the case, you will probably need to change that. The process can look like:

1. Export the Django user table from your DB, and load it to excel.
2. Export the Active Directory Users to excel.
3. Merge tables via excel.
4. Import new user table into you your Django users table in your DB.

Next, you will need to manually create a `LDAPUser` model entry for each relevant user. This can be done via Django's shell:

```
$ py manage.py shell
```

```
>>> from django.contrib.auth.models import User
>>> from windows_auth.models import LDAPUser
>>> users = User.objects.filter().all()
>>> LDAPUser.objects.bulk_create(LDAPUser(domain="EXAMPLE", user=user) for user in␣
↪users)
```

You may want to **modify the user queryset** to create LDAP Relations only for specific users. In case you are using a **different Django User Model**, you will need to use it instead of the Django `User` or use `get_user_model()` method.

At this point, when a user first visit your site after migration, it **will be synchronized** against LDAP. In case you would still want to **also** migrate all users now, you can do this via Django's shell like so:

```
>>> from windows_auth.models import LDAPUser
>>> for user in LDAPUsers.objects.all():
>>>     user.sync()
```

1.4.2 To 1.4.0

- (optional) Remove duplicated groups between `SUPERUSER_GROUPS`, `STAFF_GROUPS` and `ACTIVE_GROUPS` ldap settings, or set `PROPAGATE_GROUPS` to `False`.

1.4.3 To 1.3.0

No required changes for migration is needed.

1.4.4 To 1.2.0

- Add the `UserSyncMiddleware` to `MIDDLEWARE` setting like so:

```
MIDDLEWARE = [
    ...
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.auth.middleware.RemoteUserMiddleware',
    'windows_auth.middleware.UserSyncMiddleware',
    ...
]
```

1.5 Serve Static Files through IIS

Generally websites have static files such as CSS, JS, Images served to clients beside the primary responses. Those files are considered as "Static Files" because they can be delivered without being generated, modified or processed.

In Django, static files can be served by the Django Framework itself. This is very convenient during **Development**, but is not suitable for **Production** use.

See also:

About Serving Static Files: https://docs.djangoproject.com/en/3.1/howto/static-files/

For production use, it is advised to let the **Web Server** to serve the Static Files. This is how it can be done:

Note: This how-to describes serving both **Static Files** and **Media Files**. In case you don't need or use one of those features, you can just ignore the respective parts in the tutorial.

First you will need to configure the following settings:

```
STATIC_URL = '/static/'
STATIC_ROOT = BASE_DIR / "static"

MEDIA_URL = '/media/'
MEDIA_ROOT = BASE_DIR / "media"
```

The `STATIC_URL` represents the file path **over HTTP**, while the `STATIC_ROOT` directs to the **Physical path** of the files in the Web Server's OS. Meanwhile from the **IIS** point of view, the **HTTPS path** is derived from the file's **Physical path** location. Although this can be altered using **Virtual Directories**, it is usually advised not to.

The same applies for the `MEDIA_URL` and `MEDIA_ROOT` settings.

Next we will need to create `web.config` files in each folder to configure IIS to server Static Files.

Note: Any time the `STATIC_ROOT` setting is changes, you will need to start over from this step.

This can be done by running the `createwebconfig` management command::

```
$ py manage.py createwebconfig -s -m
```

The `-s` switch is used configure the `STATIC_ROOT` folder, while `-m` switch is used to configure the `MEDIA_ROOT` folder.

Now all we need to do is to **collect** all the static files from the many Django apps into the `STATIC_ROOT` folder. This can be done by running the `collectstatic` management command::

```
$ py manage.py collectstatic
```

See also:

About `collectstatic` command: https://docs.djangoproject.com/en/3.1/ref/contrib/staticfiles/#django-admin-collectstatic

At this point, in case you have configured the **URL path** and **Physical path** the same, the Web Server should serve all static files correctly.

In case you have configured **different paths**, you will probably want to setup **Virtual Directories**.

This can be useful when you want to store Static and / or Media file **outside** the Django project's folder (the website's root folder), on a separate disk for example.

To create the Virtual Directories:

1. Open **IIS Manager**
2. Right-click on **your website**
3. Click **"Add Virtual Directory…"**
4. Set the **"Alias"** for the same value as `STATIC_URL` setting
5. Set the **"Physical Path"** for the same value `STATIC_ROOT` setting

You may do the same with the `MEDIA_URL` and `MEDIA_ROOT` settings in order to add Virtual Directory for serving **Media Files**.

See also:

Microsoft Docs on IIS Virtual Directories https://docs.microsoft.com/en-us/iis/get-started/planning-your-iis-architecture/understanding-sites-applications-and-virtual-directories-on-iis#virtual-directories

1.6 Create Task Scheduler Jobs

It is usually necessary to **execute tasks** from you project on a schedule. This module has a shortcut to create scheduled jobs for **Django management commands** in the **Windows Task Scheduler**.

> **Warning:** This feature requires the installation of the `pywin32` module. Install it with `pip install pywin32`

1.6.1 Create a task

Creating a new task is done with the `createtask` command.

For example, lets say you want to run the following command every hour:

```
$ py manage.py say_hello --new-users
```

You can create a schedule with this command:

```
$ py manage.py createtask "say_hello --new-users" -i hours=1
```

Now the following command will be executed every hour by the Windows Task Scheduler.

1.6.2 Using predefined tasks

Included with this module are some **predefined tasks** for some Django and Third-Party app management commands. Those commands can be created using the `--predefined` or `-p` argument

clearsessions Clear expired sessions from database, once a week:

```
$ py manage.py createtask clearsessions -p
```

See also:

See more at https://docs.djangoproject.com/en/3.1/ref/django-admin/#django-admin-clearsessions

clean_duplicate_history Clean duplicate history records from all models with history every 3 hours (from django-simple-history):

```
$ py manage.py createtask clean_duplicate_history -p
```

clean_old_history Clean history records older then 30 days from all models with history every day (from django-simple-history):

```
$ py manage.py createtask clean_old_history -p
```

See also:

See more at https://django-simple-history.readthedocs.io/en/latest/utils.html#utils

process_tasks Worker for background tasks processing (from django-background-tasks):

```
$ py manage.py createtask process_tasks -p
```

You can also create **multiple workers** by specifying different names with `--name` argument.

See also:

See more at https://django-background-tasks.readthedocs.io/en/latest/#running-tasks

1.7 Using Custom User Model Mappings

Sometimes it is useful to modify the default Django User Model, to change field settings or to add extra fields. When doing so, you may want to customize the LDAP synchronization to accommodate for the extra fields and changes.

For the sake of this tutorial, we will use the following custom User model as an example:

```
class CustomUser(AbstractBaseUser):
    telephone = models.CharField(max_length=32)
    country_code = models.PositiveSmallIntegerField()

    job_title = models.CharField(max_length=64)
    department = models.CharField(max_length=64)

    REQUIRED_FIELDS = ("telephone", "job_title", "department")
```

Usually the first thing to do is to change the `USER_FIELD_MAP` in the LDAP Setting for all domains.

You can configure it for each domain, for example:

```
WAUTH_DOMAINS = {
    "EXAMPLE": {
        "SERVER": "example.local",
        "SEARCH_BASE": "DC=example,DC=local",
        "USERNAME": EXAMPLE\\bind_account",
        "PASSWORD": "<super secret>",
        "USER_FILED_MAP": {
            "username": "sAMAccountName",
            "first_name": "givenName",
            "last_name": "sn",
            "email": "mail",

            "telephone": "telephoneNumber",
            "country_code": "countryCode",
            "job_title": "title",
            "department": "department",
        }
    },
}
```

Or create a custom LDAP Settings with your defaults:

```
@dataclass()
class MyLDAPSettings(LDAPSettings):
    USER_FIELD_MAP = {
        "username": "sAMAccountName",
        "first_name": "givenName",
        "last_name": "sn",
        "email": "mail",

        "telephone": "telephoneNumber",
```

(continues on next page)

(continued from previous page)

```
        "country_code": "countryCode",
        "job_title": "title",
        "department": "department",
    }

WAUTH_DOMAINS = {
    "EXAMPLE": MyLDAPSettings(
        SERVER="example.local",
        SEARCH_BASE="DC=example,DC=local",
        USERNAME="EXAMPLE\\bind_account",
        PASSWORD="<super secret>",
    ),
}
```

See also:

Reference for `USER_FIELD_MAP` LDAP Setting at *LDAP Settings*

1.8 Using LDAP in your code

In addition to just windows authentication and IIS integration, this module provide you with an easy to use LDAP connection interface.

Throughout your code, you can use the `get_ldap_manager` function to receive an `LDAPManager` manager object fora specified domain. For example:

```
from windows_auth.ldap import get_ldap_manager

manager = get_ldap_manager("EXAMPLE")
manager.connection
```

With the manager, you can access the ldap3 `Connection` object, perform LDAP operations like search, add, modify, etc.

Also, you can use the **ldap3 Abstraction Layer** for a simple python interface. This is how you can use it to query all Active Directory Computer objects:

```
reader = manager.get_reader("computer")
reader.search("name")
```

And even write to LDAP, like this:

```
from ldap3 import Writer

writer = Writer.from_cursor(reader)
computer = writer.match("name", "test_computer")[0]
computer.description = "Hello world!"
writer.commit()
```

Note: For Security reasons, the LDAP connections are **read-only** by default. In order to write to LDAP, you will need to configure `READ_ONLY=False` in the LDAP Settings of each desired domain.

1.8.1 Advice for Model - LDAP relations

Sometimes it is useful to relate a **Django Model object** to an **LDAP object**. When doing so, you can easily implement **synchronization**, and enable easy access for **LDAP operation** on that object. If you plan to do such thing, here are some tips for you:

Store the **LDAP Domain** in which the object is from, either as a **class-level const**, or a **Model Field**. You may want to event implement a `get_ldap_manager` method get the manager for the respective domain.

```
class Computer(models.Model):
    # as a const
    DOMAIN = "EXAMPLE"

    # as a field
    domain: str = models.CharField(max_length=128)

    # using a method
    def get_ldap_manager(self) -> LDAPManager:
        return get_ldap_manager(self.domain)
```

Then implement a method to receive the exact entry for the related LDAP object.

```
class Computer(models.Model):
    # ...
    name: str = models.CharField(max_length=128)

    def get_ldap_computer(self, attributes: Optional[Iterable[str]] = None) -> Entry:
        manager = self.get_ldap_manager()
        reader = manager.get_reader("computer", f"name: {self.name}", attributes)
        return reader.search()[0]
```

1.9 Managing Secrets

This tutorial is still in the process of writing...

1.10 Securing LDAP Connections

When using your project with a production LDAP server, you should always use a secure connection. The LDAP connections probably will include sensitive information from your domain, like the credentials of the service account that is begin used by your Django Project, user information of the users on your site and any other custom uses of LDAP in your code.

Securing the connection to your LDAP is somewhat easy, yet still requires some prerequisites and configurations. Here you will be informed of some practices you can do to better secure your LDAP connections.

Note:

The information provided herein is intended to provide helpful and informative material as it is related to security equipment and services.

It is not intended to be taken as legal, accounting, investment, or other professional advice.

If you require personal assistance or advice, be sure to consult with a competent professional.

We disclaim any responsibility for any liability, loss or risk, personal or otherwise, which is incurred as a consequence, directly or indirectly, of the use and application of any answers provided here or in any of our material.

1.10.1 Using SSL/TLS

SSL/TLS use certificates to establish a secure connection between you and the LDAP service before any data is exchanged. This practice is called LDAPs, and refers for LDAP over TLS or LDAP over SSL.

To use LDAPs, it needs to be enabled on the LDAP service side. Each LDAP service has a different setup required to enable LDAPs, you can search docs for your case.

For Active Directory administrators, here is a great guide to enable it for testing: https://techexpert.tips/windows/enabling-the-active-directory-ldap-over-ssl-feature/

Once you have enabled LDAPs on the server, you just need to configure the LDAP Setting `USE_SSL` to True.

```
WAUTH_DOMAINS = {
    "EXAMPLE": LDAPSettings(
        SERVER="example.local",
        SEARCH_BASE="DC=example,DC=local",
        USERNAME="EXAMPLE\\bind_account",
        PASSWORD="<super secret>",
        USE_SSL=True,
    ),
}
```

Warning: This module uses LDAPs by default to provide an easier setup. In case your LDAP servers are not capable of LDAPs, you should configure the LDAP setting `USE_SSL` to False.

1.10.2 Using NTLM Authentication

NTLM is a protocol used to securely exchange credential information between the client and the server. It is done by hashing the password with a random generated number provided by the server before sending.

NTLM was originally created by Microsoft to be used in the Windows ecosystem. It is still in use today, yet it is considered outdated, and has been mainly replaced with Kerberos.

See also:

See more detailed explanation about NTLM

To enable NTLM authentication, you can specify connection's `authentication` options in the `CONNECTION_OPTIONS` LDAP setting.

```
WAUTH_DOMAINS = {
    "EXAMPLE": LDAPSettings(
        SERVER="example.local",
        SEARCH_BASE="DC=example,DC=local",
        USERNAME="EXAMPLE\\bind_account",
        PASSWORD="<super secret>",
        USE_SSL=True,
        CONNECTION_OPTIONS={
            "authentication": ldap3.NTLM,
        },
    ),
}
```

See also:

NTLM authentication on ldap3 docs https://ldap3.readthedocs.io/en/latest/bind.html#ntlm

1.10.3 Using Kerberos (SASL)

Kerberos is an authentication and authorization protocol designed by MIT in the late '80s. Today, kerberos is the gold standard authentication and authorization protocol used throughout Windows and other OSs. It uses tickets to represent the authenticated user and to authorize it to access desired services.

See also:

Learn more about Kerberos at https://www.simplilearn.com/what-is-kerberos-article

When using kerberos authentication, the credentials given to the LDAP server is the account's kerberos token accessed from the MIT Token Manager. Therefore, the account running your Django Project will be used when accessing LDAP servers, and no username or password needs to be provided.

In order to use the ldap3 `SASL` authentication with the `KERBEROS` mechanism, you will need to install the `gssapi` package. To install it you first need to install the MIT Kerberos on the server.

Go to https://web.mit.edu/KERBEROS/dist/ and download the latest MIT Kerberos for Windows as 64-bit MSI Installer, and install it on the server. Restart will be required after installation is done.

After the restart, edit the `C:\ProgramData\MIT\Kerberos5\krb5.ini` and provide the `default_realm` setting. For example:

```
[libdefaults]
    default_realm = EXAMPLE.LOCAL
```

See also:

More about the `krb5.ini` config file https://web.mit.edu/kerberos/www/krb5-latest/doc/admin/conf_files/krb5_conf.html

Then, install the `gssapi` package in your virtualenv:

```
$ pip install gssapi
```

Then configure your LDAP connections to use the `SASL` authentication with `KERBEROS` mechanism like so:

```
WAUTH_DOMAINS = {
    "EXAMPLE": LDAPSettings(
        SERVER="example.local",
        SEARCH_BASE="DC=example,DC=local",
        USERNAME="",
        PASSWORD="",
        USE_SSL=True,
        CONNECTION_OPTIONS={
            "authentication": ldap3.SASL,
            "sasl_mechanism": ldap3.KERBEROS,
        }
    ),
}
```

Now you need to get the ticket for that account configured though MIT Token Manager.

Note: Notice the username and password kept as empty strings as they are not necessary in this setup.

1.10.4 Optimize your code

Securing the LDAP connection in at the protocol level is good, but do not let it deceive you. It is very important to restrict any unintended operation on the LDAP server, especially write operations.

Minimize the permissions and delegations of the bind account to the bare minimum possible. You can never know how and what could be done through vulnerabilities in your code.

Never ever write user password or other credentials explicitly inside your code. Use instead another way to store your secrets in a protected place. See the tutorial about *Managing Secrets*

Use ``Reader`` and ``Writer`` cursors from ldap3's abstraction module. Using them can help you to avoid unwanted behaviors by simplifying the interface.

Restrict access to views performing LDAP operations. Allow only authenticated users, and implement permission check to avoid compromising your views.

Use read-only connection when possible. By default, LDAP connections are made read-only. It restricts the execution of write operations at the client level.

In case you need to perform write operations, you will need to explicitly disable read-only. When doing so, consider **creating a dedicated connection** for writing, with a **different bind account** with the minimal permissions. This can be done by adding another domain to `WAUTH_DOMAINS` setting for the same domain, but with different account and read-only disabled.

1.11 Customize Error Pages

This tutorial is still in the process of writing...

1.12 Debug with django-debug-toolbar

When using LDAP throughout your project, it is useful to see what operations did perform on the server side. This can be done using the `django-debug-toolbar` with the *LDAP Panel`* provided with this module.

This panel shows you the metrics for each domain (if you have configured them to collect metrics), and every operation the server perform against the LDAP server, including the LDAP filter and every each entry it responded with.

1.12.1 Installation

In order to view the LDAP Panel to the debug toolbar, you will need to install the `django-debug-toolbar` package:

$ pip install django-debug-toolbar

And follow the installation guide on `django-debug-toolbar` docs https://django-debug-toolbar.readthedocs.io/en/latest/installation.html

Then to add the LDAP Panel, insert it to the DEBUG_TOOLBAR_PANELS setting like so:

```
DEBUG_TOOLBAR_PANELS = [
    # ...
    'windows_auth.panels.LDAPPanel',
    # ...
]
```

To enable all of the LDAP Panel feature, you may want to enable metrics collection for all your domains:

```
WAUTH_DOMAINS = {
    "EXAMPLE1": LDAPSettings(
        SERVER="example.local",
        SEARCH_BASE="DC=example,DC=local",
        USERNAME="EXAMPLE\\bind_account",
        PASSWORD="<super secret>",
        COLLECT_METRICS=True,
    ),
    "EXAMPLE2": {
        "SERVER": "example.local",
        "SEARCH_BASE": "DC=example,DC=local",
        "USERNAME": "EXAMPLE\\bind_account",
        "PASSWORD": "<super secret>",
        "COLLECT_METRICS": True,
    },
}
```

1.13 Collect Metrics

Sometimes collecting metrics and usage data can be very helpful in detecting mistakes and problems. Using `ldap3` Connection Metrics system, you are able to get an inside look about the connections.

1.13.1 Installation

First, you will need to add the `ldap_metrics` app to the `INSTALLED_APPS` setting:

```
INSTALLED_APPS = [
    ...
    'windows_auth',
    'windows_auth.ldap_metrics',
    ...
]
```

Next you will need to migrate to create the LDAP Usage table:

```
$ py manage.py migrate ldap_metrics
```

1.13.2 Usage

In order to start collecting usage metrics, you will configure the `COLLECT_METRICS` LDAP Setting for each domain. For example:

```
WAUTH_DOMAINS = {
    "EXAMPLE": LDAPSettings(
        SERVER="example.local",
        SEARCH_BASE="DC=example,DC=local",
        USERNAME="EXAMPLE\\bind_account",
        PASSWORD="*********",
        COLLECT_METRICS=True,
    ),
}
```

Now, every time a Django process exists, the LDAP Connection usage metrics will be saved. The connection metrics can be viewed in your Django project's admin site.

Note: In case you want to collect metrics only when developing, you can set this setting to `DEBUG`.

1.14 Settings

1.14.1 WAUTH_USE_SPN

Type `bool`; Default to `False`; Not Required.
Expect the REMOTE_USER header value to be in Windows SPN username scheme.

By default, IIS will present the authenticated user by it's Down-Level Logon Name, for example "EXAMPLE\username". Setting this value to `True` will will expect the authenticated user to be presented by it's User Principal Name, for example "username@example.local".

Note: When using `SPN` the domain of the authenticated user will be detected by the **Domain's FQDN** instead of it's **NetBIOS Name**!

This means that you will need to configure `WAUTH_DOMAINS` by created with the FQDN of their domain, and not their NetBIOS Name. This is also means all new LDAPUser domain values will be FQDNs and not NetBIOS Names

If you are planning to migrate between using Down-Level to SPN, first of all **don't**. In case you still need to switch between them, you can either **manually replace** the LDAPUser's domain values from the old NetBIOS Names to the new FQDNs, or just **delete** all LDAPUsers and let them be created again when a user login again after change.

1.14.2 WAUTH_DOMAINS (Required)

Type `dict`; Default to `None`; Required.
LDAP Settings for each domain.

Dictionary of domain NetBIOS Names and their settings for LDAP connection. Domain LDAP Settings can be written as a dictionary with the settings in UPPERCASE and their values, or as an `LDAPSettings` object.

A default domain settings can be used as a fallback settings for requested domains that are missing from `WAUTH_DOMAINS` by using **"__default__"** as the domain name. When using only the default domain settings, you may want to specify manually the `WAUTH_PRELOAD_DOMAINS` setting.

Each of the domain settings can be configured as a **function** that will be used as callback when accessing the setting and be called with the **domain as it first argument**. This can be used with `lambda` functions for lazy setting values.

See also:

More information about domain LDAP Settings can be found at *LDAP Settings* reference.

1.14.3 WAUTH_RESYNC_DELTA

Type `timedelta`, `str`, `int` or `None`; Default to `timedelta(days=1)`; Not Required.
Minimum time (seconds) until automatic re-sync user's fields and permissions against LDAP.

Configure when to **automatically synchronize** the user's fields and groups (and permissions) against Active Directory via LDAP. On each request the user makes, if the user **haven't synchronized** in the time specified, the `WindowsAuthBackend` attempt to perform synchronization again on the user. This is used to make sure the user permissions and properties match those in Active Directory.

The value is used as **number of seconds** in `int`, `str` or any other object that can be casted to `int`. The value can also be a `django.utils.timezone.timedelta` object.

In case you need to synchronize the user on every request, you can configure the setting to `0`.
To disable automatic synchronizations via LDAP, you can remove the `UserSyncMiddleware` or configure the setting to `None` or `False`.

Note: Synchronizing user via LDAP can delay the Request / Response processing by only few ms, but your experience may vary. You can debug your setup using *Debug with django-debug-toolbar*.

1.14.4 WAUTH_USE_CACHE

Type `boot`; Default to `DEBUG`, otherwise `False`; Not Required.
Use cache backend instead of DB for determining user re-sync.

When using user automatic synchronization, the check whether user requires a re-sync is verified against the `LDAPUser` model and it requires an SQL Query.

To avoid this query and allow for better performance, this setting can allow you to use Django's cache framework instead of the default model verification against the DB. This will require you to setup your cache backend in setting `CACHES`.

In production, it is advised to use the cache setting instead of the default model based verification.

1.14.5 WAUTH_REQUIRE_RESYNC

Type `boot`; Default to `False`; Not Required.
Raise exception and return Error 500 when user failed to synced to domain.

When using user automatic synchronization, propagate any exception raised during synchronization. This will result with the user receiving a **Error 500** when they fail to synchronize properly.

This is useful for security sake, when **requiring** users to have the most updated fields and permissions. While developing in debug, it is usually useful to **receive information** about the synchronization exception.

In any case, the synchronization exception **will be logged** as error with the exception information included. If you have setup logging and email reporting for server admins, you can also **receive the exception details by email**.

Note: You can configure this per view with the `ldap_sync_required` decorator. See the reference at *View Decorators*

1.14.6 WAUTH_ERROR_RESPONSE

Type `int` or `Callable`; Default to `None`; Not Required.
Configure custom HTTP Response for Errors while User automatic LDAP Synchronization.

When a user synchronization fails, you can define a **custom HTTP Response** to send to clients.

This can be configured as a `int`, it is used as the **Response Code** for response with the default text `Authorization Failed`. This also can be a **function** that receive the `request` and `exception` as first and second arguments, and returning a Django `HttpResponse` object.

When configured to `None` the exception is propagated, and usually results in a **Error 500** for clients.

Note: This setting is only relevant when `WAUTH_REQUIRE_SYNC` is set to `True`, otherwise the **exception will be ignored**.

1.14.7 WAUTH_LOWERCASE_USERNAME

Type `boot`; Default to `True`; Not Required.
Lowercase the username to mimic non-case sensitive LDAP backends like Active Directory.

Windows systems, like Active Directory are **non-case sensitive**. While python, Django, and most Databases are **case sensitive**, you can lower case every username to **mimic** the non-case sensitive behavior of the Windows system.

1.14.8 WAUTH_IGNORE_SETTING_WARNINGS

Type `boot`; Default to `True`; Not Required.
Skip verification of domain settings on server startup.

By default, on every startup of you Django project the settings are validated.

This setting can be used to ignore the warnings raised by detecting users with domains missing from settings in `WAUTH_DOMAINS`, and **Unknown Settings** detected in domain LDAP Settings.

1.14.9 WAUTH_PRELOAD_DOMAINS

Type `tuple` or `bool`; Default to `None`; Not Required.
List of domains to preload and connect during Django project startup

LDAP Connections are **cached in process memory** to retain connections for multiple request / response cycles. This setting lists the domains to preload, connection and bind during you **Django project startup**. This way, the first request for a process will not have wait extra time for the LDAP connection to load and connect.

When the setting is configured to `None` or `True`, all the domains configured in `WAUTH_DOMAINS` settings are **preloaded**. In case you use only the **default domain settings** in the `WAUTH_DOMAINS` setting, it is advised to **manually** configure this setting to preload the relevant domains.

To enable LDAP Connection **lazy loading**, you can set this setting to `False`.

Note: When using `runserver` command, due to the server first **validating models** before loading the project, it may seam like **multiple connections** get initiated for the same domains.

By setting this setting, it may cause **multiple LDAP connections** to be established and terminate quickly for each domain.

You should **not be warned** by this behavior as this is behaves like a **quick connection test** to your LDAP server, and this is should only happened during **development phase**. In case you would like to **avoid this behavior** anyway, you can use the `runserver --noreload` parameter, or modifying the `WAUTH_PRELOAD_DOMAINS` setting to `False` when debugging.

1.15 LDAP Settings

LDAP Settings are the settings used to configure **LDAP connection** to domains. They are configured inside the `WAUTH_DOMAINS` setting of your Django project settings file, as the **value** for each domain key.

1.15.1 Configuring

LDAP Settings can be represented as a regular **Python Dictionary**, like this:

```
WAUTH_DOMAINS = {
    "EXAMPLE": {
        "SERVER": "example.local",
        "SEARCH_BASE": "DC=example,DC=local",
        "USERNAME": "EXAMPLE\\bind_account",
        "PASSWORD": "*********",
    }
}
```

Or as an LDAPSettings object, like this:

```
WAUTH_DOMAINS = {
    "EXAMPLE": LDAPSettings(
        SERVER="example.local",
        SEARCH_BASE="DC=example,DC=local",
        USERNAME="EXAMPLE\\bind_account",
```

(continues on next page)

(continued from previous page)

```
        PASSWORD="*********",
    ),
}
```

When using a **Python Dictionary**, each setting can be configured to a **callback function** that will be called with the specified domain as first and only argument. For example:

```
WAUTH_DOMAINS = {
    "EXAMPLE": {
        "USERNAME": lambda domain: f"{domain}\\bind_account",
    }
}
```

1.15.2 Using defaults

Sometimes when using multiple domains it is easier to configure settings **globally** or to **specify defaults** for unanticipated domains.

When configuring LDAP Settings as a **Python Dictionary**, this can be done by using the `"__default__"` key in `WAUTH_DOMAINS` settings. Every setting configured in the `"__default__"`, and are **not configured explicitly** for the domain, in will propagate. For example:

```
WAUTH_DOMAINS = {
    "__default__": {
        "USE_SSL": True,
    },
    "EXAMPLE1": {
        "SERVER": "example.local",
        "SEARCH_BASE": "DC=example,DC=local",
        "USERNAME": "EXAMPLE\\bind_account",
        "PASSWORD": "*********",
        "USE_SSL": False,
    },
    "EXAMPLE2": {
        "SERVER": "example.local",
        "SEARCH_BASE": "DC=example,DC=local",
        "USERNAME": "EXAMPLE\\bind_account",
        "PASSWORD": "*********",
    }
}
```

In this case, `EXAMPLE1` will have `USE_SSL = False` and `EXAMPLE2` will have `USE_SSL = True`.

When using `LDAPSettings` objects, this can be done by inheriting and creating a custom `LDAPSettings` class. For example:

```
@dataclass()
class MyLDAPSettings(LDAPSettings):
    USE_SSL: bool = False


WAUTH_DOMAINS = {
    "EXAMPLE": MyLDAPSettings(
        SERVER="example.local",
        SEARCH_BASE="DC=example,DC=local",
```

(continues on next page)

(continued from previous page)

```
        USERNAME="EXAMPLE\\bind_account",
        PASSWORD="*********",
    ),
}
```

1.15.3 Extending LDAP Settings

Sometimes it is useful to have some **extra LDAP Settings** for use with the LDAP Manager.

It is possible to create a custom `LDAPSettings` class and use it to configure the LDAP Settings for domains. Those extra setting will be available in the **settings attribute** of `LDAPManager` objects, and can be used **throughout your code**. Those settings should not affect the existing settings used by `django-windowsauth` for User synchronization or any other uses.

Custom LDAP Settings objects can be created by inheriting from the `LDAPSettings` dataclass, like so:

```
@dataclass()
class MyLDAPSettings(LDAPSettings):
    EXTRA_SETTING: str = "Hello, world!"


WAUTH_DOMAINS = {
    "EXAMPLE": MyLDAPSettings(
        SERVER="example.local",
        SEARCH_BASE="DC=example,DC=local",
        USERNAME="EXAMPLE\\bind_account",
        PASSWORD="*********",
    ),
}
```

Then the setting could be accessed from `LDAPManager` object:

```
>>> from windows_auth.ldap import get_ldap_manager
>>> manager = get_ldap_manager("EXAMPLE")
>>> manager.settings.EXTRA_SETTING
"Hello, world!"
```

1.15.4 Base Settings

SERVER

Type `bool`; **Required**.
FQDN, IP, or URL of the LDAP Server.

The Fully Qualified Domain Name, IP Address or complete URL in the scheme `scheme://hostname:hostport` of the LDAP Server. This setting will be used as `host` property for ldap3's Server object.

When using Active Directory, this address should direct to a DC Server (Domain Controller) for the domain. By default, the FQDN of the domain itself will be resolved into your current configured DC Server. That way, in case you have multiple DC servers in your domain, you will be dynamically changing the server you are accessing.

See also:

From the Microsoft Docs https://docs.microsoft.com/en-us/windows-server/identity/ad-ds/plan/domain-controller-location

USERNAME

Type `str`; **Required**.
The account to be used when binding to the LDAP Server.

The username in one of the Credentials Manager API's formats (Down-Level or SPN) of a user with the permissions needed for your application. By default, **read-only permissions** for the user accounts that are able to authenticate via IIS Windows Authentication to **your website** is needed.

If you are planing to use **NTLM authentication** to your LDAP Server, the username must be in the Down-Level Logon Name format (DOMAIN\username).

In production, it is advised to use a **dedicated Service Account** to authorize your application in your Active Directory domain.

PASSWORD

Type `str`; **Required**.
Password of the user to be used when binding to the LDAP Server.

The password for the user used to authenticate to the LDAP Server.

> **Warning:** It is highly advised not to store sensitive secrets like password in your code. You should use a safe and secure place to store the password. See the tutorial manage_secrets

SEARCH_BASE

Type `str`; **Required**.
The DN of the container used as starting point for LDAP searches.

When querying LDAP Directories, it is required to specify the **root container** to start the search from. Then, depending on the search scope, the objects are searched **directly or indirectly** in respect to the search base container.

For searches throughout all the domain's containers, the search base DN is usually in the format `DC=<domain name>,DC=<parent domain>`.

See also:

Microsoft docs about search bases https://docs.microsoft.com/en-us/windows/win32/ad/binding-to-a-search-start-point

USE_SSL

Type `bool`; Default to `True`; Not Required.
Connect to LDAP over secure port, usually 636.

This setting is used as the `use_ssl` parameter for the ldap3 `Server` object.

See also:

ldap3 Server object docs https://ldap3.readthedocs.io/en/latest/server.html

READ_ONLY

Type `bool`; Default to `True`; Not Required.
Prevent modify, delete, add and modifyDn (move) operations.

Connect to the LDAP Server with a read only protection. This can farther minimize risks and vulnerabilities from unwanted operations against the LDAP Server.

This setting is used as the `read_only` parameter for the ldap3 `Connection` object.

> **Warning:** This is not guaranteed to be a risk / vulnerabilities free connection, **make sure to minimize the bind account's permissions**

COLLECT_METRICS

Type `bool`; Default to `True`; Not Required.
Collecting connection usage metrics.

Enabling `ldap3`'s Connection Metrics collection. Those usage metrics can be accessed via `manager.get_usage()` method.

Connection metrics can be saved automatically, see how to *Collect Metrics*

See also:

`ldap3` documentation https://ldap3.readthedocs.io/en/latest/metrics.html

SERVER_OPTIONS

Type `dict`; Default to `{}`; Not Required.
Extra parameters for the ldap3 `Server` object.

A dictionary of extra keyword arguments to pass when creating the ldap3 `Server` object.

See also:

For more information, see ldap3 docs https://ldap3.readthedocs.io/en/latest/server.html

CONNECTION_OPTIONS

Type `dict`; Default to `{}`; Not Required.
Extra parameters for the ldap3 `Connection` object.

A dictionary of extra keyword arguments to pass when creating the ldap3 `Connection` object.

See also:

For more information, see ldap3 docs https://ldap3.readthedocs.io/en/latest/connection.html

PRELOAD_DEFINITIONS

Type `tuple`; Default is shown below; Not Required.
Preload LDAP schema for defining LDAP objects in Python.

A list of LDAP Object definitions to **preload** while connecting to the LDAP Server. This **caches** ldap3 `ObjectDef` objects on the `LDAPManager` object for each defined object class. The object definitions are later get used for **parsing the objects** received from querying the LDAP Directory. Preloading the object definitions can **minimize the extra delay** for first query for an object.

The definitions can be listed as a **simple string** referring to an LDAP object class, or a **2 valued tuple** with the LDAP object class string on the first value, and a list of **extra attributes** on the second value. For example:

```
{
    "PRELOAD_DEFINITIONS": (
        ("user", ["sAMAccountName"]),
        "group"
    ),
}
```

The configuration above is the actual default configuration for this setting.

USER_FIELD_MAP

Type `dict`; Default is shown below; Not Required.
Translate User Model fields to LDAP User object attributes.

Provide a mapping for your **Django User Model fields** to the **LDAP User object attributes**. Those mappings are used when synchronizing Django Users to their related LDAP Users.

In case you using a **Custom User Model** in your Django project, you also will be able to map them to LDAP Attributes. This is mentioned in the tutorial *Using Custom User Model Mappings*.

Note: Make sure to specify the needed attributes when **preloading definitions** for non-default attributes.

```
{
    "USER_FIELD_MAP": {
        "username": "sAMAccountName",
        "first_name": "givenName",
```

(continues on next page)

(continued from previous page)

```
        "last_name": "sn",
        "email": "mail",
    }
}
```

The configuration above is the actual default configuration for this setting.

USER_QUERY_FIELD

Type `str`; Default to `username`; Not Required.
The User Model field used for searching the related LDAP User object.

When synchronizing users to LDAP, they are first need to be searched. This setting can allow you to specify the **Django User Model field** that will be compared to the related **LDAP Attribute** using the `USER_FIELD_MAP` setting when searching for the related user.

This setting may be useful when using a **Custom User Model** in your Django project. This is mentioned in the tutorial *Using Custom User Model Mappings*.

Note: Make sure to use a unique field, that is unique at the **LDAP side** too. If multiple objects are found, the synchronization will fail.

GROUP_ATTRS

Type `str` or `tuple`; Default to `cn`; Not Required.
The LDAP group attributes to search when matching to Django groups.

When synchronizing users against LDAP, you can **replicate group memberships**. When used, you may want to specify what **LDAP attributes** are used when comparing the **Django Group's names** to LDAP Groups.

This setting can be a **single string** for comparing a single attribute, or a **tuple** for comparing multiple attributes. When comparing multiple attributes, if one of them matches the Django Group's name, the user is added to that group.

Warning: The comparing is done on the **Python side** by the ldap3 library. Using many attributes to search groups may result in **longer synchronization times**.

SUPERUSER_GROUPS

Type `tuple` or `str`; Default to `Domain Admins`; Not Required.
LDAP Groups to check membership for setting Django User's "is_superuser" flag.

When synchronizing users against LDAP, you can specify a **list of LDAP Groups** to match for setting the Django User's `is_superuser` flag. If the user is member in **one** of the listed LDAP groups, the `is_superuser` flag will be set to `True`, otherwise it is set to `False`.

Configuring this setting to `None` will not modify the `is_superuser` flag.
Configuring this setting to a **string** is equal to a **single length tuple**.

The group membership is checked by comparing the **groups listed in this setting** to the **LDAP Group Attributes** listed in `GROUP_ATTRS` setting.

STAFF_GROUPS

Type `tuple` or `str`; Default to `Administrators`; Not Required.
LDAP Groups to check membership for setting Django User's "is_staff" flag.

When synchronizing users against LDAP, you can specify a **list of LDAP Groups** to match for setting the Django User's `is_staff` flag. If the user is member in **one** of the listed LDAP groups, the `is_staff` flag will be set to `True`, otherwise it is set to `False`.

Configuring this setting to `None` will not modify the `is_staff` flag.
Configuring this setting to a **string** is equal to a **single length tuple**.

The group membership is checked by comparing the **groups listed in this setting** to the **LDAP Group Attributes** listed in `GROUP_ATTRS` setting.

ACTIVE_GROUPS

Type `tuple` or `str`; Default to `None`; Not Required.
LDAP Groups to check membership for setting Django User's "is_active" flag.

When synchronizing users against LDAP, you can specify a **list of LDAP Groups** to match for setting the Django User's `is_active` flag. If the user is member in **one** of the listed LDAP groups, the `is_active` flag will be set to `True`, otherwise it is set to `False`.

Configuring this setting to `None` will not modify the `is_active` flag.
Configuring this setting to a **string** is equal to a **single length tuple**.

The group membership is checked by comparing the **groups listed in this setting** to the **LDAP Group Attributes** listed in `GROUP_ATTRS` setting.

PROPAGATE_GROUPS

Type `bool`; Default to `True`; Not Required.
Propagate groups in order Superusers > Staff > Active.

When set to `True`, all groups configured in `SUPERUSER_GROUPS` will be added to `STAFF_GROUPS` and `ACTIVE_GROUPS`, and groups configured in `STAFF_GROUPS` with be added to `ACTIVE_GROUPS`.

GROUP_MAP

Type `dict`; Default is `{}`; Not Required.
Map one or more LDAP Groups membership to Django Group membership.

When synchronizing users against LDAP, you can specify a mapping of LDAP Groups to Django Groups. If the user is member in **one** of the listed LDAP groups, it will be added to the respective Django Group.

The setting is configured as a dictionary where the **keys are Django Groups names** and the value is **a string or a list of LDAP Groups**. The LDAP Groups are compared using the attributes listed in the `GROUP_ATTRS` setting.

When a user synchronizes, LDAP group membership will be checked **directly or in-directly** for at least one of the listed groups. For each LDAP group that the user is found to be a member of, it will be **added** to the related Django group, otherwise it will be **removed** from it. Groups that are **not listed** in this setting will **not be affected** by this.

> **Warning:** When a group that is configured in this setting is missing, it will be **created automatically**.

FLAG_MAP

Type `dict`; Default is `{}`; Not Required.
Map User object boolean fields to one or more LDAP Groups membership check.

Used to synchronize boolean fields from the User object as a group membership check. If the user is member in **one** of the listed LDAP groups, the respective boolean field will be set to `True`, otherwise it is set to `False`.

Configuring this setting to `None` will not modify the field.
Configuring this setting to a **string** is equal to a **single length tuple**.

The group membership is checked by comparing the **groups listed in this setting** to the **LDAP Group Attributes** listed in `GROUP_ATTRS` setting.

User's `is_superuser`, `is_staff` and `is_active` are added automatically from settings `SUPERUSER_GROUPS`, `STAFF_GROUPS` and `ACTIVE_GROUPS` respectively.

1.16 View Decorators

Provided with this module are some useful decorators to use on your views. Those decorators can be used on function based view normally:

```
@domain_required
def my_view(request):
    # ...
```

And for class-based view with the `@method_decorator` decorator

```
@method_decorator(domain_required, name='dispatch')
class MyView(TemplateView):
    # ...
```

See also:

https://docs.djangoproject.com/en/3.1/topics/class-based-views/intro/#decorating-class-based-views

1.16.1 domain_required

Require that the logged on user has LDAP relation with a domain. In case it is not, redirect to login page.

Parameters

- **domain**: Check if is member in a specific domain (default: None)
- **login_url**: Login page URL (default: None)
- **bypass_superuser**: Allow superusers to bypass this requirement (default: True)

1.16.2 ldap_sync_required

Require the logged on user to be synced against LDAP. This can be used to override the global `WAUTH_REQUIRE_RESYNC` and `WAUTH_RESYNC_DELTA` settings.

Parameters

- **timedelta**: Maximum acceptable time since the last synchronization (default: None)
- **login_url**: Login page URL (default: None)
- **allow_non_ldap**: Allow non-LDAP users to access (default: True)
- **raise_exception**: When sync fails, raise the exception and cause response status code 500 (default: False

> **Warning:** When configuring `WAUTH_USE_CACHE` to True, this decorator will re-sync the user in regards to the `timedelta` parameter

1.17 Signals

1.17.1 ldap_user_sync

Whenever a user is synced against LDAP, pre-saving.

Arguments:

- **sender** The LDAPUser instance that is being synced.
- **ldap_user** The `ldap3` Entry instance received from LDAP server of the user being synced
- **group_reader** Reader cursor for all the user's groups, already queried.

Example:

```
from django.dispatch import receiver
from ldap3 import Entry, Reader

from windows_auth.models import LDAPUser
from windows_auth.signals import ldap_user_sync


@receiver(ldap_user_sync)
def on_ldap_sync(sender: LDAPUser, ldap_user: Entry = None, group_reader: Reader =␣
↪None):
    # do something...
    pass
```

Warning: Any unhandled exception raised during the signal will terminate the sync process.

1.18 Models

1.18.1 LDAPUser

Used to store user domain information and perform domain related actions.

Fields:

- **user** - One to one relation for user model using `get_user_model` function.
- **domain** - User's domain name (usually) as NetBIOS Name.

Methods:

- **get_ldap_manager()** - Get `LDAPManager` for user's domain.
- **get_ldap_attr(attribute, as_list)** - Get LDAP attribute of the related LDAP user.
- **get_ldap_user()** - Get related LDAP user as ldap3 `Entry` object.
- **get_ldap_groups()** - get LDAP Reader for all groups the user is a member of.
- **sync()** - Synchronize Django user to related LDAP User.

The `LDAPUser` for a Django User can be accessed via `user.ldap`. For example, you can trigger sync with `request.user.ldap.sync()`, or display the user's Windows Logon Name with `request.user.ldap`.

Note: The `LDAPUser` is represented by the **Down-level Logon Name** or **SPN** determined by the `WAUTH_USE_SPN` setting. More on that in the *Settings*.

1.18.2 LDAPUserManager

LDAPUser Accessible via `LDAPUser.objects`.

Methods:

- **create_user()** - Create a new user from LDAP.

1.19 Management Commands

1.19.1 createwebconfig

Generate `web.config` files with configurations for your Django Project's IIS website

Arguments

- **–name**, **-n** FastCGI Handler Name (default: Django FastCGI).
- **–static**, **-s** Create a `web.config` to configure IIS to serve the static folder.
- **–media**, **-m** Create a `web.config` to configure IIS to serve the media folder.
- **–windowsauth**, **-w** Configure Windows Authentication as the only IIS Authentication option.
- **–https** Configure HTTP to HTTPS Redirect using IIS's URL Rewrite module.
- **–logs**, **-l** Path for the WFastCGI logs.
- **–override**, **-f** Force override existing files.

Note: Before using the **–static** or **–media** flags, make sure to configure correctly the `STATIC_ROOT` and `MEDIA_ROOT` settings.

Warning: In order for the `web.config` files to work correctly, you will need to **unlock** some IIS Configuration Section. See the **Install and Setup IIS** section at *Installation and Setup* docs.

1.19.2 createtask

Add a management command to Windows Task Scheduler.

Arguments

- **command** Management command, wrapped with "command".
- **–predefined**, **-p** Create from a predefined task.
- **–name**, **-n** Task name.
- **–desc**, **-d** Task description.

- **–identity**, **-u** Task principal identity (default: “NT Authority\LocalSystem”).
- **–folder**, **-f** Task folder location (default: Project’s name).
- **–interval**, **-i** Task execution interval as timedelta kwargs, e.g. “days=1,hours=12.5”.
- **–random**, **-r** Randomize execution time as timedelta kwargs, e.g. “days=1,hours=12.5”.
- **–timeout**, **-t** Execution time limit as timedelta kwargs, e.g. “days=1,hours=12.5” (default: 1 hour).
- **–priority** Task priority https://docs.microsoft.com/en-us/windows/win32/taskschd/tasksettings-priority

Predefined tasks

- **clearsessions** Clear sessions from database every week.
- **clean_duplicate_history** Clean duplicate history records from all models with history every 3 hours (from django-simple-history).
- **clean_old_history** Clean history records older then 30 days from all models with history every day (from django-simple-history).
- **process_tasks** Worker for background tasks processing (from django-background-tasks).

1.20 Change Log

1.20.1 1.4.0

Release date: Feb. 20, 2021

- **ADDED**: LDAPUserManager for manually creating users from LDAP.
- **ADDED**: `createtask` management command for creating Task Scheduler jobs.
- **ADDED**: `ldap_user_sync` signal.
- **IMPROVED**: LDAP Settings for Group Membership check propagate to one another.
- **MODIFIED**: Increased the default `WAUTH_RESYNC_DELTA` to every 1 day.

1.20.2 1.3.2

Release date: Jan 26, 2021

- **FIXED**: `ProgrammingError` Exception before first migration
- **FIXED**: Packaging configuration missing templates

1.20.3 1.3.1

Release date: Jan 15, 2021

- **MODIFIED**: Remove requirement for `WAUTH_DOMAIN` setting
- **FIXED**: `OperationalError` Exception before first migration
- **FIXED**: Incorrect packaging configuration

1.20.4 1.3.0

Release date: Jan 10, 2021

- **ADDED**: LDAP Metrics collection to Database
- **ADDED**: LDAP Panel for `django-debug-toolbar`
- **ADDED**: LDAP Setting `COLLECT_METRICS`
- **ADDED**: Auto-close all LDAP connection on before process exit
- **ADDED**: View decorators `domain_required` and `ldap_sync_required`
- **ADDED**: `--https` parameter for `createwebconfig` for HTTPS Redirection

1.20.5 1.2.0

Release date: Dec 19, 2020

- **ADDED**: Setting `WAUTH_ERROR_RESPONSE` for custom sync error responses
- **ADDED**: Moved automatic sync login from Authentication Backend `WindowsAuthBackend` to a new Middleware `UserSyncMiddleware`.

1.20.6 1.1

Release date: Dec 17, 2020

- First published version