Learning Objectives

- **Identify the pros and cons of using a third-party library**
- **Generate a GitHub access token and integrate it with Django**
- **Use Python to query GitHub and render the response in HTML**

Clone Course 4 Project

Clone Course 4 Project Repo

Before we continue, you need to clone the `course4_proj` repo so you have all of your code. You will need the SSH information for your repo.

In the Terminal

- Clone the repo. Your command should look something like this:

```
git clone git@github.com:<your_github_username>/course4_proj.git
```

- You should see a `course4_proj` directory appear in the file tree.

You are now ready for the next assignment.

GitHub Library

Third-Party Libraries vs Raw Requests

In order to make interacting with their APIs easier, many services provide Python libraries that wrap the API calls.

If you're ever needing to interact with a remote API, the chances are a Python library has been written for it, possibly by a third party. A Google search for *python library* is a good starting point.

When working with a library instead of making requests yourself, obviously you have to write less code. The libraries sometimes implement handy features to make navigating between objects easier.

To demonstrate, here's a sample of working with the PyGithub library, which we'll be looking at a bit more in depth soon.

The `Github` client class (`github.Github`) is the starting point. It's instantiated with an access token (which we'll see how to get later), which works in lieu of the username and password. Once you have an instance of `Github`, you can retrieve information about the logged-in user with the `get_user()` method, which returns a `github.AuthenticatedUser.AuthenticatedUser` class:

```
>>> from github import Github
>>> access_token = "…"
>>> g = Github(access_token)
>>> user = g.get_user()
>>> print(type(user))
<class 'github.AuthenticatedUser.AuthenticatedUser'>
```

We can then get information about the user, like their name:

```
>>> print(user.name)
Ben Shaw
```

What we should notice from this is that PyGithub is automatically translating GitHub's REST responses into Python objects. We can traverse through to other Python objects by using methods on these objects. For example, we can iterate over a list of repositories that the user has access to, by using the `get_repos()` method:

```
>>> for repo in user.get_repos():
...     if "django" in repo.name:
...             print(repo.name)
...             print(type(repo))
...
advanced-django
<class 'github.Repository.Repository'>
```

Here when we call `get_repos()`, PyGithub's `AuthenticatedUser` class is, behind the scenes, making another request (or requests) to GitHub to fetch that information. For each repository it finds in that response, it's transforming it to a `github.Repository.Repository` object, which we could then use to traverse further.

Compare that to making all these requests ourselves. We'd have to parse the data that was retrieved and figure out what API keys and IDs need to be fed into each URL to get the data we want. Using the library means all that hard work is done for us.

One downside that you might come across though, is that sometimes libraries get out of date if they're not well maintained. A service might make changes to their API and if the library is authored by a third party they could lag behind in updating their code to support the changes. This doesn't happen very often so on the balance of things, a library should be preferred if one is available.

Now let's see how to integrate Django with PyGithub. In particular, we're interested in how to make use of the user's access token.

Django Integration

The method of setting up integration/authentication of your application with a remote service is going to depend on your use case, and how the remote service works. We saw back in Course 1 how to set up authentication with Google. A client ID and key were generated and used to authenticate our application. Sometimes this information is all you need to work with a remote API. Other times, you also need a user key to identify each user who's logged in through your service.

With GitHub and Django Allauth, it's possible for a user to log in to your site using GitHub for authentication. You can then fetch their Allauth Social Application Token and use it to access GitHub as them (but only with the permissions they have granted your application). This is quite an elegant solution as all the token handling is done for you as part of authentication. However it does require quite a bit of setup at GitHub to register your application – similar to the process involved for setting up an application for Google authentication.

GitHub also allows a user to create personal tokens for authentication, which can be revoked at any time to remove access, in case the token is leaked. The user can feel more secure giving these tokens to a web site than handing over their username and password. For simplicity, it's this type of token we're going to use when building a site with GitHub integration.

In our example app we'll create a `Profile` class to store the token. Let's do that now, continuing work on the `course4_proj` project that we created in the last section.

Try It Out

Try It Out

First you'll need to install PyGithub, with pip:

```
pip3 install PyGithub
```

Then we'll create a new app, called `gh` (short for GitHub; if we call it `github` it will conflict with the PyGithub library). Create the app with the `startapp` management command:

```
python3 manage.py startapp gh
```

Then make sure to add `gh` to the list of `INSTALLED_APPS` in `settings.py`.

Open settings.py

Next, let's set up the model for the profile. This should be a familiar process to you, but to make it quicker you can just copy and paste this code into `models.py` in the `gh` app.

Open gh/models.py

```
from django.contrib.auth import get_user_model
from django.db import models


class Profile(models.Model):
    user = models.OneToOneField(get_user_model(),
        on_delete=models.CASCADE)
    token = models.TextField()

    def __str__(self):
        return f"Profile for {self.user.username}"
```

Then make sure to run the `makemigrations` and `migrate` management commands.

Next, we'll just use the Django Admin site to manage user profiles. The `Profile` model needs to be registered in Django admin, so open `admin.py` in `gh` and paste in this code:

Open gh/admin.py

```
from django.contrib import admin

from gh.models import Profile

admin.site.register(Profile)
```

We'll then need to create an admin user that can log in to the admin site. Remember this is done with the `createsuperuser` management command. Do this now:

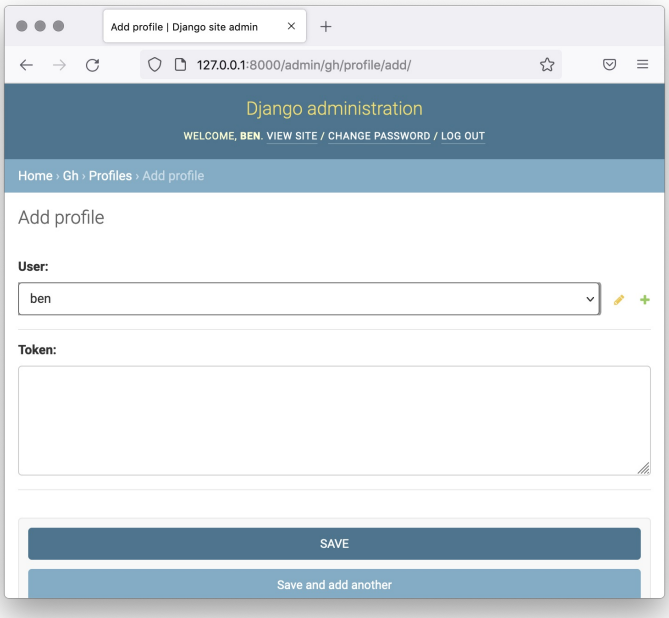
Open the terminal

```
python3 manage.py createsuperuser
```

Enter information according to the prompts:

```
Username (leave blank to use 'codio'):
Email address:
Password:
Password (again):
Superuser created successfully.
```

To verify that this is all working, start the Django dev server, and navigate to the `/admin/` page. Log in with the username (no email address) and password that you created. Navigate to the *Profiles* list, then the *Add Profile* page. Verify that you can see the form for selecting a user and entering a token.



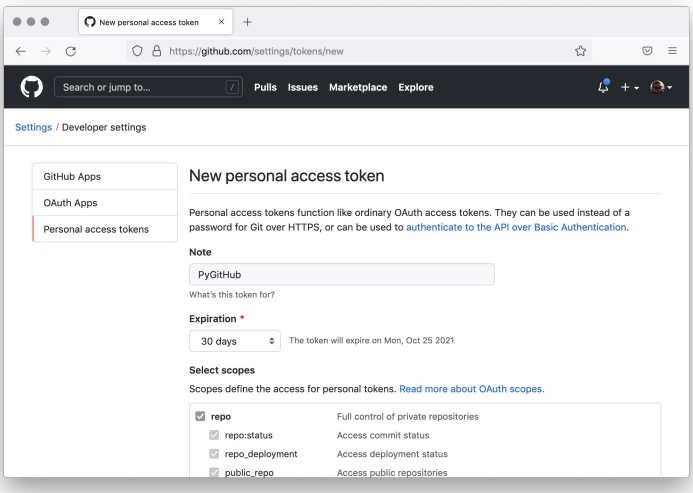
add profile form

View Blog

But right now, you don't have a token! So let's go and get one.

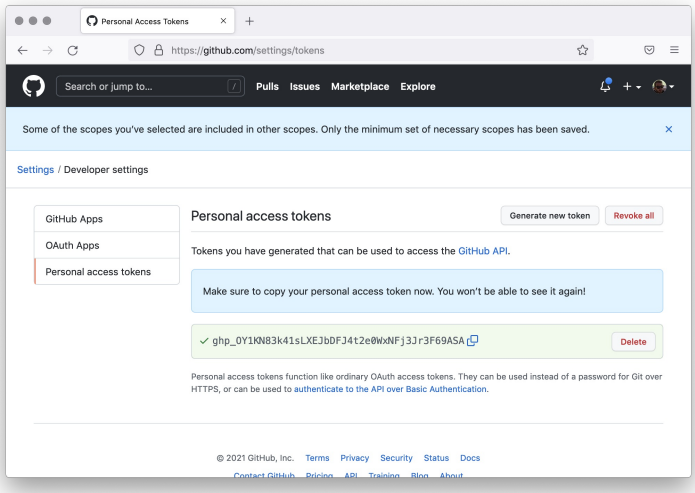
To continue with the setup, these steps assume you already have a GitHub account. If not, sign up for one now, it's free. You'll also need to verify your email address with GitHub; this is required.

Once your account is ready and you've logged in, you can visit the New personal access token page to generate a new token. The *Note* doesn't matter as it's just for your reference. Under *Select scopes*, just select **repo** which will select all the repository scopes.



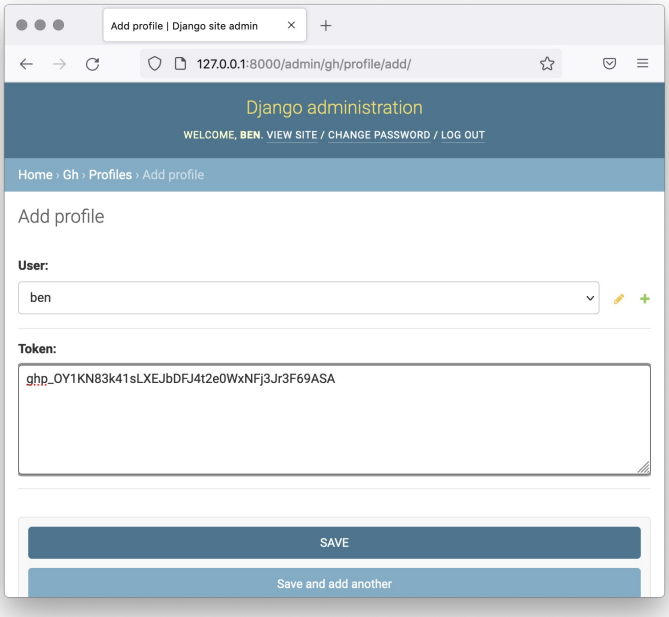
github token create page

Then scroll down the page and click **Generate Token**. On the next screen, you'll be shown the token. Make sure to copy it as it won't be shown again. Although, since you're not using it for anything else you can just create another new token should you misplace this one.



github token display

Now that you have the token, head back to the Django admin page and fill in the *Token* field on your user's Profile, then **Save** it.



token added to profile

Now that's done, we'll just need a list and template to show the information. We're just going to display a list of repositories to which the user has access.

For simplicity we're not going to add caching or repeated search prevention or anything like that. We'll query the GitHub API each time to fetch the list of repositories. Obviously, this is not the most efficient approach. For a real site, we would implement strategies as we saw in the previous section to cut down the number of requests. For example, on the user's `Profile`, also store a datetime of when the repositories were last queried and only update them once per minute/hour/day. We could also come up with our own `Repository` model to cache data in our local database. But since the focus of this section is integrating the PyGithub library with Django and how to store the user tokens, we're looking at that aspect rather than Django models, caching, and so forth.

Start by creating a `templates` directory inside the `gh` app, then create a file in it called `index.html`. Copy and paste this content into it:

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport"
          content="width=device-width, user-scalable=no,
        initial-scale=1.0, maximum-scale=1.0, minimum-
        scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Repositories</title>
</head>
<body>
<h2>{{ github_user.name }} has access to:</h2>
<ul>
    {% for repository in github_user.get_repos %}
        <li>{{ repository.name }}</li>
    {% empty %}
        <li>No repositories.</li>
    {% endfor %}
</ul>
</body>
</html>
```

It takes a GitHub user (`github_user`) as a context variable.Then it displays the GitHub user's name (`github_user.name`) and a list of repositories (from `github_user.get_repos()`).

Next open `views.py` in `gh`. Add these two imports:

Open gh/views.py

```
from django.core.exceptions import PermissionDenied
from github import Github
```

Then implement a single view function called `index`. It just creates a `Github` instance using the user's `token` from their `Profile`. We have some basic error handling in case the `Profile` or `token` is not set. Then, it passes the GitHub user instance to the template:

```
def index(request):
    if request.user.is_anonymous:
        raise PermissionDenied("Anonymous user not allowed.")

    if not request.user.profile:
        raise PermissionDenied("User does not have a Profile.")

    if not request.user.profile.token:
        raise PermissionDenied("User Profile does not have a
        token.")

    g = Github(request.user.profile.token)

    return render(request, "index.html", {"github_user":
        g.get_user()})
```

We won't be implementing any login or other use management views since we can just use Django admin's authentication system, so that's it for `views.py`.

Finally open `course4_proj/urls.py`. Add an import of the `gh` views:

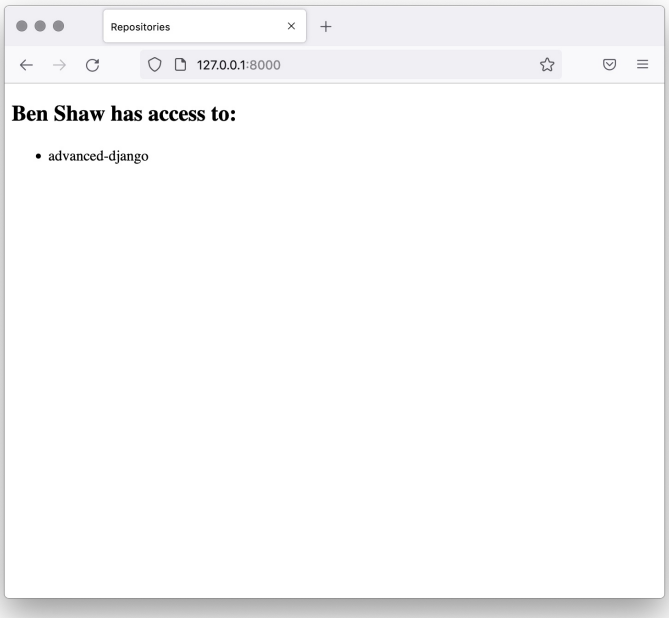
Open urls.py

```
import gh.views
```

Then route the index path by adding this to `urlpatterns`:

```
path("", gh.views.index)
```

Start your Django dev server, and navigate to the `/` (index) page. You should see a page like the one below:



repository list

View Blog

Although, you'll see your name and the list of repositories to which you have access.

Wrap-Up

Wrap-Up

This section was intended as a quick intro to using third-party libraries for API interaction, and how to store user-specific authentication data. We took a few shortcuts by using built-in Django features, like the admin authentication system, and no UI for users to add their own token. To summarize:

- Third-party libraries should usually be preferred, if they exist for the service you're working with. There can be risks that they become out of date, so check that they appear to be actively maintained before using.
- If you're using a service for authentication with Django Allauth, as well as its API, check if the tokens that Django Allauth stores can be used for API authentication too.
- A `Profile` (or other) class can be created to store tokens for a user. Don't store their username and password for the remote service.

And to summarize the things that we *didn't* do, but *should*, if we were building this into a real site:
- Set up Django authentication instead of requiring the user to log in via Django admin.
- Create a GUI for the user to set up their token(s).
- Add models to represent the remote objects we're using. For example, our own `Repository` model, that would allow for local caching.
- Caching to prevent multiple requests for the same user within a short period of time.

We saw how to set up these features in the previous section, and in earlier courses.

That brings us to the end of this module. In the next module, we're going to look at task queueing with the Celery library.

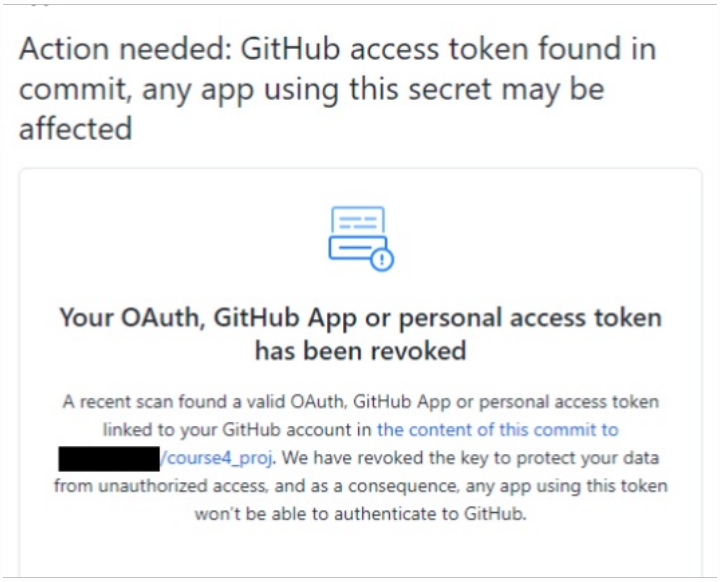
Pushing to GitHub

Pushing to GitHub

info

Important

You may see the following message when pushing to GitHub.



GitHub revocation of token

Some repositories have advanced security settings that scan all uploaded files for sensitive information. GitHub will revoke your token so others cannot access if they use your code hosted on GitHub. You would need to generate a new token and add it to Django through the Admin panel.

You can avoid this problem by disabling the secret scanning for your GitHub account, but that would decrease your security.

Another solution is to create a `.gitignore` file which allows us to ignore the file with sensitive information when pushing to GitHub. To do this, right-click on `course4_pro` and select `"New File..."`. Name this file `.gitignore`. In this new file, enter the name of the file we want GitHub to ignore.

```
db.sqilte3
```

Note, ignoring `db.sqlite3` will ignore the database for the project. Any information saved in the database (movie searches, GitHub repos, etc.) will no longer be available when you pull from the repo. However, Django will create a new database when you do a `manage.py migrate`.

Before continuing, you must push your work to GitHub. In the terminal:

- Commit your changes:

```
git add .
git commit -m "Finish github library"
```

- Push to GitHub:

```
git push
```