Implement an OpenID Connect Client

[**Description**](#_qrxwy8crl1e5) **1**

[**Create a Client ID and Secret**](#_qfv8lxs4egb8) **1**

[**First Things First**](#_et65jmw7gkf1) **2**

[**Make a Button, use 3-Legged OAuth Flow**](#_929hkc5buqal) **4**

[CSRF Protection Required](#_rksnuwq0cojr) 7

[Redirect URI](#_d094yvvd0frm) 7

[JWT Unpacking](#_9yw4zbn3m9a2) 9

[**Creating a Session**](#_ghksut39s6d9) **9**

[**Grading**](#_amewoxifsjql) **9**

# Description

For this project we will add a third-party OpenID Connect button to our login page. This will allow us to, for example, login with Google. You may choose any ID provider that you like. Instructions for getting the client ID and client secret for Google are listed here, if you decide to go that way: <https://developers.google.com/identity/protocols/OpenIDConnect>. Note that these instructions are great for getting keys and IDs, but not so great for actually implementing your login button and redirect URI. Better instructions will be given for that below.

Do explore just a little bit. You might want to try using Amazon Cognito, or Auth0, or Okta, or some other identity provider. Any of these are fine. Be aware that your redirect URI should probably be your **deployed application URL**. If you set one up for localhost (which can work, since the browser is making the requests), you will actually need a *separate* client ID. Just be careful of that and pick one.

# Create a Client ID and Secret

Whatever provider you use, you will need to do some setup before starting. You need to get an account with them and find a way to get a **Client ID** and a **Client Secret**.

You will be asked to provide

* Application Name: a string that identifies your application. Will go into your request URL
* Redirect URI: this should be “https://yourserver.appspot.com/oidcauth”.

Your URL **can be different** if you want it to be, but I’ll assume it’s `/oidcauth’ for the discussion here just to make it concrete. Whatever you use, *it has to be what you registered* with the ID provider.

There are many free providers, or providers with free trials. Use one of those.

If you go with Google, the instructions are here for getting a Client ID and Secret: <https://developers.google.com/identity/protocols/OpenIDConnect>

Once you have a Client ID and Client Secret, it’s time to implement OpenID Connect!

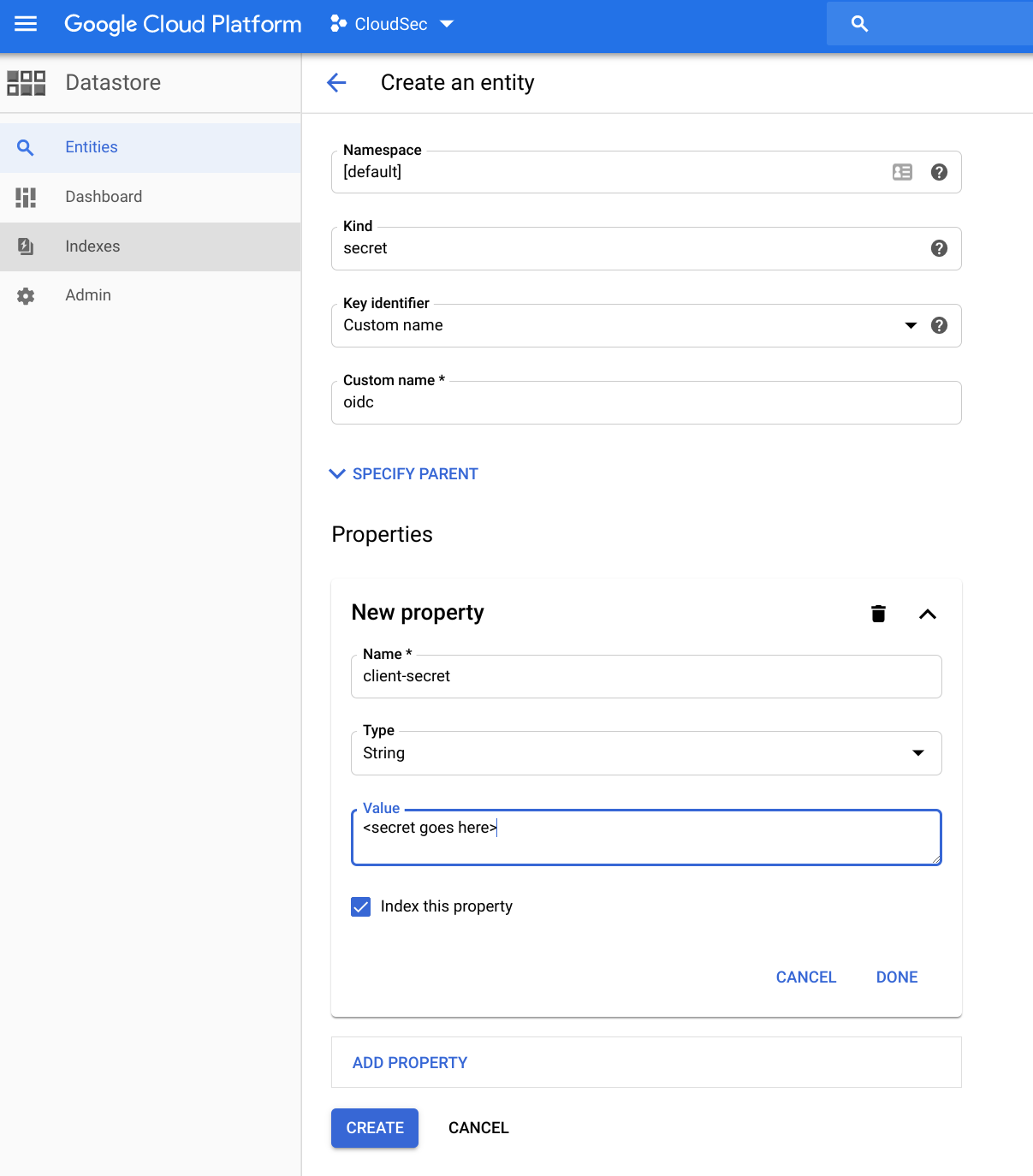
# First Things First

What do you do with your client secret? You’ll need to put it somewhere, but where?

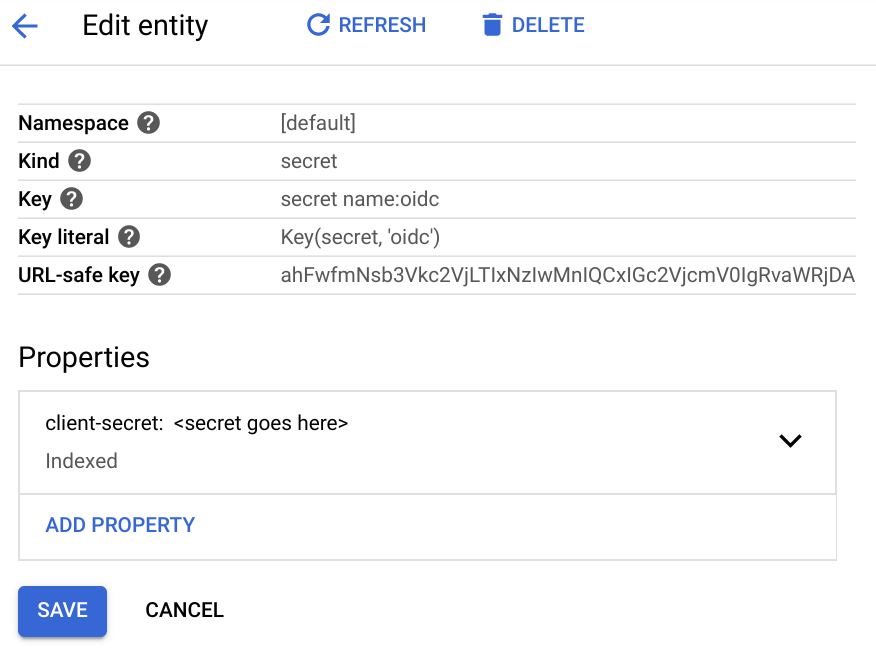
The answer is definitely ***not*** “in your code”. That is a terrible location for secrets, largely because *your code is rarely actually secret*. So, no secrets in your code.

There are key management services like <https://cloud.google.com/kms/>, but those might be overkill for a lab. Instead, for this lab you can store your secret in the datastore. That, at least, can only be viewed by an administrator, and you won’t leak the secret to either the code or the browser.

In my case, I created a new kind and key name like this:



After you create that entity, you can click on it to see its key, and use that to access it from your code. If you follow the above example, the key screen (after selecting the new entity) looks like this:



Once it’s in there, you can easily get it using the same exact key:

|  |
| --- |
| client = datastore.Client() secret = client.get(client.key('secret', 'oidc'))['client-secret'] |

Now we’re ready. We have a place to store our client secret that is protected behind a login (the admin page after deployment will only let **you** view the datastore), and it is not in code. Cool.

# Make a Button, use 3-Legged OAuth Flow

Your button or link will basically send the user to an identity provider site. You need a CSRF authentication token (called “state” in the standard) to do so. Thus, when you generate your login form with the button, it will need to have that (random) state token in it.

Let’s say we have the following, either from registering with our provider, or generated:

* Assigned by provider:
  + Client ID
  + Client Secret
* Chosen by you:
  + Application Name
  + State (a CSRF token)
  + Redirect URI (pick a name like **yourserver.appspot.com/oidcauth**)[[1]](#footnote-0)
  + Nonce (random integer)

Let’s assume you have the following values for the sake of discussion:

* client\_id=myClientID
* client\_secret=myClientSecret
* state=abcd1234
* nonce=1234
* redirect\_uri=https://yourserver.appspot.com/oidcauth

You’ll create a login link that has everything in it but the client\_secret:

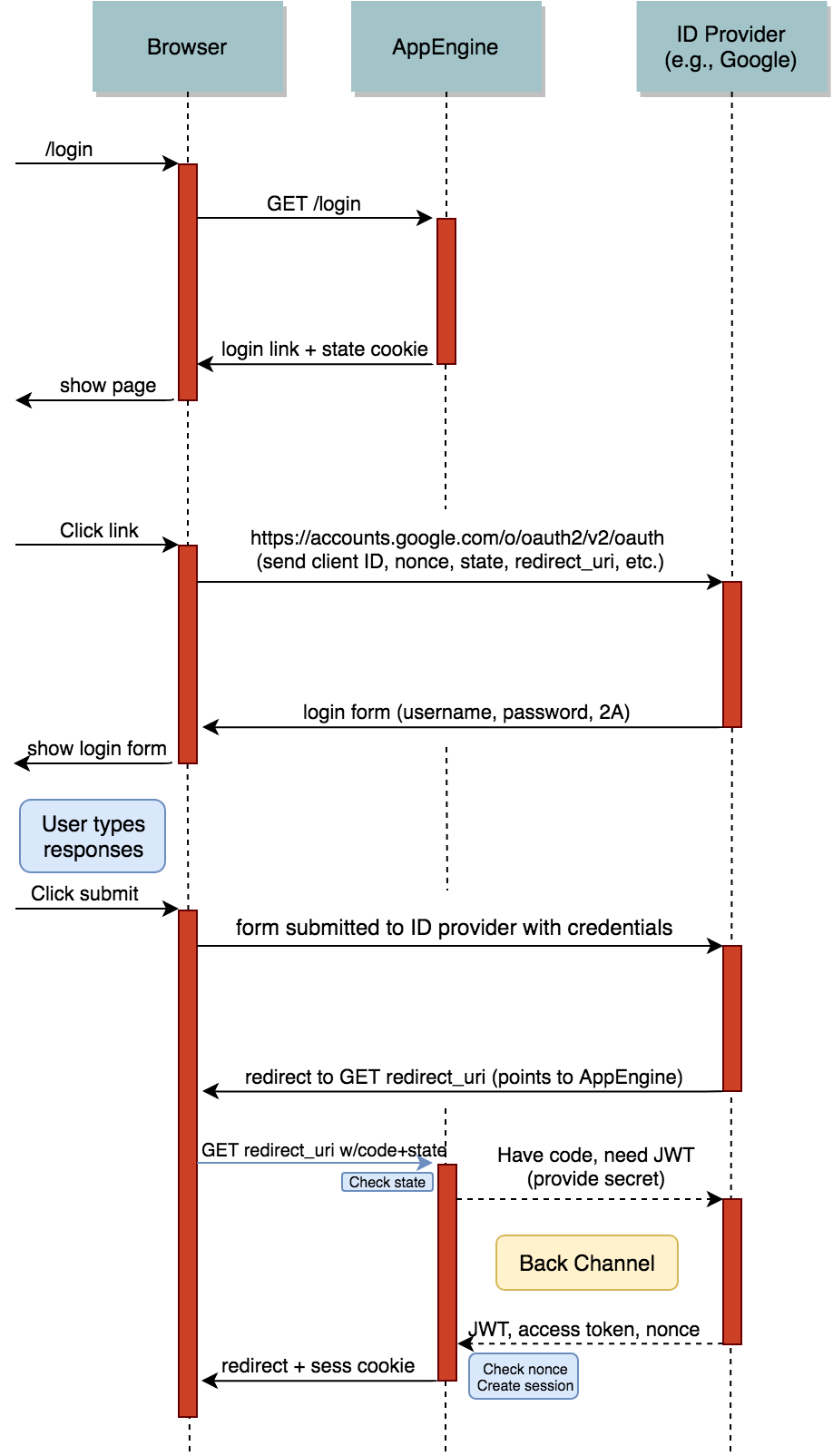
|  |
| --- |
| <a href="https://accounts.google.com/o/oauth2/v2/auth?response\_type=code&client\_id=myClientID&scope=openid%20email&state=abcd1234&nonce=1234&redirect\_uri=https%3A//yourserver.appspot.com/oidcauth" >Login with Google</a> |

There are numerous ways you can get that link into your login form. You can use a template system like Jinja, or you can use reqJSON from previous labs to get all of this data and create a request. You can even use a form with method “GET”, and set all of the parameters in hidden fields, only exposing the “submit” button.

Note that this example is using Google’s OpenID Connect service. In real life, you might use another service, and they will have what’s called a “[Discovery Document](https://developers.google.com/identity/protocols/OpenIDConnect#discovery)” that you can use to get the host and path of the URL for your link. In this case it was “<https://accounts.google.com/o/oauth2/v2/auth>”, but you might use something different.

That ***scheme://host/path*** part was obtained from Google’s discovery document, located here under the key “authorization\_endpoint”: <https://accounts.google.com/.well-known/openid-configuration>

So, what is that link all about, and how does the information flow around? Check out the following graphic, and see if it makes any sense to you:



This is the OIDC 3-legged auth flow. It goes something like this:

* Go to your site’s /login page
  + Request /login
  + Get back a login form with the link above inside of it. That link contains a bunch of information. Constant Client ID and redirect\_uri, and random state and nonce.
* Click the “login with google” link (or similar)
  + Request login page from Google (or other provider)
  + Get back a login form or something where you can affirmatively say “yes, use this account to authenticate”.
* Submit responses, get logged in.
  + Responses get submitted to e.g., Google, which does the heavy lifting of authenticating.
  + Google responds with a redirect that your browser intercepts and acts on immediately. That redirect goes *to your site, the URI you specified*.
  + The browser sends a request **to your site** with the information from Google.
    - Your site checks that the state matches what is expected, then
      * Sends a request to Google with the code it got, asking for an actual access token. Only here is the client secret used.
      * Google responds with the JWT (identity), access token, and nonce, which your site can check to ensure that it’s the expected nonce.
    - Your site creates and stores a session token for the user in the JWT.
  + Your site responds with a cookie containing the session token. Typically it will do this in a 302 redirect response that causes you to go to the main page of your site (because now you are logged in!)

## CSRF Protection Required

There are two kinds of protections that OpenID Connect requires us to implement, both related to CSRF, but not quite the same. The first is called the “state”, which is a plain old CSRF token. The second is the “nonce”, and it acts in much the same way, but for a different stage of the process, ensuring that front and back channel communication are part of the same login.

I used **str(uuid.uuid4())** to generate both of mine, then I used the double-cookie-submit approach, where I sent them down in cookies *and* as form values. The former is done using “set\_cookie”, and the latter is done either by template injection or JavaScript creating the link from cookie values. You can choose whichever you like. Yes, templatizing your form page is allowed for this purpose.

When your redirect\_uri is hit, you will check that the cookies match the values in the request, and that will help you be sure that you are really talking to the right person.

## Redirect URI

Your redirect\_uri is an endpoint on *your server* that the *browser* will contact after the user has logged in to whatever ID provider you are sending them to. I’m calling mine “/oidcauth” inside of my Python file. When I send it to the ID provider, it will need to be a complete URL, including host, etc. That means I’ll need to know *where I’m deployed* before I attempt to log someone in.

If you are on “yourserver.appspot.com”, that means the redirect\_uri needs to include that hostname, which is how we’ve been doing it above.

What does the handler (in Python) for the redirect\_uri do? It is basically a combination of registration and login all in one, but with a twist. Here are the steps it takes:

* Get code from request.args[‘code’]
* Get state from request.args[‘state’]
* Get nonce from your cookies (you should have set it in in a cookie when sending back the login form)
* Verify that “state” is what it should be (I’m comparing it to a cookie called “oidc\_state”, which I set when sending down the login form)
* Get the client secret from the datastore
* Create a URL that contains parameters for
  + code
  + client\_id
  + client\_secret
  + redirect\_uri
  + grant\_type = “authorization\_code”
* Send a POST request to the ID provider’s token endpoint (for Google that’s “<https://oauth2.googleapis.com/token>”, but check the [Discovery Document](https://accounts.google.com/.well-known/openid-configuration) to be sure, as it may change, and that’s why discovery documents exist). You can do this with the Python *requests* library, like this: ***requests.post(url, data)***, where the data is a dictionary of all of the parameters you are supposed to send on the back channel to Google.
* Get a response back that contains an “id\_token”, which is a JWT that will contain the user’s name and other information.
* Check that the nonce matches the one from the cookie.
* If that user doesn’t exist in the database, create it.
* Create a session token as normal, with this new username, set a cookie, and redirect.

And then you’re done!

It’s a lot less than it appears to be. But there are indeed some moving parts.

To reiterate, you can send a POST request using the requests library (if you are using Flask, this comes down with it as a dependency):

|  |
| --- |
| import requests  response = requests.post("https://oath2.googleapis.com/token",  {"code": code,  "client\_id": your\_client\_id,  "client\_secret": your\_client\_secret\_from\_the\_datastore,  "redirect\_uri": your\_redirect\_uri,  "grant\_type": "authorization\_code"}) |

## JWT Unpacking

I don’t know of any libraries *in particular* that you should use for unpacking the JWT, but you should know that traditional Base64 padding is **not** included in the JWT body. That means you need to add an appropriate number of padding bytes (=) before it will work, depending on its length. Here’s my code that unpacks the JWT (called “id\_token” here) to get the user information:

|  |
| --- |
| \_, body, \_ = id\_token.split('.') body += '=' \* (-len(body) % 4) claims = json.loads(base64.urlsafe\_b64decode(body.encode('utf-8'))) |

That should give you a Python dictionary that contains things like “sub” and “email” keys.

At that point, you can create a user. I personally use the “sub” as the user’s **key**, and “email” is stashed away in there in case I need it later. Note that even though you don’t use the user’s password hash anymore, you **still need the user object** because it will be a parent of all of the events stored for that user.

# Creating a Session

Once you have the claims as described above, you can get the user’s unique identifier from the “sub” field. I check that the user doesn’t exist, and if they don’t, I create them. Then I create a session that refers to that user’s sub (their “username” is the sub - a real username isn’t meaningful in this context, since you never need to interact with that directly - you just click the “Login with Google” button instead), create a cookie, and redirect to the home page. The user has now logged in using OpenID Connect!

# Grading

This lab will be graded on level of completion, at the discretion of the CA. Please ensure that your code is all your own. The purpose of this lab is not to create a client, but to create a client-creating *person*. That’s you. Write your own code, reach out early for help, and everything will be fine.

Remember, *be* ***curious***. If something isn’t working, ask yourself what it appears that you don’t know. If you have wondered about something and then dismissed it because you didn’t have time to pursue it, it could very well be that the gap in your knowledge is right there. Ask questions, get help from the class, and everything is going to work out.

1. This has to point to your service. You will be handling it. [↑](#footnote-ref-0)