UNIVERSITY OF CRETE
Computer Science Department

Single Sign-on for Internet Services

Thesis submitted in partial fulfilment of the requirements for the degree of Computer Science

LOUKAS MERKOURIS, University of Crete, Greece

The Internet hosts various amount of web applications, services, and even more users. These users, while using an application, often want to access their data that resides in another, third-party application, i.e., when a user uses an email application and wants to access his contacts that exist in another application, how will the email application eventually get his contacts? In the old-fashioned way, the user had to hand over the credentials of his third-party application account to the application he eventually wants to get his data. This was insecure for the user, the user gave access to all of his data in the 3rd party application, including his password. With that method, the application that took his credentials could store them in plain text, the database could be hacked, and to revoke access, there was only one option; to change his password. With Single Sign-on (SSO), this problem no longer exists, as users can authorize the application to access a specific set of data residing elsewhere without disclosing passwords. Today there are many protocols and frameworks to achieve SSO, for users to be able to authorize and authenticate using third-party applications. In this thesis, we examine SSO between Internet services using the OAuth 2.0 [6] and OpenID Connect [7] protocols, by implementing example web-based applications in Python using the Django [4] framework. We also extend one of the example applications to use a remote LDAP [5] server as a user directory instead of a local user database.

1 INTRODUCTION

Single Sign-on (SSO), is an authentication and authorization scheme where users can securely authorize, authenticate and gain access to multiple applications and websites by only logging in with a single username and password. In this project we use the OAuth 2.0 (OAuth) and OpenID Connect (OIDC) technologies to enable SSO. With SSO we can authorize and authenticate applications and access resources in a secure and fast way. Users can be in a local or a remote database. To enable authentication with users in a remote database, we use authentication via the Lightweight Directory Access Protocol (LDAP), which is commonly deployed for accessing and maintaining data in a company/organization server.

2 DESIGN

To test SSO with OAuth and OIDC, we implemented 3 applications with Django, which is a high-level Python web framework that encourages rapid development and clean, pragmatic design. Django follows the model–template–views (MTV) architectural pattern. Django's primary goal is to ease the creation of complex, database-driven websites. The framework emphasizes reusability and "pluggability" of components, less code, low coupling, rapid development, and the principle of

This thesis was supervised by Prof. Angelos Bilas, co-supervised by Associate Prof. Polyvios Pratikakis and Antony Chazapis, PhD, Institute of Computer Science, FORTH, and submitted in December 2021.

don't repeat yourself. Python is used throughout, even for settings, files, and data models. Django also provides an optional administrative create, read, update and delete interface that is generated dynamically through introspection and configured via admin models.

2.1 OAuth 2.0

OAuth is an open-standard authorization protocol that provides applications the ability for "secure designated access". For example, you can tell Facebook that it's ok for YouTube to access your profile or post updates to your timeline without having to give YouTube your Facebook password. This minimizes risk in a major way: in the event, YouTube suffers a breach, your Facebook password remains safe. Hence, OAuth solves the problem of sharing resources, as users don't have to disclose passwords or create accounts to use a service. It's easy for end-users to use, and the access can be revoked anytime.

The glossary of OAuth includes the following components:

- **Resource Owner:** The user or system that owns the protected resources and can grant access to them.
- **Access Token:** A piece of data that represents the authorization to access resources on behalf of the Resource Owner.
- **Client:** A system that requires access to the protected resources. To access resources, the Client must hold the appropriate Access Token.
- **Authorization Server:** This server receives requests from the Client for Access Tokens and issues them upon successful authentication and consent by the Resource Owner. The authorization server exposes two endpoints: the Authorization endpoint, which handles the interactive authentication and consent of the user, and the Token endpoint, which is involved in a machine to machine interaction.
- **Resource Server:** A server that protects the user's resources and receives access requests from the Client. It accepts and validates an Access Token from the Client and returns the appropriate resources to it.
- **Scope:** A mechanism to limit an application's access to a user's account.

Scopes are used to specify exactly the reason for which access to resources may be granted. An application can request one or more scopes. This information is presented to the user in the consent screen, and the Access Token issued to the application will be limited to the scopes granted. An Access Token is something like a key, while scopes specify which doors the key can open. Acceptable scope values and which resources they relate to are dependent on the Resource Server.

To enable the OAuth protocol in our project we created the **Provider**, acting both as the Resource Server and the Authorization Server, (the Resource and Authorization Servers can be separate applications, for this project they are the same), and the **Client**. For implementing the Provider we used the **Django OAuth Toolkit** [2] package, which provides all the endpoints, data, and logic needed to add OAuth capabilities to our Django project. With the toolkit, we made our Authorization Server issue Access Tokens to Client applications for a certain API.

Before our Application (the Client), can use the Authorization Server for authorization or authentication, we must first register the app. Once registered, the Client will be granted access to the API, subject to approval by its users. The registration process will automatically generate a unique Client_id and Client_secret, which are used so the Client and the Authorization Server establish a working relationship. The Authorization Server generates the Client ID and Client Secret (sometimes called the App ID and App Secret) and gives them to the Client to use for all future OAuth exchanges. The Client Secret must be kept secret so that only the Client and Authorization

Server know what it is. This is how the Authorization Server can verify the Client. We have to provide the rest of the information:

- **User:** The owner of the Application (e.g., a developer, or the currently logged in user).
- **Redirect URIs:** Applications must register at least one redirection endpoint before using the authorization endpoint. The Authorization Server will deliver the Access Token to the Client only if the Client specifies one of the verified redirection URIs.
- **Client type:** This value affects the security level at which some communications between the Client application and the Authorization Server are performed. For this project we use Confidential.
- **Authorization grant type:** Authorization Code.
- **Name:** This is the name of the Client application on the server, and will be displayed on the authorization request page, where users can allow/deny access to their data.

The authorization grant type refers to the way an application (the Client) gets an Access Token. OAuth defines several grant types, while extensions can also define new grant types. Each grant type is optimized for a particular use case, whether that's a web app, a native app, a device without the ability to launch a web browser, or server-to-server applications. We used the **Authorization Code** grant type, because when authorizing an application to access OAuth-protected data with this grant type, the flow is always initiated by the user and the application can prompt users to click a special link to start the process. When a user clicks the link, the Client redirects the browser to the Authorization Server. If the user is not logged in, he will be prompted for a username and password. This is because the authorization page is login-protected by Django-OAuth-Toolkit. After login the user should see be presented with a form, including the scopes, to give his authorization to the Client application. User will flag the checkbox, click Authorize, and will be redirected again to the Client.

At this point the Authorization Server redirects the user to a special page on the Client passing in an Authorization Code and a special token that the Client will use to obtain the final Access Token. This operation is usually done automatically by the Client application during the request/response cycle. Access Tokens have a specific lifetime, given by the Authorization Server. Access Token has a Refresh Token as well which is used for a brand new Access Token when necessary, without repeating the authorization process, as it has no expire time.

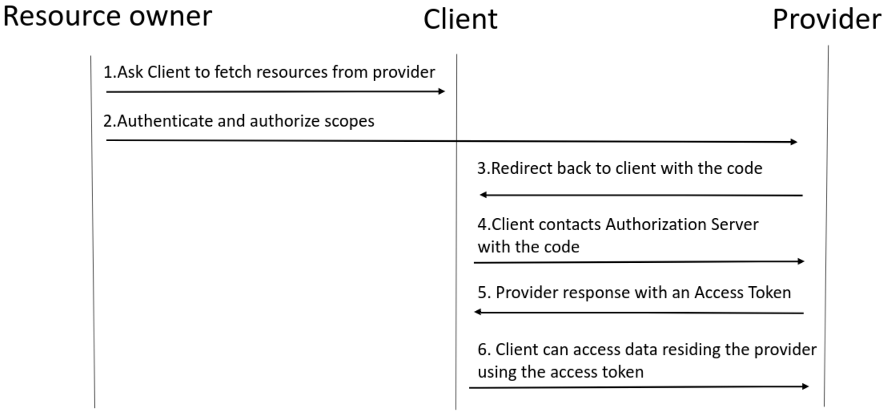

Resource owner
Client
Provider
1.Ask Client to fetch resources from provider
2.Authenticate and authorize scopes
3.Redirect back to client with the code
4.Client contacts Authorization Server with the code
5. Provider response with an Access Token
6. Client can access data residing the provider using the access token

Fig. 1. OAuth flow

The OAuth flow, shown in Fig. 1, goes as follows:

(1) The **Resource Owner**, wants to allow the **Client** to access data residing in the **Provider**.
(2) The **Resource Owner** clicks a special link to start the flow and the **Client** redirects the browser to the **Authorization Server** including with the request the **Client ID**, **Redirect URI**, **Response Type**, and one or more **Scopes** it needs. The **Authorization Server** verifies the user, and if necessary prompts for a login, then presents the user with a consent form based on the scopes requested by the **Client**. The **Resource Owner** grants (or denies) permission.
(3) The **Authorization Server** redirects back to the **Client** using the **Redirect URI** along with an **Authorization Code**.
(4) The **Client** contacts the **Authorization Server** directly (does not use the Resource Owner's browser) and securely sends its **Client ID**, **Client Secret**, and the **Authorization Code**.
(5) The **Authorization Server** verifies the data and responds with an **Access Token**.
(6) The **Client** can now use the **Access Token** to send requests to the **Resource Server** for accessing data.

Note that OAuth is used for generic authorization; for authentication, we need to create an API endpoint at the Provider and call it in order to learn information about the user (ID, username, etc.).

2.2 OpenID Connect

OAuth is designed only for authorization, for granting access to data and features from one application to another. After the OAuth flow, usually the first step for the Client is to get information about the Resource Owner (the end-user), by sending the proper request to the Resource Server. That step can be avoided by using OIDC. OpenID Connect (OIDC) is a thin layer that sits on top of OAuth that adds login and profile information about the person who is logged in. Establishing a login session is often referred to as authentication, and information about the person logged in (i.e. the Resource Owner) is called identity. When an Authorization Server supports OIDC, it is sometimes called an identity provider, since it provides information about the Resource Owner back to the Client. OIDC flow looks the same as OAuth, the only differences are that in the initial request, a specific scope of **openid** is used, and the Server responds with an Access Token and A JSON Web Token (JWT) that contains the information about the Resource owner.

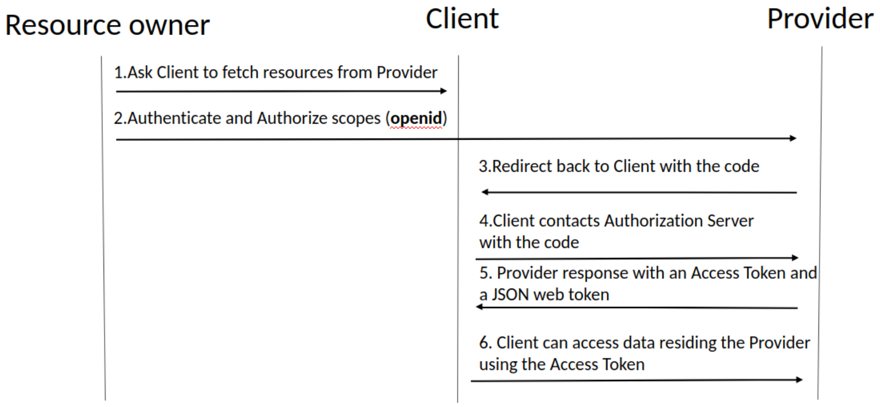

Resource owner
Client
Provider
1.Ask Client to fetch resources from Provider
2.Authenticate and Authorize scopes (openid)
3.Redirect back to Client with the code
4.Client contacts Authorization Server with the code
5. Provider response with an Access Token and a JSON web token
6. Client can access data residing the Provider using the Access Token

Fig. 2. OIDC flow

The OIDC flow, shown in Fig. 2, is the same with the OAuth flow with two extra steps:

- In step 2, the **Authorization Server** presents the user with a consent form based on the scopes and a specific scope of **openid** is requested by the **Client**.
- In step 5, the **Authorization Server** verifies the data and responds with both an **Access Token** and a **JSON Web Token**.

Finally, the Client uses the Access Token to access data that resides in the Resource Server on behalf of the Resource Owner; using the JWT the Client has all authentication information about the Resource Owner.

2.3 Lightweight Directory Access Protocol

The Lightweight Directory Access Protocol (LDAP) is one of the core authentication protocols that was developed for directory services. LDAP historically has been used as a database of information, primarily storing information like **users**, **attributes about those users**, **group membership privileges** and more. This information was then used to enable authentication to IT resources such as an application or server. They would be pointed to the LDAP database, which would then validate whether that user would have access to it or not. That validation would be done by passing a user's credentials. LDAP authentication follows the client/server model. In this scenario, the client is generally an LDAP-ready system or application that is requesting information from an associated LDAP database and the server is, of course, the LDAP server. The server side of LDAP is a database that has a flexible schema. In other words, not only can LDAP store username and password information, but it can also store a variety of attributes including address, telephone number, group associations, and more. As a result, a common LDAP use case is to store core user identities. In doing so, IT can point LDAP-enabled systems and applications (for example) to an associated LDAP directory database, which acts as the source of truth for authenticating user access.

We add LDAP authentication in our OIDC Provider using the **auth-ldap** [1] package. This package is a Django authentication backend that authenticates against an LDAP service (in our case FORTH'S LDAP Server). Configuration can be as simple as a single **distinguished name** (DN) template, but there are many rich configuration options for working with users, groups, and permissions. For authentication with the LDAP server, we need to add the LDAP backend from the auth-ldap package and point to the LDAP server.

Now that we can talk to the LDAP server, the next step is to authenticate a username and password. There are two ways to do this, called **search/bind** and **direct bind**. The first one involves connecting to the LDAP server either anonymously or with a fixed account, and searching for the distinguished name of the authenticating user. Then we can attempt to bind again with the user's password. The second method is to derive the user's DN from his username and attempt to bind as the user directly.

We used Direct Bind, to skip the search phase. All we need to do was to set `AUTH_LDAP_USER_DN_TEMPLATE` to a template that will produce the authenticating user's DN directly. This template should have one placeholder, %(user)s. By setting `AUTH_LDAP_USER_ATTR_MAP` we are able to match our user's model attributes with the LDAP user. With these settings, we are able to connect to an LDAP server. When the LDAP server authenticates the user's credentials, the auth-ldap backend will return an ldap_user and will create a Django user on return with the attributes that were given.

The flow of authentication when using an LDAP database with OIDC (Fig. 3), is the same as the OIDC flow, the only difference being that the Provider searches for users in a remote database:

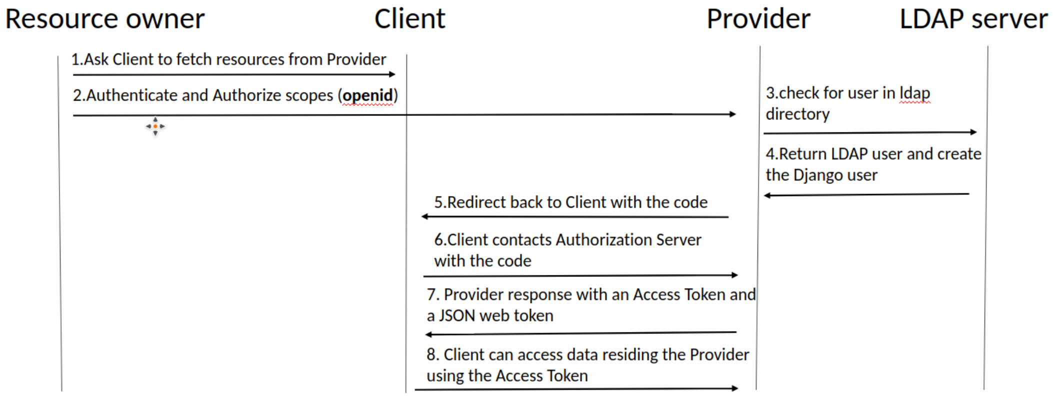

Resource owner
Client
Provider
LDAP server
1.Ask Client to fetch resources from Provider
2.Authenticate and Authorize scopes (openid)
3.check for user in ldap directory
4.Return LDAP user and create the Django user
5.Redirect back to Client with the code
6.Client contacts Authorization Server with the code
7. Provider response with an Access Token and a JSON web token
8. Client can access data residing the Provider using the Access Token

Fig. 3. OIDC with LDAP flow

- In step 3, the **Authorization Server** validates given credentials with the remote LDAP database.
- In step 4, if the remote user exists, a local Django user is created. The **Authorization Server** presents the user with a consent form based on the scopes (like OIDC, a specific scope **openid** is used) requested by the **Client**. User grants (or denies) permission.

The Resource Owner, is authenticated by the Client using his credentials that exist in a remote database and the Client has all the profile information of the LDAP user, as the data is copied into the OIDC JWT.

3 IMPLEMENTATION

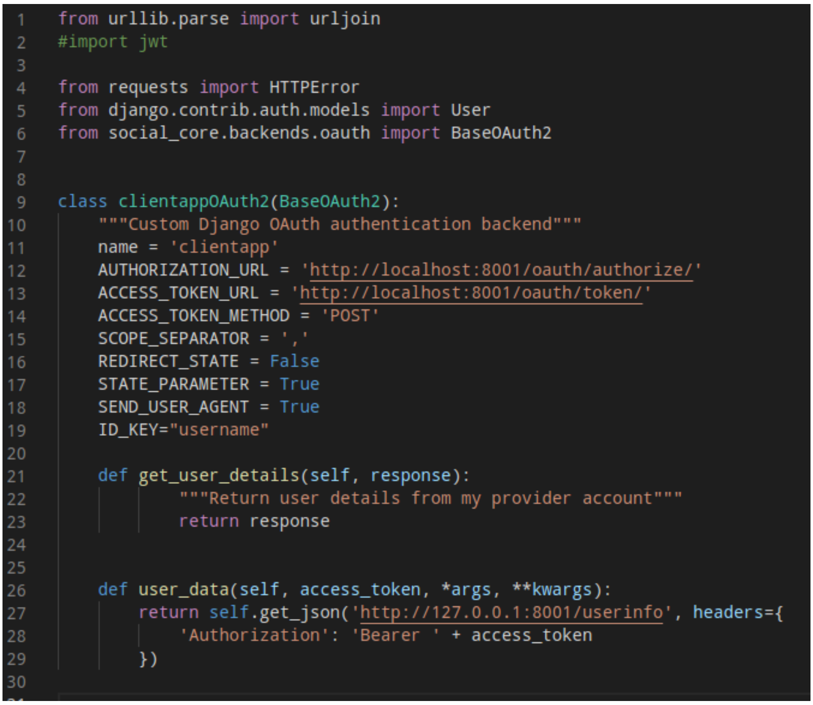
In this section, we will take a closer look at how the implementation works (Fig. 6). Starting from the Client and following to the Providers, we show how we add the components in each application, what extra code we use to enable different authentication and authorization schemes.

3.1 The Client

The client is a Django application with a custom user model. Django comes with a user authentication system. It handles user accounts, groups, and permissions, using cookie-based user sessions. For authenticating, Django searches for the given user's credentials in its backends, and if the user exists, Django authenticates and redirects the user to the proper URL. We implement 2 custom backends for the Client application, one for searching users in the OAuth Provider (Fig. 4) and one for OIDC Provider (Fig. 5).

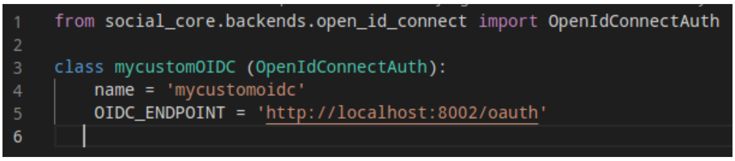
We installed the Python **django-social-auth** [3] module, which is a package for authenticating against third-party applications, like GitHub, Google, Facebook, and other websites. It also contains the base class for the OAuth authentication backend, called BaseOAuth2. In our custom OAuth backend, we override some of the BaseOAuth2 components, so the backend knows where to search for the Authorization Server, where to get the token, and other details.

For the OIDC custom backend, all we need to do is to overwrite the OpenIdConnectAuth class, defined by django-social-auth, provide the name of our backend and the URL of the Authorization Server. The Client is ready to redirect the Resource User to the Providers and initiate the OAuth/OIDC process.



```python
from urllib.parse import urljoin
#import jwt

from requests import HTTPError
from django.contrib.auth.models import User
from social_core.backends.oauth import BaseOAuth2


class clientappOAuth2(BaseOAuth2):
    """Custom Django OAuth authentication backend"""
    name = 'clientapp'
    AUTHORIZATION_URL = 'http://localhost:8001/oauth/authorize/'
    ACCESS_TOKEN_URL = 'http://localhost:8001/oauth/token/'
    ACCESS_TOKEN_METHOD = 'POST'
    SCOPE_SEPARATOR = ','
    REDIRECT_STATE = False
    STATE_PARAMETER = True
    SEND_USER_AGENT = True
    ID_KEY="username"

    def get_user_details(self, response):
            """Return user details from my provider account"""
            return response


    def user_data(self, access_token, *args, **kwargs):
        return self.get_json('http://127.0.0.1:8001/userinfo', headers={
            'Authorization': 'Bearer ' + access_token
        })

```

Fig. 4. OAuth custom backend



```python
from social_core.backends.open_id_connect import OpenIdConnectAuth

class mycustomOIDC (OpenIdConnectAuth):
    name = 'mycustomoidc'
    OIDC_ENDPOINT = 'http://localhost:8002/oauth'
    
```

Fig. 5. OIDC custom backend

3.2 The Providers

As mentioned, we used the Django-OAuth-Toolkit for the implementation of the Providers. To enable the Django-OAuth-Toolkit we need to add "oauth2_provider" in installed apps and add the middleware "oauth2_provider.middleware.OAuth2TokenMiddleware" in settings.py. The next step is to add the toolkit's backend for OAuth requests by adding: "oauth2_provider.backends.OAuth2Backend" in authentication backends, and specifying the scopes as:
'SCOPES': {'read': 'Read scope', 'write': 'Write scope'}. In urls.py we add an extra URL: path('userinfo', user_info_secret), which is the endpoint in order for the custom backends to know where to search for the user's information after authenticating. We also provide the OAuth Toolkit's endpoints by adding url(r"^oauth/", include ('oauth2_provider.urls', namespace='oauth2_provider')).

For the OIDC Provider implementation, we need to do two more steps. Add the specific scope of **openid** in 'SCOPES', and create a CustomOAuth2Validator method, which is used for mapping the user's info from the OIDC Provider to the Client.

Finally, we have to register the Client in both of the Providers and set the correct Client_id and Client_secret in Client's settings.py. For the OIDC implementation, we need to provide an algorithm, which is used for signing the JWT that the Provider sends back to the Client, for which we used the RS256 algorithm. After the registration process, Providers are ready to give Access Tokens to the Client.

In the OIDC implementation, we also add the LDAP authentication scheme, by setting:

- `AUTH_LDAP_SERVER_URL`: Points to the LDAP server.
- `AUTH_LDAP_BIND_DN`: Username.
- `AUTH_LDAP_BIND_PASSWORD`: Password.
- `AUTH_LDAP_USER_DN_TEMPLATE`: How to find the user.
- `AUTH_LDAP_USER_ATTR_MAP`: What attributes we want to populate the Django user who is going to be created.

The whole implementation is shown in Fig. 6. The Resource Owner wants to use the Client which is a Django application with two custom backends and can authenticate the user via the OAuth or OIDC implementation of the Provider. The OAuth and OIDC Providers are Django apps and use the Django-oauth-toolkit.

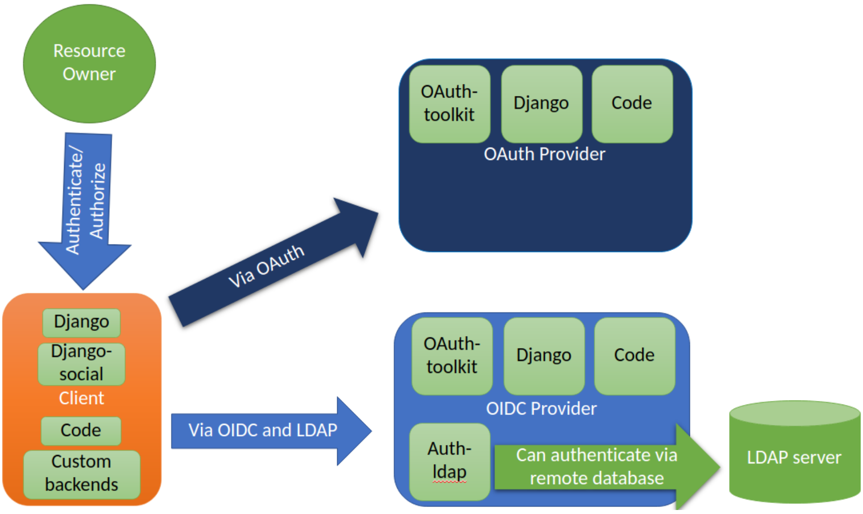


Fig. 6. Implementation

4 CONCLUSION

We managed to enable both OAuth and OIDC protocols to authorize and authenticate between Django applications. We built two Provider applications with Django using the Django-OAuth-Toolkit, acting as both Resource Servers and Authentication Servers, one for OAuth and one for OIDC. With Django we also created the Client, an application for users to login via the Providers, by creating 2 custom backends (one for OAuth and one for OIDC) which override the base OAuth class provided by django-social-auth, which allowed us to authenticate and authorize from the Client application. With the auth-ldap package, we added LDAP authentication functionality in the OIDC implementation of the Provider using FORTH's LDAP server. Code is available at https://github.com/CARV-ICS-FORTH/django-oauth2-oidc-example.

ACKNOWLEDGMENTS

I would like to thank all the members of the CARV lab at FORTH for their help and support, especially Antony Chazapis for providing guidance, giving valuable comments and suggestions during this work.

REFERENCES

[1] Django Authentication Using LDAP. https://django-auth-ldap.readthedocs.io/en/latest/.
[2] Django OAuth Toolkit Documentation. https://django-oauth-toolkit.readthedocs.io/en/latest/.
[3] Django social auth documentation. https://python-social-auth.readthedocs.io/en/latest/configuration/django.html.
[4] Django, the high-level Python web framework. https://www.djangoproject.com/.
[5] LDAP, the Lightweight Directory Access Protocol. https://ldap.com/.
[6] OAuth 2.0, the industry-standard protocol for authorization. https://oauth.net/2/.
[7] OpenID Connect, a simple identity layer on top of the OAuth 2.0 protocol. https://openid.net/connect/.