

Learning Objectives

- Describe the Django Allauth library
- Create a Google project through their developer portal
- Allow users to sign in with their Google credentials with Allauth
- Set up Google app credentials as a SocialApp

Clone Blango Repo

Clone Blango Repo

Before we continue, you need to clone the blango 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>/blango.git
```

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

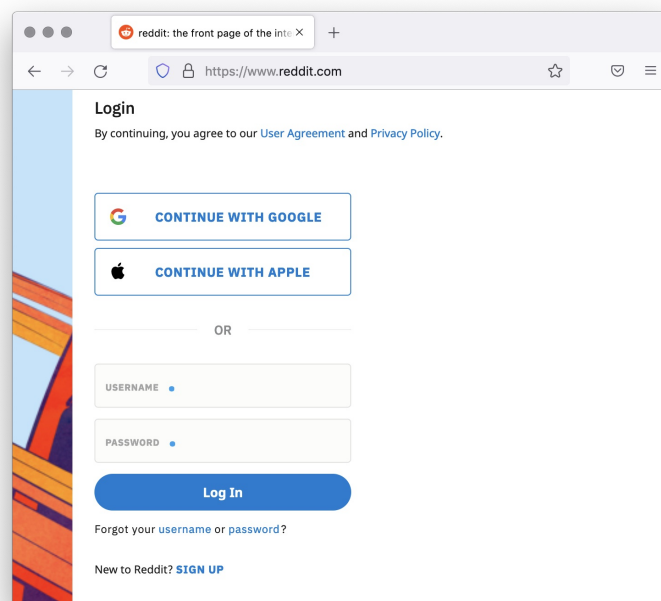
You are now ready for the next assignment.

Introduction & Purpose

Introduction & Purpose

Django Allauth is an extensive library that handles just about every aspect of account management and authentication in Django. This includes the signup and login process (replacing Django's built-in login views), verifying email address, managing multiple email addresses for users (and setting a default), forgotten password management, and more.

It also provides functionality to allow your users to log in via third-party or social apps, like Google, Facebook, Twitter, Github and more. You've no doubt seen other sites give this functionality. For example, here's Reddit's login page which allows you to log in with your Google Account, Apple Account or the username/password you created with Reddit.



reddit login screen

▼ Login and registration

Since we've already implemented login and registration with other means, we will just be using Django Allauth for social login. However in your own project you could elect to use Django Allauth for everything auth-related. If you do want social authentication, it's best to not remove the username field from a custom User model, as it makes it easier for Django Allauth to save the user data. Also if using Django Allauth, you can use its email

verification system, removing the need for Django Registration. We will make sure all the libraries work together in our project though.

Third-Party Login Flow

How does third-party authentication work? When you click a link to log in with a third-party, say, Google, you're taken to a page hosted by Google, which will ask you to confirm that you want the authenticating web site (Blango) to get access to your Google account.

▼ Credentials

Before the third party will allow an application to connect and authenticate, you must have credentials (like a client ID and secret key) set up with them. A Django (or other) application can't just arbitrarily authenticate a user and fetch their private information without proving they aren't up to no good.

Once you confirm, Google redirects your browser back to a page hosted on Blango, and includes a token identifying you. Blango then makes a request in the background to Google, and provides this token, to fetch your user ID and other metadata that you allow from Google. Django Allauth will store the token so that later you can re-authenticate the user (in lieu of a username/password).

▼ Google OAuth

There is a standard for this kind of authentication, which Google, and others, implement. It's called OAuth 2.0, and if you're curious about the full specification you can read about it on the [OAuth 2.0 Homepage](#)

Once your Google profile information is retrieved by Blango, it has to be saved somewhere. Django Allauth will create a Django User that's available to be used in ForeignKeys and so on. It then creates a mapping between the Django User and the Google (or other third-party service) profile. To save the mapping, and other things, Django Allauth uses these models:

- **SocialApp**: Stores information about a third-party provider, such as your app's credentials (client ID/secret key).
- **SocialAccount**: Stores the profile information about a user from the third-party provider.
- **SocialToken**: Stores the token data that can be used to authenticate the user against the third-party service.

When configuring a third-party provider you'll need to create a **SocialApp** object (usually through the Django Admin, but can be done programmatically). Other than that, you shouldn't need to interact with these models, Django Allauth will handle it automatically.

▼ Syncing problems

That being said, sometimes your stored tokens and the third-party service can get out of sync, and to debug the problem you might have to delete `SocialToken` and `SocialAccount` objects.

At a high level, integrating social authentication is similar for every provider.

- Create an account with the third-party provider, and set up an authentication application with them.
- The third party provides you with a client ID and secret key (although these terms might differ based on the provider).
- Add a `SocialApp` object to the Django database, provide the client ID and secret key for your application.
- Add a link/button to your login page to initiate the third-party workflow.

On the next page, we'll go through this process in more detail, to set up authentication to Blango using Django Allauth to connect to Google.

Authenticating with Google OAuth

Authenticating with Google OAuth

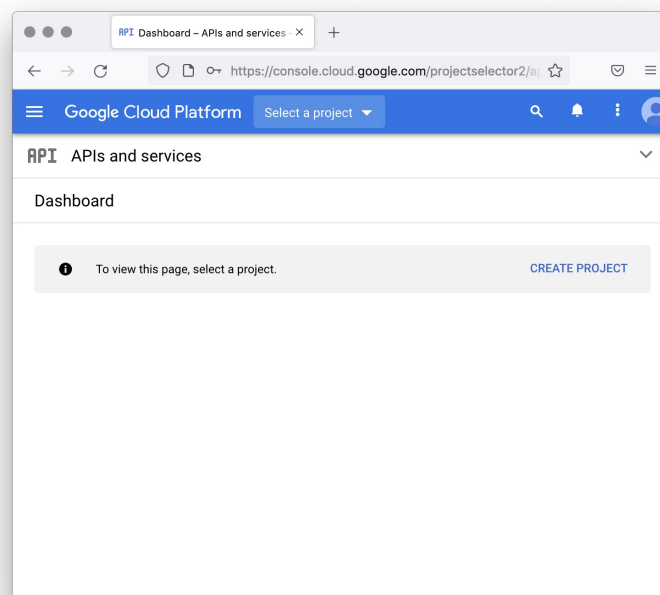
A Tutorial

The previous page of this course is designed to be like a walkthrough. It will go through the process of setting up a Google project, then generating OAuth credentials for it. Following that we'll install Django Allauth, configure it, then set up the URLs and a link to initiate the Google login flow. Then we can test it out.

Creating a Google Project

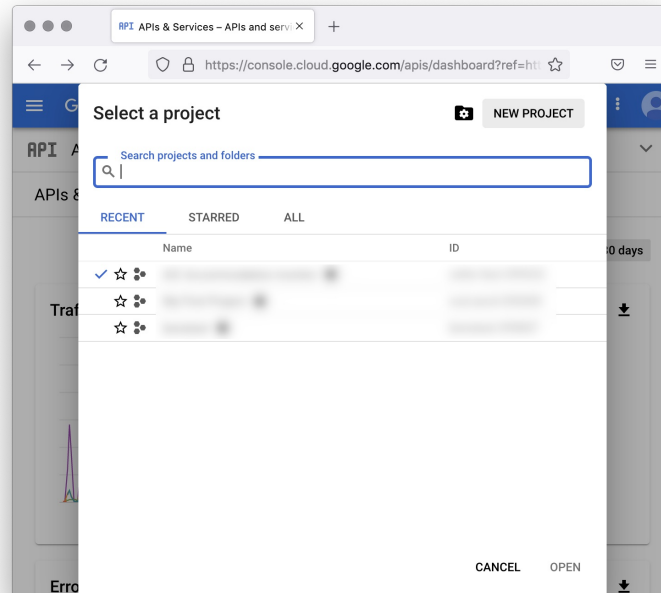
Creating a Google Project is done through the [Google Developer Console](#). Provided you have a Google account, you should be able to log in there.

Once logged in, if it's your first time using the Google Developer Console, you'll have to set your region. But once that's done (or you've used the Console before), click on the **Select a project** button at the top of the page.



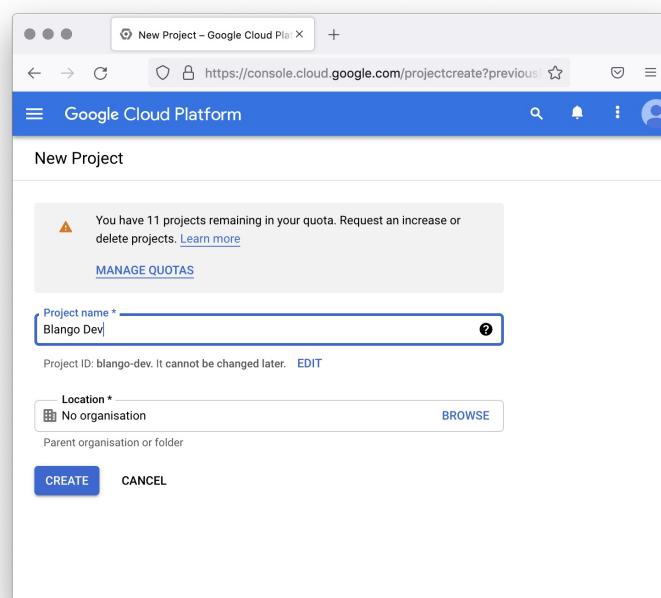
select a project

That will bring up a modal window showing the current projects you have access to (it might be empty). Click the **NEW PROJECT** button at the top of the modal.



click new project

You'll be taken to the *New Project* form. Enter a *Project name*, and you can leave *Location* as **No organization**. Then click **CREATE**.



new project form

After it's created, you should see the name of the project at the top of the page. Now that we have a project set up we can generate API credentials (client ID and key).

Generating API Credentials

Generating API Credentials

In the Google Developer Console, open the left menu, then click on **APIs and services** > **Credentials**. **Important**, your Google screens may look slightly different, but you will still be able to accomplish each task.

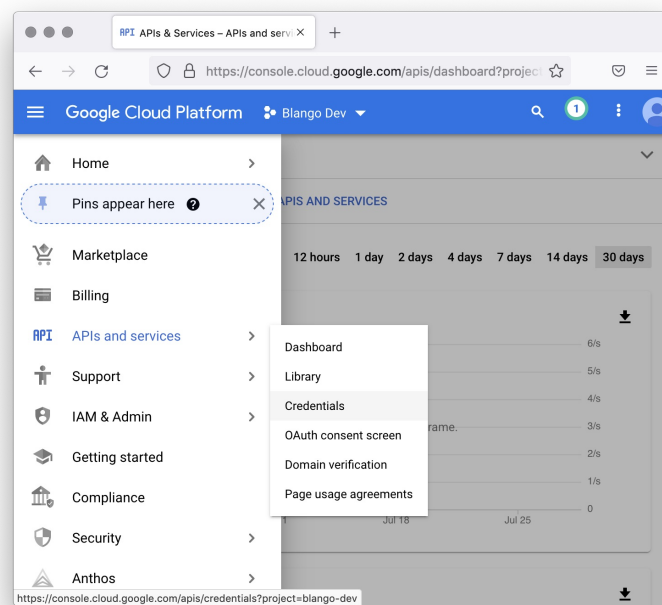


Figure 4

At the top of the page, click on the **CREATE CREDENTIALS** button, which will show a dropdown. Select **OAuth client ID** from the dropdown menu.

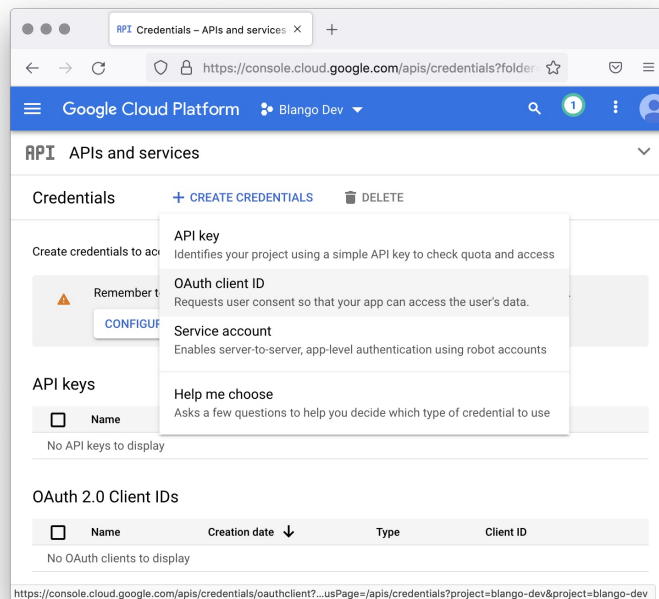


Figure 5

Before you can create a client ID, you'll need to set up the consent screen so that users know the name of your app. Luckily Google provides a big button to guide you. Click the **CONFIGURE CONSENT SCREEN** button.

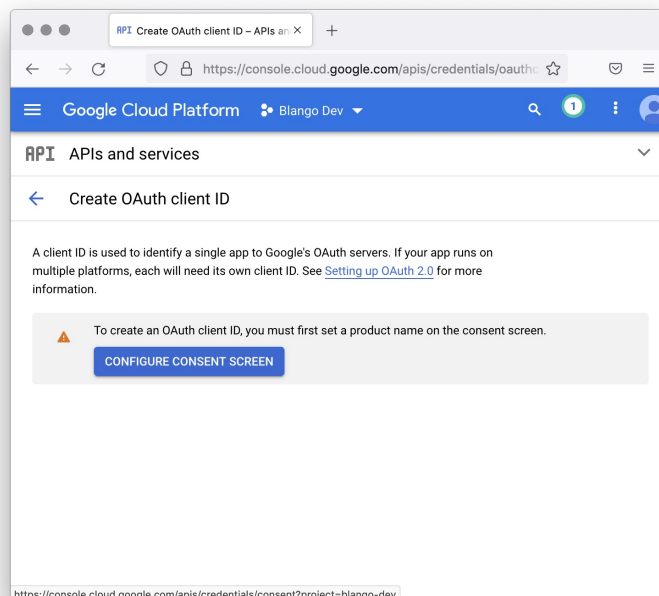


Figure 6

On the *OAuth consent screen* page, choose the **External** option then click **CREATE**.

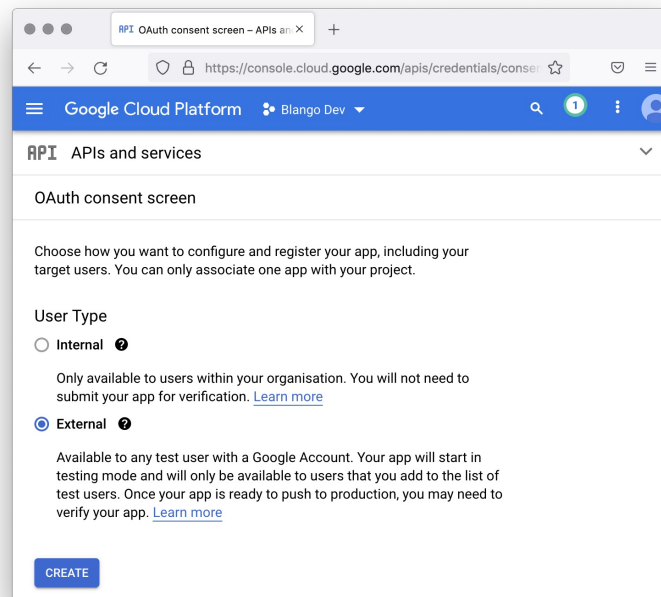


Figure 7

On the *Edit app registration* page you'll need to enter an *App Name* and your email address for the *User support email* (don't worry, since we're just testing you won't receive a bunch of support emails).

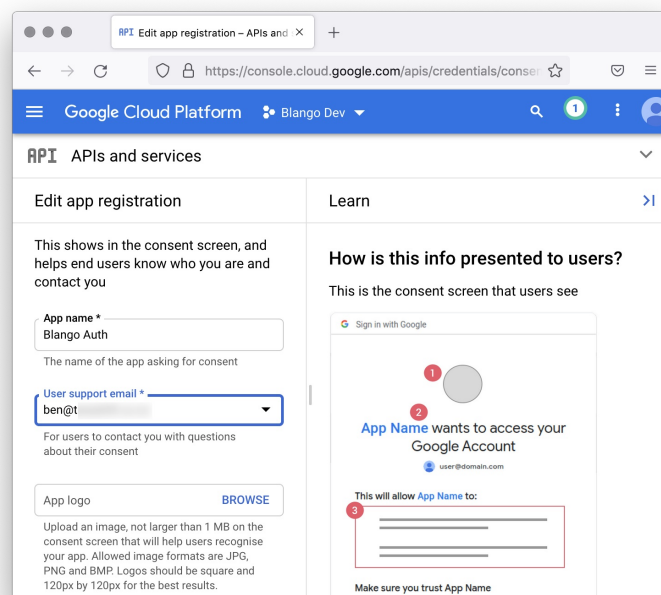


Figure 8

Scroll down and enter your email address in the *Email addresses* field under *Developer contact information*. You can leave the other fields blank. Then click **SAVE AND CONTINUE**.

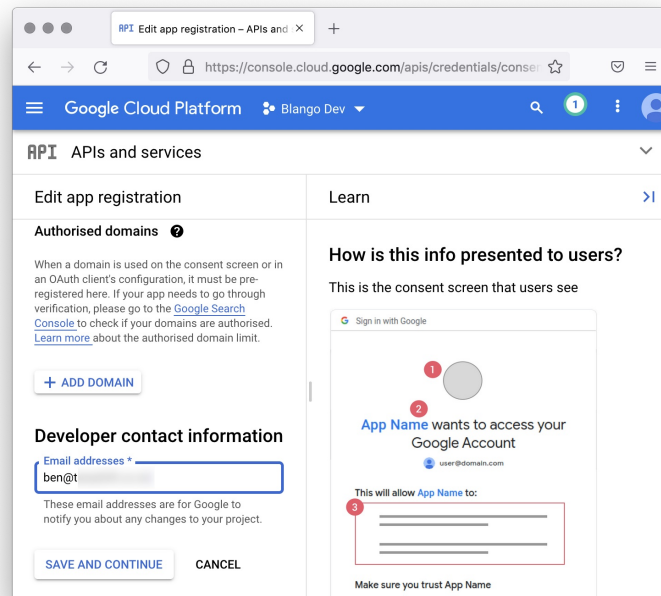


Figure 9

Now that you've create the consent screen, we'll set up the scopes for the app. These are the permissions that you request to access, so that your users can trust that your app isn't accessing any resources that they don't expect. Click the **ADD OR REMOVE SCOPES** button.

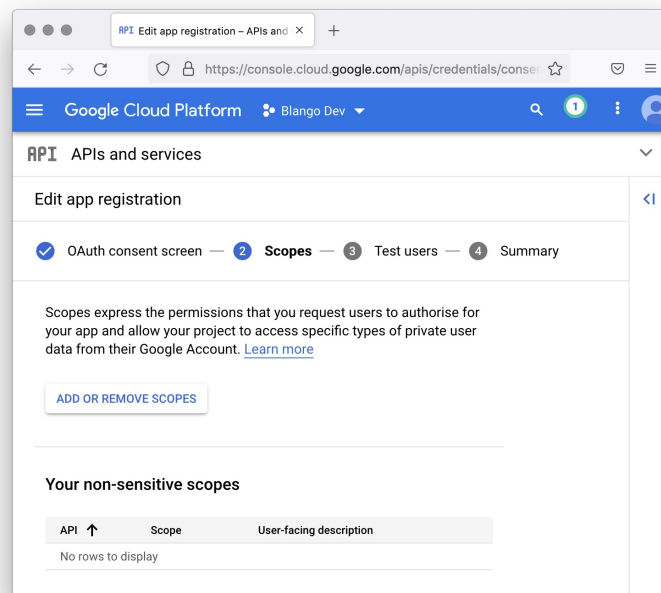


Figure 10

On the *Update selected scopes* screen, check the top two scopes, *.../auth/userinfo.email* and *.../auth/userinfo.profile*. We need access to the email address so we can set it on the user as the primary key, and we need access to the profile to populate the user's name.

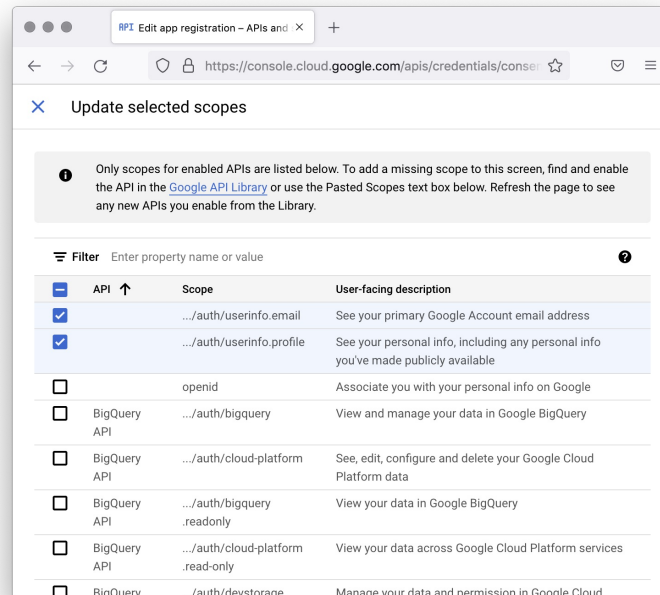


Figure 11

Scroll to the bottom of the page and click **UPDATE**.

The modal will close. Scroll to the bottom of this page, and click **SAVE AND CONTINUE**.

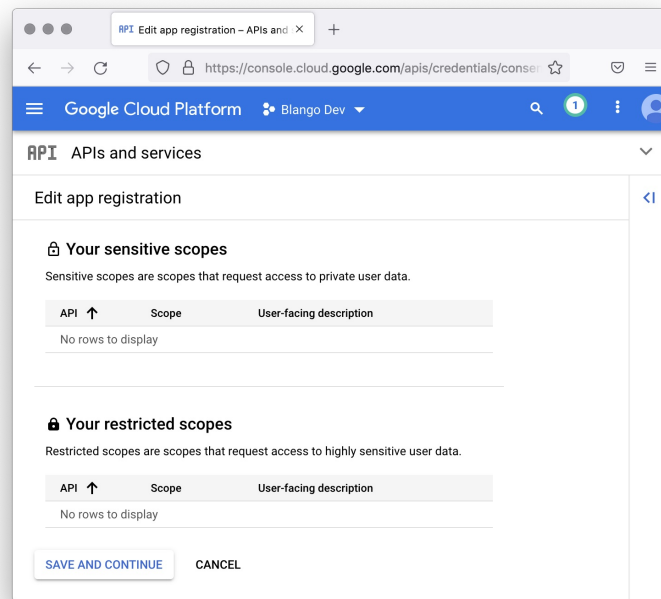


Figure 12

Since the app is running in test mode, it will only allow certain users to log in. We need to add them on this *Test users* screen. Click on the **ADD USERS** button, and enter your email address in the modal that appears. Then click **ADD**

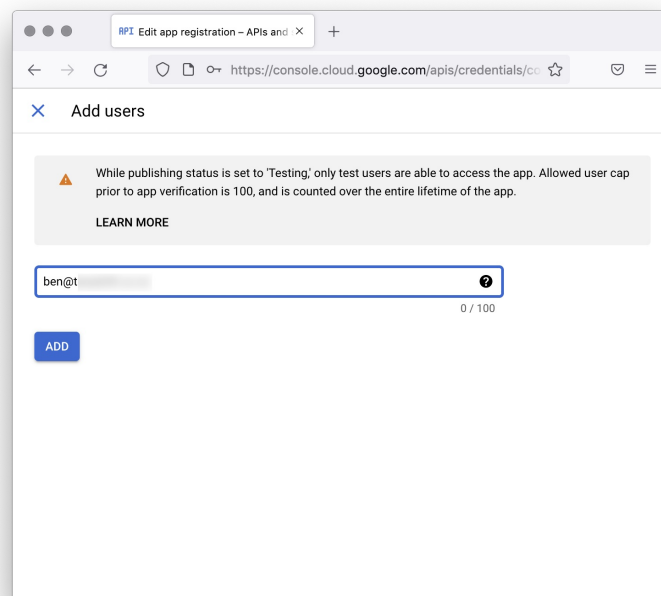


Figure 13

▼ Email address

If the email address you've entered is already assigned to a user in Blango

you might run into problems, it's best to change the user to have a different address in Blango first.

You'll see your email address in the *Test users* table. Click **SAVE AND CONTINUE**.

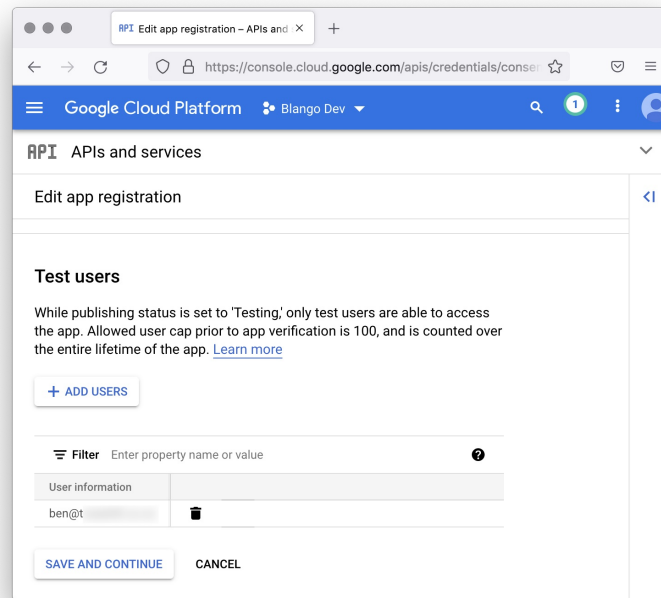


Figure 14

You'll now see a summary of your *OAuth consent screen* information. Scroll to the bottom (you can review the details on the way down) and then click **BACK TO DASHBOARD**.

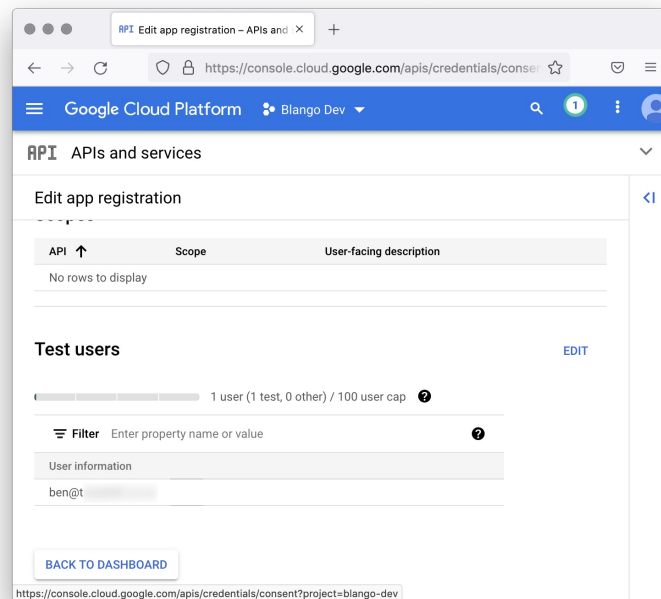


Figure 15

Now that you have the OAuth consent screen set up, you can return to the credentials screen by clicking through the left menu and then **APIs and services > Credentials**.

Once again, at the top of the page that loads, click on the **CREATE CREDENTIALS** button, which will show a dropdown. Select **OAuth client ID** from the dropdown menu.

On the *Create OAuth client ID* page, from the *Application type* menu, select **Web application**.

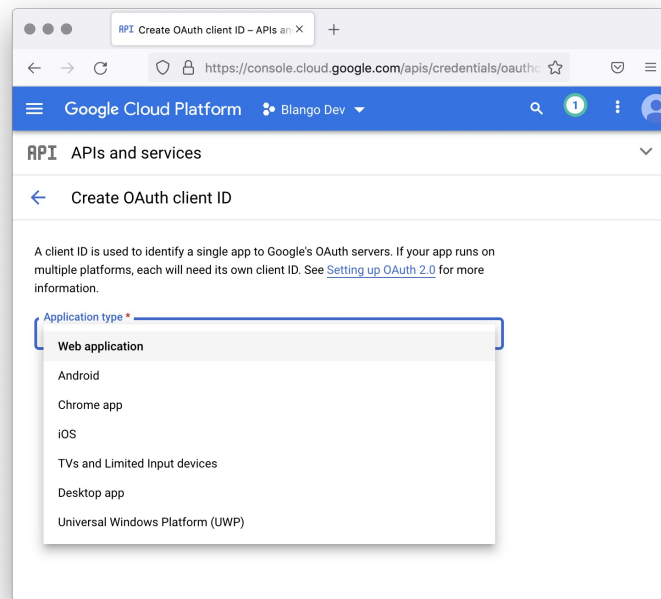


Figure 16

This will reveal some more fields. Enter a name for the app in *Name* field. Then under *Authorised JavaScript origins*, you should add your Django base URL, including the `http://` or `https://`. You can add multiple URLs, so you can add your `http://[host]-8000.codio.io` host as well as your `127.0.0.1` (including the port) for local development.

▼ Finding your Codio host

In the Codio menu bar, click “Project” => “Box Info”. This will open a tab with your Codio host name in it.

The *Authorised redirect URIs* are the URLs that Google will allow redirecting back to, to pass in the secret token. This is an extra security measure to prevent a bad request from redirecting to a malicious site after generating the token. We’ll need to put in the URLs that Django Allauth will use, which have the path `/accounts/google/login/callback/`. Once again you can add both your Codio URL

(`http://[host]-8000.codio.io/accounts/google/login/callback/`) and a locally hosted version (`http://127.0.0.1:8000/accounts/google/login/callback/`).

Note, use `http` and not `https` for these URLs. Once you’ve entered them, scroll down and click **CREATE**.

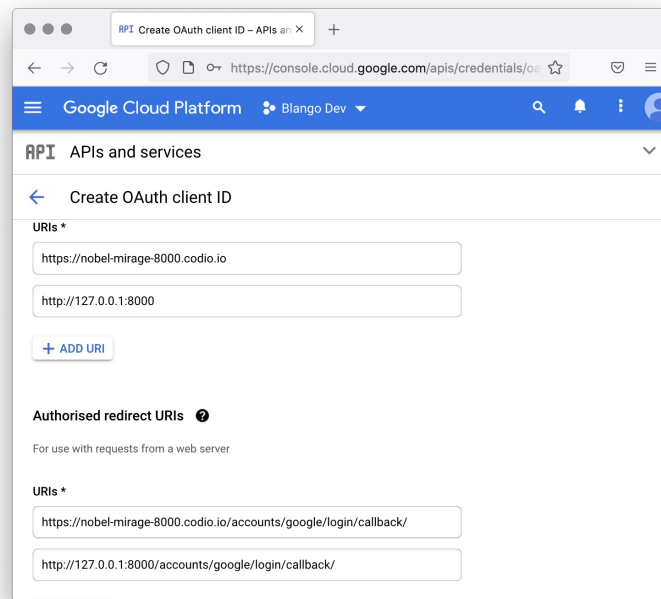


Figure 17

You'll then be shown your *Client ID* and *Client Secret* values. Copy these for reference.

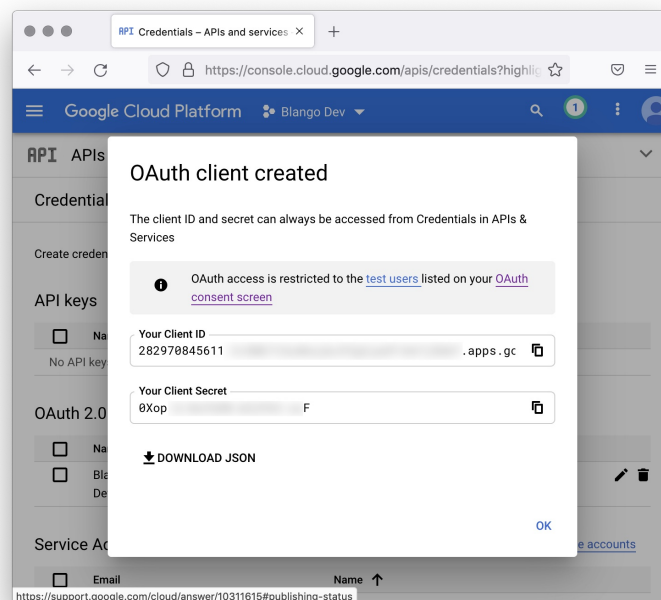


Figure 18

Now our Google app is set up, we can get Django Allauth installed and configured.

Django Allauth Setup

Django Allauth Setup

As you might have guess, Django Allauth is installed using pip. Do that now:

```
pip3 install django-allauth
```

▼ Installation error

If you see an installation error for Django Allauth, that could mean there was a recent change to Python wheels. Run these two commands to update pip and setuptools.

```
sudo python3 -m pip install -U pip
sudo python3 -m pip install -U setuptools
```

▼ Warning

At the time of writing, there is an incompatibility between Django 3.2+ and Django Allauth 0.45.0. Django 3.2 introduced a requirement that the auto ID field type must be defined in the settings, but third-party apps need to be updated to set this too. The symptom of this bug is that after migrating to set up Django Allauth, the next time you migrate in the future you'll receive the error:

```
Your models in app(s): 'account', 'socialaccount' have changes
that are not yet reflected in a migration, and so won't
be applied. Run 'manage.py makemigrations' to make new
migrations, and then re-run 'manage.py migrate' to apply
them.
```

You can follow the directions to make the migrations and re-migrate, which fixes the issue, but it ends up creating the migrations files in the Python library directory and is not an ideal solution.

We'll remind you to perform this fix after the first migration, but we hope the bug is fixed in a future release.

After Django Allauth is installed we need to start by making some changes to settings.py, first by adding some `INSTALLED_APPS`.

First, "django.contrib.sites" must be added, put it after "django.contrib.messages".

▼ Django sites

The Django Sites app/framework is a way for two sites/domains to share the same Django project. You would do this if you had two similar sites, such as two sister news sites, that shared the same models, data and views. Each site can have its own settings and templates, and you could, for example, choose if a news story was displayed in one or both of the sites. You can think of it as a virtual Django project. If it sounds like something useful for a site you're working on, check out the [Sites framework documentation](#).

Django Allauth uses the Sites framework so that different SocialApp objects (i.e. the client ID/keys for configuration to communicate with the third-party service) can be assigned to different sites. This is important so that, for example, the correct consent screen is shown depending on which site the user is logging into.

Now, back to INSTALLED_APPS updates.

Next add "allauth", "allauth.account", "allauth.socialaccount", "allauth.socialaccount.providers.google" to the end of your INSTALLED_APPS.

The first three are required, and then you'll add the provider modules that you're using for authentication. For Blango, it's just Google, but Django Allauth has [a huge list of providers it supports](#).

Next we need to add a setting so the Django knows which Site object our settings file applies to. Django will automatically create one when we next migrate, and we'll only use this single Site object in Blango. The site will have the ID 1 so we need to add this setting:

```
SITE_ID = 1
```

Normally when Django Allauth creates a User object from a social account login, it will generate it a username based on the user ID at the third party. Since our custom User model doesn't have a username field, Django Allauth will fail, unless we make some settings changes.

Add these settings:

```
ACCOUNT_USER_MODEL_USERNAME_FIELD = None
ACCOUNT_EMAIL_REQUIRED = True
ACCOUNT_USERNAME_REQUIRED = False
ACCOUNT_AUTHENTICATION_METHOD = "email"
```

These four settings tell Django Allauth (respectively) that:

- There is no username field on the User model.
- The third-party provider must provide an email address when authenticating.
- The username of the User is not required.
- The user authenticates by entering their email address.

▼ Authenticating with email

Since we earlier decided that our User model would require a valid email address, it does limit the third-party providers we can use to ones that know the user's email address. We don't have an extensive list of which do or do not, but it is something to consider when making decisions around authentication. Furthermore, we're making the assumption that the email addresses are valid and that the third-party provider has performed some sort of adequate validation.

That's all for the settings changes, now we need to set up the URLs. Open `urls.py`, and add this rule to the `urlpatterns`:

Open `urls.py`

```
path("accounts/", include("allauth.urls")),
```

This rule must come after the mapping from `accounts/` to `include("django.contrib.auth.urls")`, since both sets of URLs define some of the same rules (for example, `login/`), by putting the Django URLs first they will have precedence and we'll continue to use them instead of Django Allauth's.

Finally, let's add a link to do the Google login onto our login page. Open the `login.html` template and underneath the lost password link, add this:

Open `login.html`

```
<p>
  <a href="{% url 'google_login' %}">Log in with Google</a>
</p>
```

Django Allauth provides a named URL for each of its providers, so you can create links to them in this way. For example, link to the Twitter auth flow with `{% url 'twitter_login' %}`.

That's all the code changes we need. Go ahead and run the `migrate` management command.

```
python3 manage.py migrate
```

You should see the following output:

```
Operations to perform:
  Apply all migrations: account, admin, auth, blango_auth, blog,
contenttypes, sessions, sites, socialaccount
Running migrations:
  Applying account.0001_initial... OK
  Applying account.0002_email_max_length... OK
  Applying sites.0001_initial... OK
  Applying sites.0002_alter_domain_unique... OK
  Applying socialaccount.0001_initial... OK
  Applying socialaccount.0002_token_max_lengths... OK
  Applying socialaccount.0003_extra_data_default_dict... OK
```

▼ Migrate before makemigrations

You normally wouldn't have to do this, but since we're trying to workaround the Django 3.2 / Django Allauth 0.45.0 bug, run the migrate command again. If you see the error

```
Your models in app(s): 'account', 'socialaccount' have changes
    that are not yet reflected in a migration, and so won't
    be applied.
```

Then run makemigrations and migrate.

Now run makemigrations.

```
python3 manage.py makemigrations
```

Notice that the migrations are generated in Django Allauth's install directory.

```
Migrations for 'account':
  /.../site-packages/allauth/account/migrations/0003_auto_20210728_1100.py

  - Alter field id on emailaddress
  - Alter field id on emailconfirmation
Migrations for 'socialaccount':
  /.../site-packages/allauth/socialaccount/migrations/0004_auto_20210728_1100.py

  - Alter field id on socialaccount
  - Alter field id on socialapp
  - Alter field id on socialtoken
```

You can then run the migrate command one final time to assure yourself that everything is back to normal and you see the message No migrations to apply..

```
python3 manage.py migrate
```

Your output should look something like this:

```
Operations to perform:
  Apply all migrations: account, admin, auth, blango_auth, blog,
                        contenttypes, sessions, sites, socialaccount
Running migrations:
  Applying account.0003_auto_20210728_1100... OK
  Applying socialaccount.0004_auto_20210728_1100... OK
```

Now we need to set up our Google app credentials as a SocialApp.

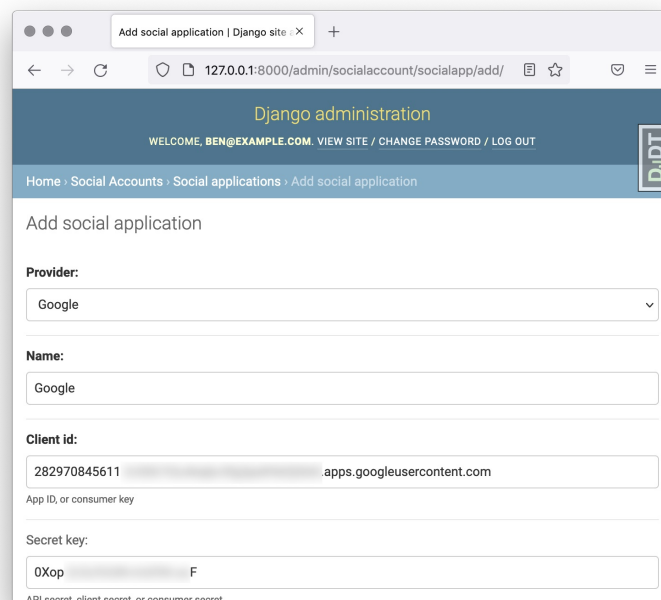
SocialApp Setup

SocialApp Setup

Setting up a `SocialApp` object could be done programmatically, but it's easy to do it through the Django Admin. Start the Django development server if it's not already running, then head to the Django Admin page. Under *Social Accounts*, create a new *Social Application*. Enter the following details:

[View Blog](#)

- There should be only one *Provider* available, select *Google*.
- The *Name* doesn't matter too much, but pick something meaningful. You could choose something like **Google** or **Google Blango Dev** (including the app name at the provider).
- The *Client ID* is the *Client ID* from Google, which ends with `.apps.googleusercontent.com`.
- *Key* is not required.
- *Secret Key* is the *Client Secret* from Google, it's a string containing alphabet characters and dashes.



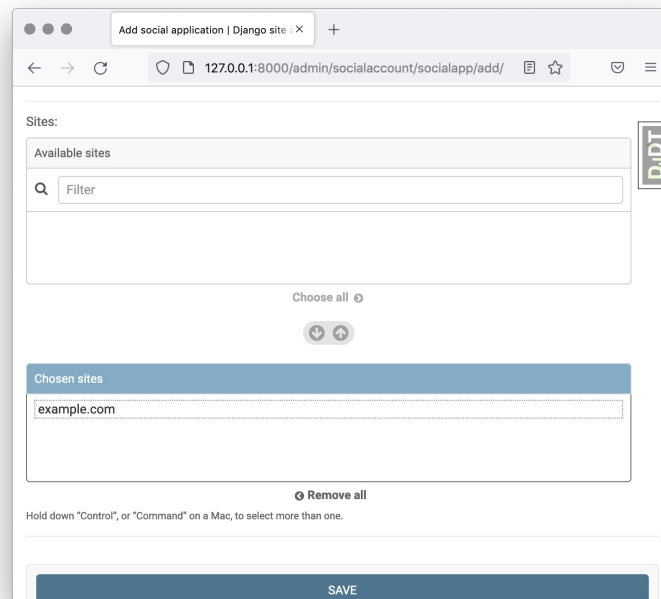
The screenshot shows a web browser window with the Django Admin interface. The page title is 'Add social application | Django site'. The URL bar shows '127.0.0.1:8000/admin/socialaccount/socialapp/add/'. The page header includes 'Django administration' and a welcome message for 'BEN@EXAMPLE.COM'. The breadcrumb trail is 'Home > Social Accounts > Social applications > Add social application'. The form itself is titled 'Add social application' and contains the following fields:

- Provider:** A dropdown menu with 'Google' selected.
- Name:** A text input field containing 'Google'.
- Client id:** A text input field containing '282970845611' followed by a truncated string ending in '.apps.googleusercontent.com'.
- Secret key:** A text input field containing '0Xop' followed by a truncated string ending in 'F'.

Below the 'Client id' field, there is a small text label: 'App ID, or consumer key'. Below the 'Secret key' field, there is a small text label: 'API secret, client secret, or consumer secret'.

social app credentials 1

- Scroll down, and add the Site object that your `SocialApp` will belong to. We only have 1, which is `example.com`

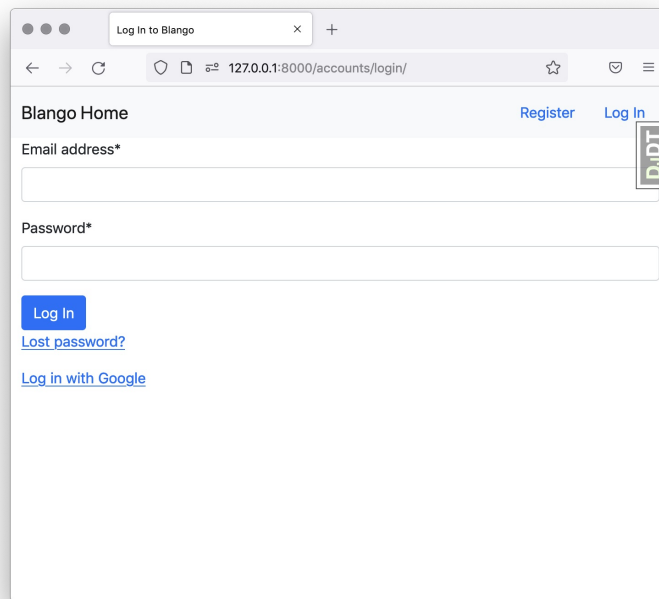


social app credentials 2

▼ Site

The name of the Site can be changed by editing the Site object through the Django Admin. Since we're not going into production then leaving it as `example.com` is fine for us.

Now save the SocialApp. Log out of the admin site, and then return to our normal login page. You should see the *Log in with Google* under the lost password link.



login with google link

Click on it, and you'll be taken to a Google page to log in, or select your account. The next screenshot shows an already logged-in user who just needs to select their account. Make sure the account you choose is one that has been added to the test users list in the Google Developer Console.

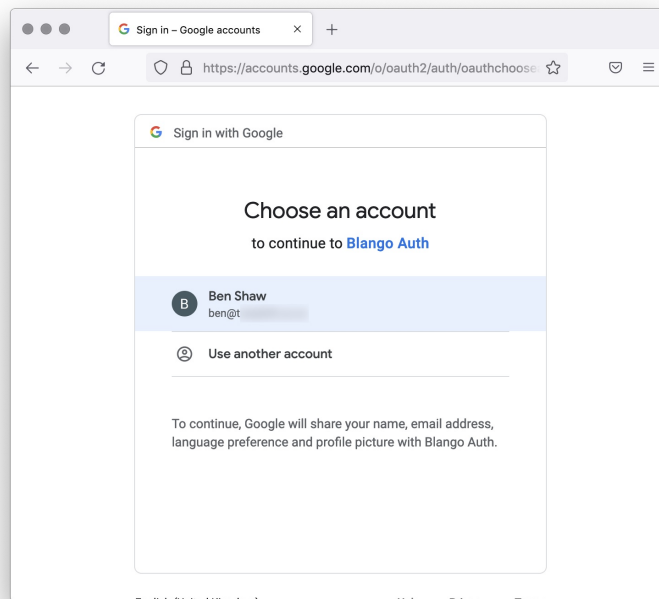
info

Logging in With Google

To log in with Google you will need to load your blog into its own tab. Click on the blue arrow on the left to load Blango in a separate tab. From there, you can complete the log in process.

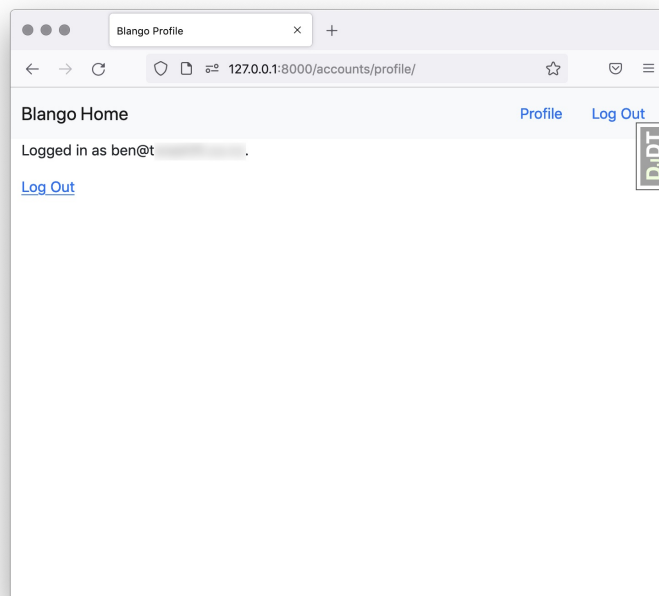


open in a new tab



select account page

Your browser will do a few redirects and after a few seconds you'll end up back on the user profile page, logged in as the user you selected through Google.



social auth logged in user

It's not a very difficult process to set up third-party authentication, and once you've done one provider others are similar. We haven't covered the process of users signing up for Blango with third-party auth when they

already have an account with the same email address. If you try this, you'll see that Django Allauth attempts to resolve the problem for you by asking the user for their email address, without verifying it, which circumvents Django Registration's protection we've set up.

In a real site, you probably wouldn't use both Django Registration and Django Allauth together. Use Django Registration for a simple two-step signup process for a site that doesn't require any social authentication. Otherwise, use Django AllAuth and have it handle login, signup and verification of emails (by adding the `ACCOUNT_EMAIL_VERIFICATION = True` setting). You can then also get social authentication and have it work more seamlessly.

The purpose of this module was to illustrate a number of different ways of customizing Django authentication, building up from simple to more complex solutions.

Conclusion

That's the end of the module and the first course. We continue extending Blango with a REST API in the next course.

Pushing to GitHub

Pushing to GitHub

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

- Commit your changes:

```
git add .  
git commit -m "Finish django Allauth"
```

- Push to GitHub:

```
git push
```