Authentication and Authorization

Basic Introduction for Web Apps

Authorization in Django

Authenticate & Authorize

- **Authentication** - validate the identity of a "user", agent, or process
- **Authorization** - specifying rights to access a resource

Authentication is responsible for identifying who the user is.

Authorization is responsible for deciding what the user has permission to do.

Other Aspects of Security

- Access Control - how app controls access to resources
- Data Integrity - ability to prevent data from being modified, and *prove* that data hasn't been modified
- Confidentiality & Privacy - (privacy is about people, confidentiality is about data)
- Non-repudiation - ability to prove that user has made a request
  - "repudiate" means to deny doing something
- Auditing - make a pamper-resistant record of security related events

Authentication Methods

Authentication methods for humans:

1. Username & password

2. Username & one-time password (TOTP, codes, SMS)

3. Biometrics - fingerprint, facial recognition, iris scan

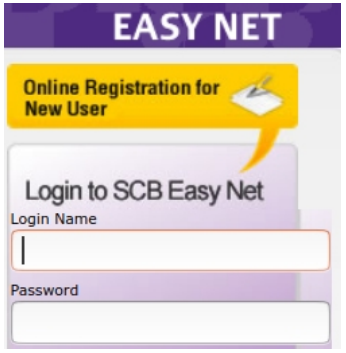
4. Trusted 3rd party - OAuth and OpenId
   "Login with Google" or "Login with Facebook"

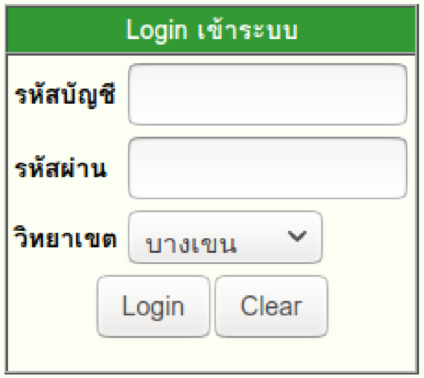
5. Public-private keys

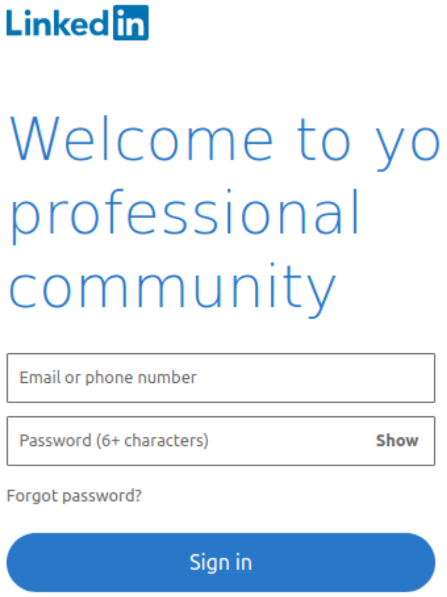
6. SQRL - a new method by Steve Gibson

Username & Password

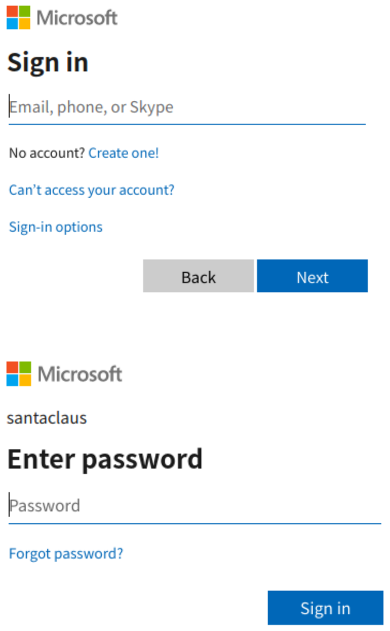
The oldest and one of the worst authentication methods.







Two page design



Username & Password

The oldest computer-based auth method.

Not very secure.

- passwords can be stolen
- passwords can be guessed
- vulnerable to man-in-the-middle & replay attack
- does not use the computational ability of user's device (it just sends a constant string)

Exercise: Have You Been Pwned?

Has your email address (and data) been stolen?

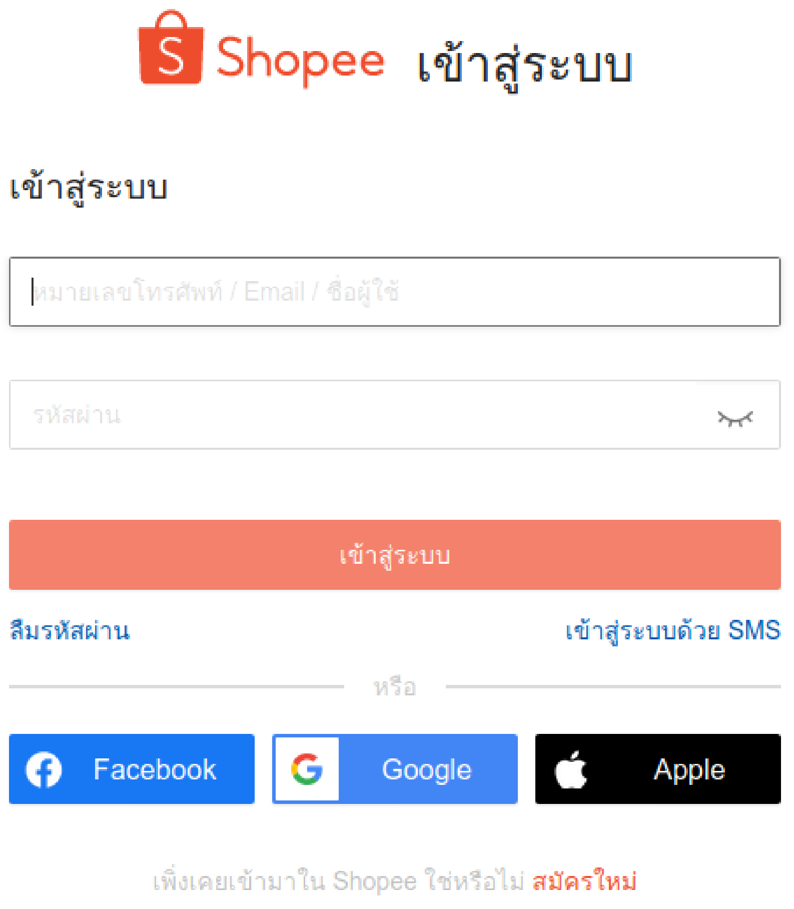
```
https://haveibeenpwned.com/
```

Has your password been seen in a breach?

```
https://haveibeenpwned.com/Passwords
```

OAuth & OpenID

Use a 3rd party to validate the user's identity


Shopee เข้าสู่ระบบ

เข้าสู่ระบบ

หมายเลขโทรศัพท์ / Email / ชื่อผู้ใช้

รหัสผ่าน

เข้าสู่ระบบ

ลืมรหัสผ่าน

เข้าสู่ระบบด้วย SMS

หรือ

Facebook

Google

Apple

เพิ่งเคยเข้ามาใน Shopee ใช่หรือไม่ สมัครใหม่

OAuth providers

OAuth 2.0

OAuth is *really* about granting access to resources.

But, as a side effect, you confirm your identity.

Example:

Google:

"*Grant shopee.co.th access to your name and email?*"

- tells shopee who you are, and proves that you can authenticate yourself to Google.

OAuth Use Cases or "Flows"

Server-side web app: The web app gets a "secret" that it uses when requesting access to a user's resources.

Browser-based app: Javascript code running in web browser. Cannot keep a secret, so the flow is different.

Mobile app: uses the mobile browser or native app as intermediary to grant OAuth access. Cannot keep a secret.

When you apply for <u>your app</u> to use OAuth on Google, etc, it is important to choose the correct "flow" or "grant type".

OAuth Exercise

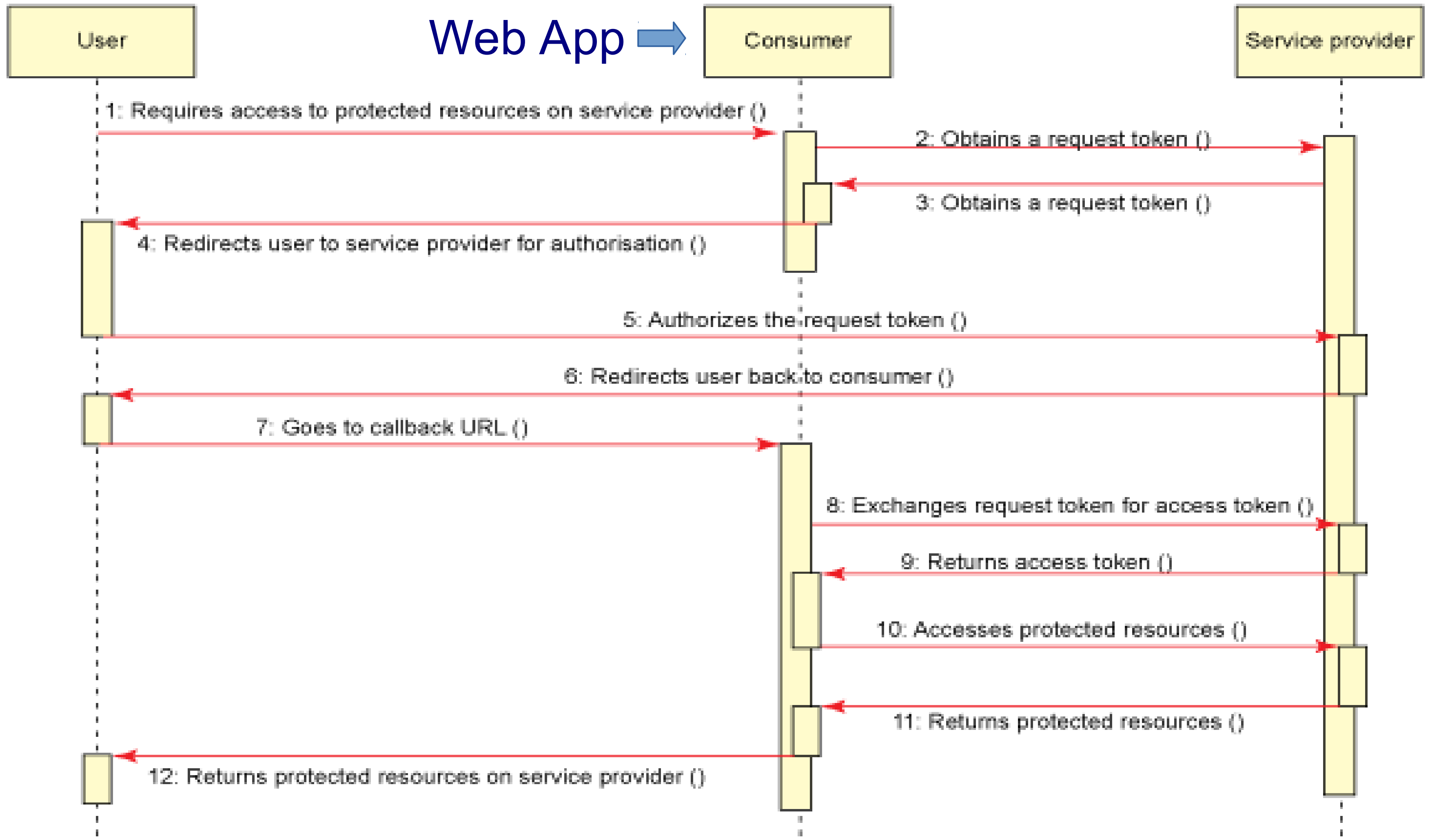
1. Login to Google. What apps or web sites have you granted access to your Google resources?

   Any that you don't use anymore? Delete them.

2. What apps or sites have been granted access to your Facebook account?

Software devs need to have ***good cybersecurity habits****.*
This exercise is part of that.

OAuth Flow for Server-Side Web Apps


User
Web App
Consumer
Service provider
1: Requires access to protected resources on service provider ()
2: Obtains a request token ()
3: Obtains a request token ()
4: Redirects user to service provider for authorisation ()
5: Authorizes the request token ()
6: Redirects user back to consumer ()
7: Goes to callback URL ()
8: Exchanges request token for access token ()
9: Returns access token ()
10: Accesses protected resources ()
11: Returns protected resources ()
12: Returns protected resources on service provider ()

This is the OAuth 1.0 flow, some names and parameters are different in OAuth 2.0

Step 0: Register your app

Go to the OAuth provider and request OAuth access.

"Register an application" using "Authorization Code" flow

give the server:

app. name and URL,

a callback URL,

requested scopes

server gives you:

- client_id
- client_secret
- authorization URL - where you send the user's browser

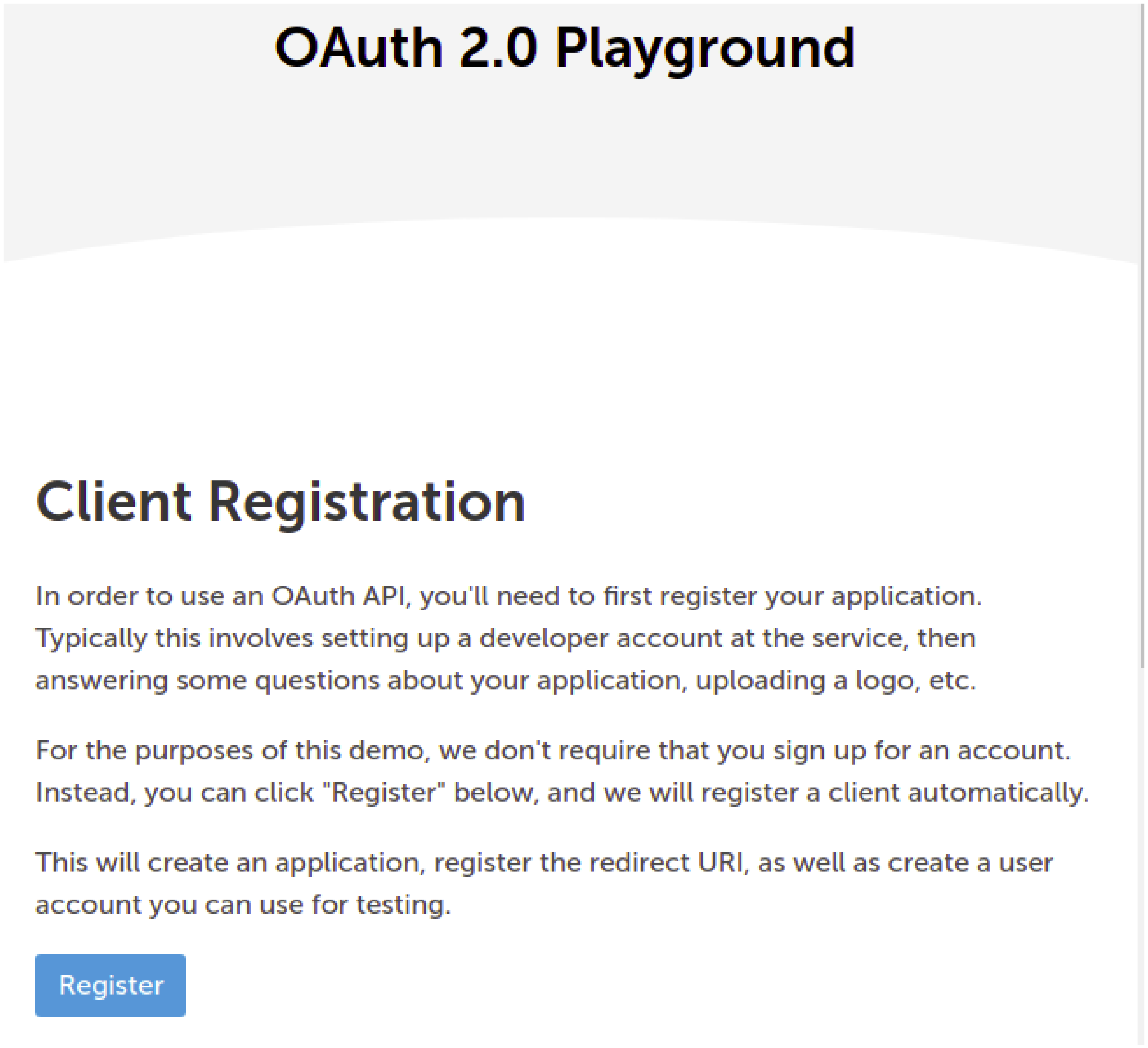
Exercise: OAuth Playground

Go to **`https://www.oauth.com/playground/`**

Choose "Authorization Code" flow.

Work through the exercise.

Note: you need to remember (or write down) the username and password the site gives you!



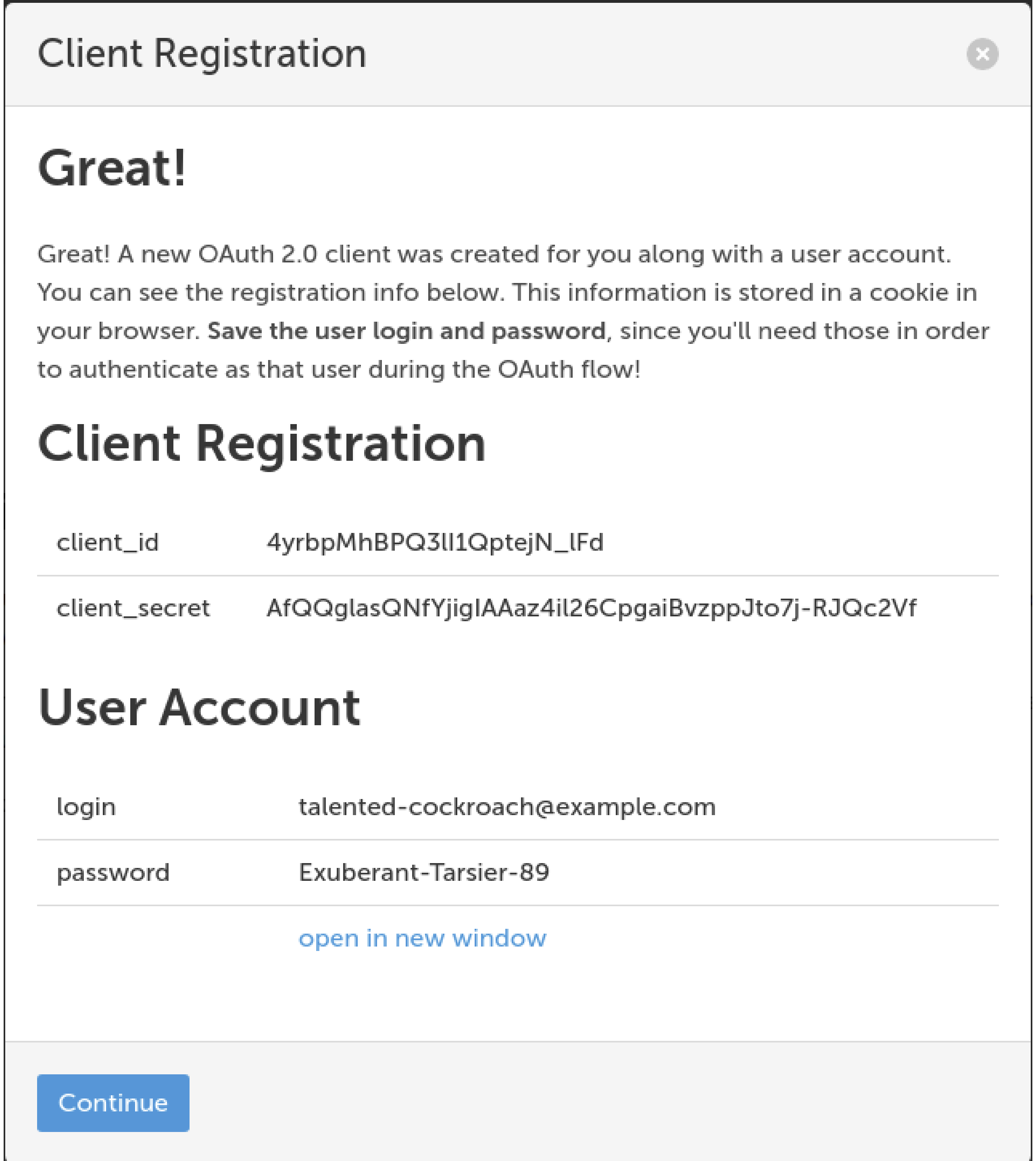
OAuth 2.0 Playground

Client Registration

In order to use an OAuth API, you'll need to first register your application. Typically this involves setting up a developer account at the service, then answering some questions about your application, uploading a logo, etc.

For the purposes of this demo, we don't require that you sign up for an account. Instead, you can click "Register" below, and we will register a client automatically.

This will create an application, register the redirect URI, as well as create a user account you can use for testing.



Client Registration

Great!

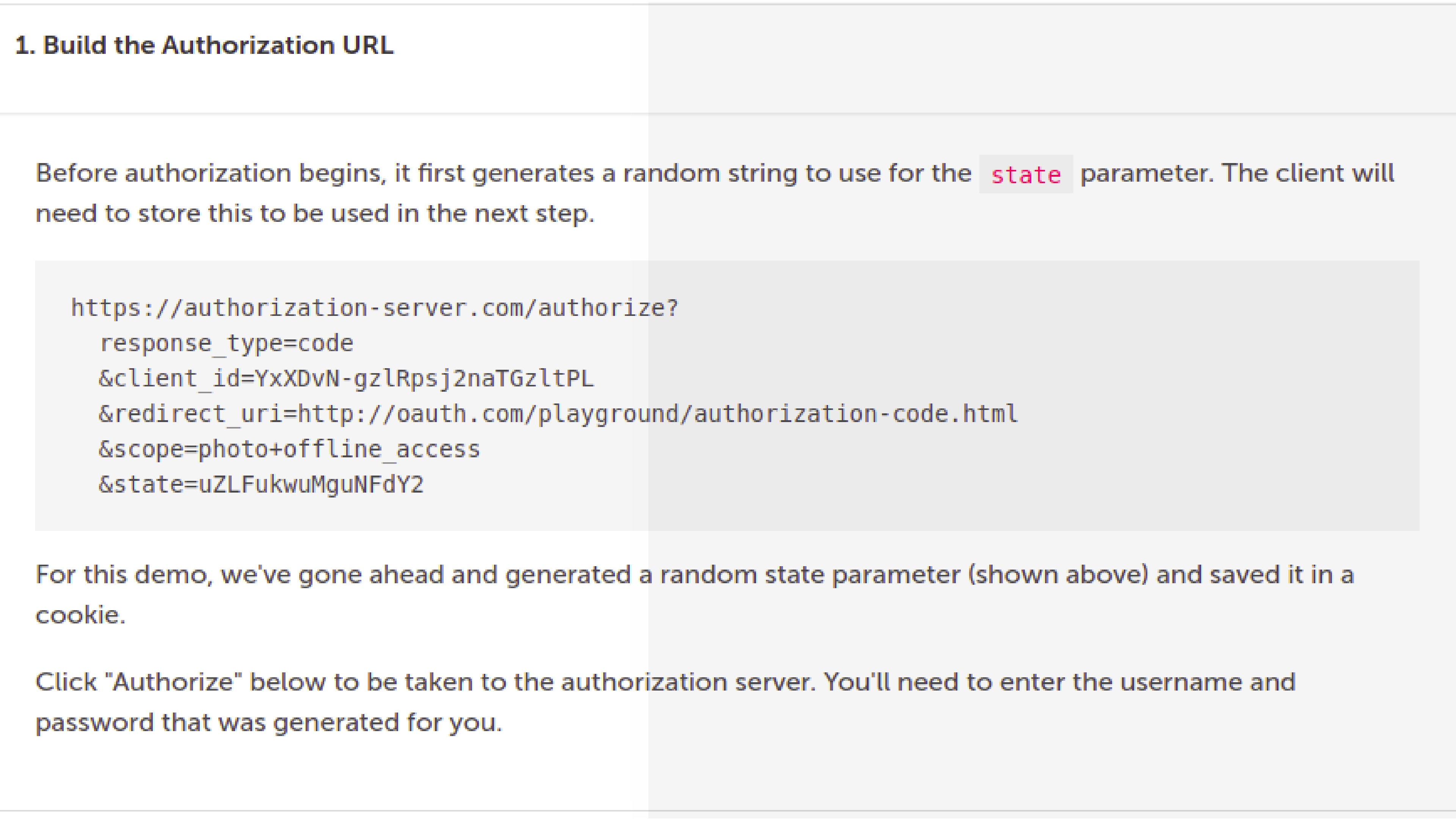
Great! A new OAuth 2.0 client was created for you along with a user account. You can see the registration info below. This information is stored in a cookie in your browser. **Save the user login and password**, since you'll need those in order to authenticate as that user during the OAuth flow!

Client Registration

client_id	4yrbpMhBPQ3lI1QptejN_lFd
client_secret	AfQQglasQNfYjigIAAaz4il26CpgaiBvzppJto7j-RJQc2Vf

User Account

login	talented-cockroach@example.com
password	Exuberant-Tarsier-89
	open in new window



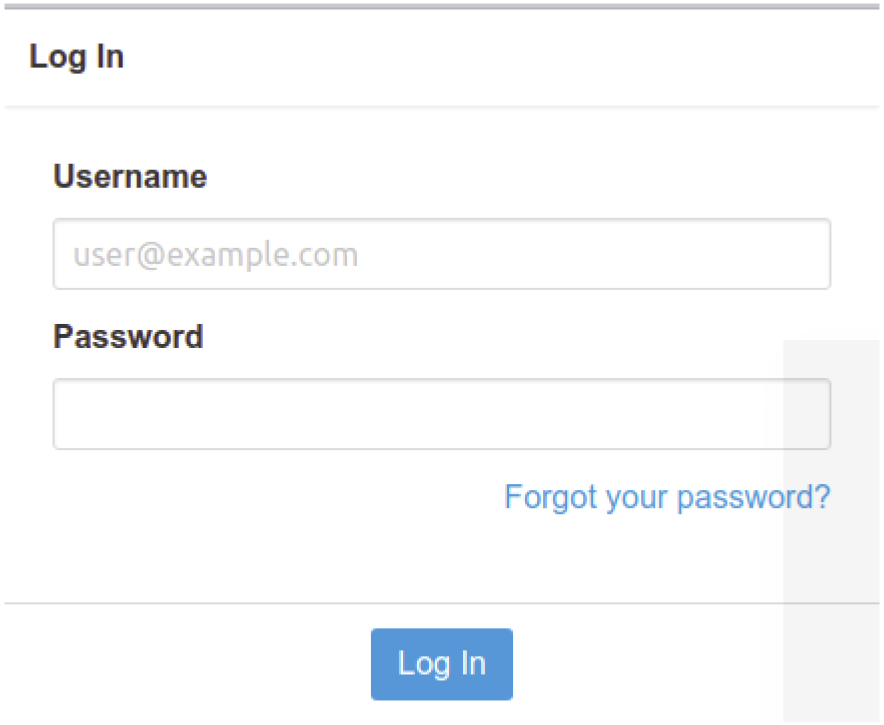
1. Build the Authorization URL

Before authorization begins, it first generates a random string to use for the `state` parameter. The client will need to store this to be used in the next step.

```
https://authorization-server.com/authorize?
  response_type=code
  &client_id=YxXDvN-gzlRpsj2naTGzltPL
  &redirect_uri=http://oauth.com/playground/authorization-code.html
  &scope=photo+offline_access
  &state=uZLFukwuMguNFdY2
```

For this demo, we've gone ahead and generated a random state parameter (shown above) and saved it in a cookie.

Click "Authorize" below to be taken to the authorization server. You'll need to enter the username and password that was generated for you.



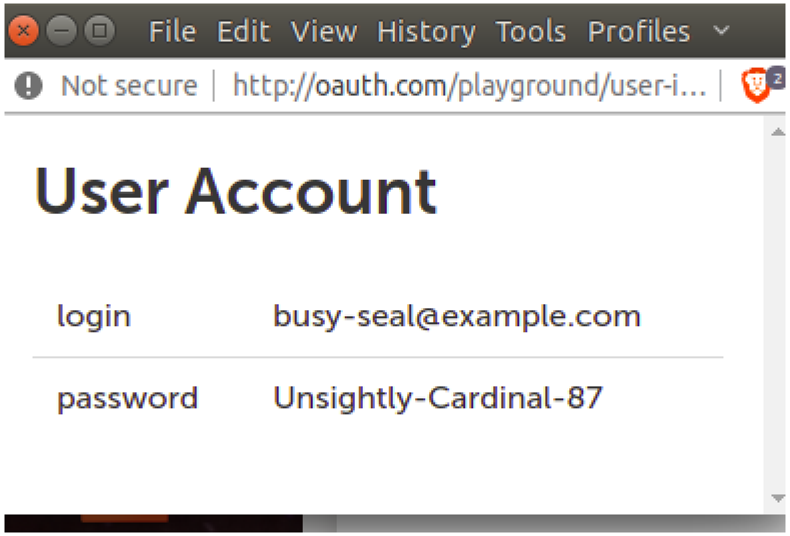
Log In

Username

user@example.com

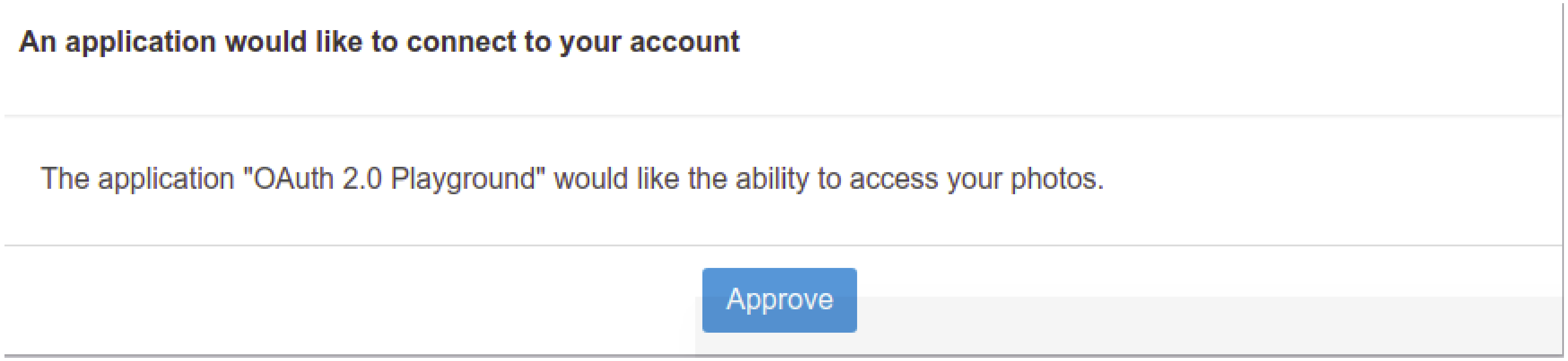
Password

Forgot your password?


File Edit View History Tools Profiles
Not secure
http://oauth.com/playground/user-i...

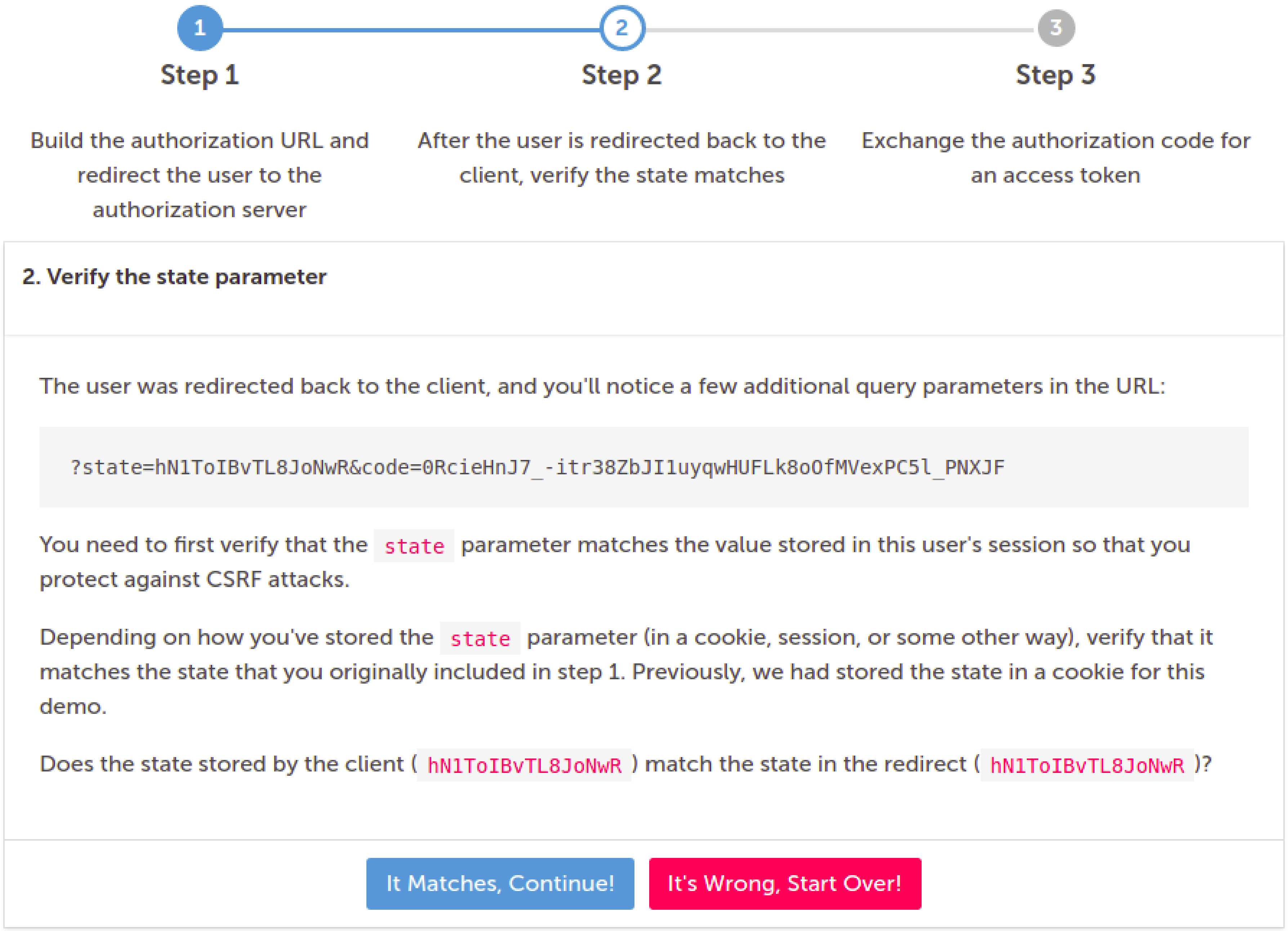
User Account

login	busy-seal@example.com
password	Unsightly-Cardinal-87



An application would like to connect to your account

The application "OAuth 2.0 Playground" would like the ability to access your photos.



Build the authorization URL and redirect the user to the authorization server

After the user is redirected back to the client, verify the state matches

Exchange the authorization code for an access token

2. Verify the state parameter

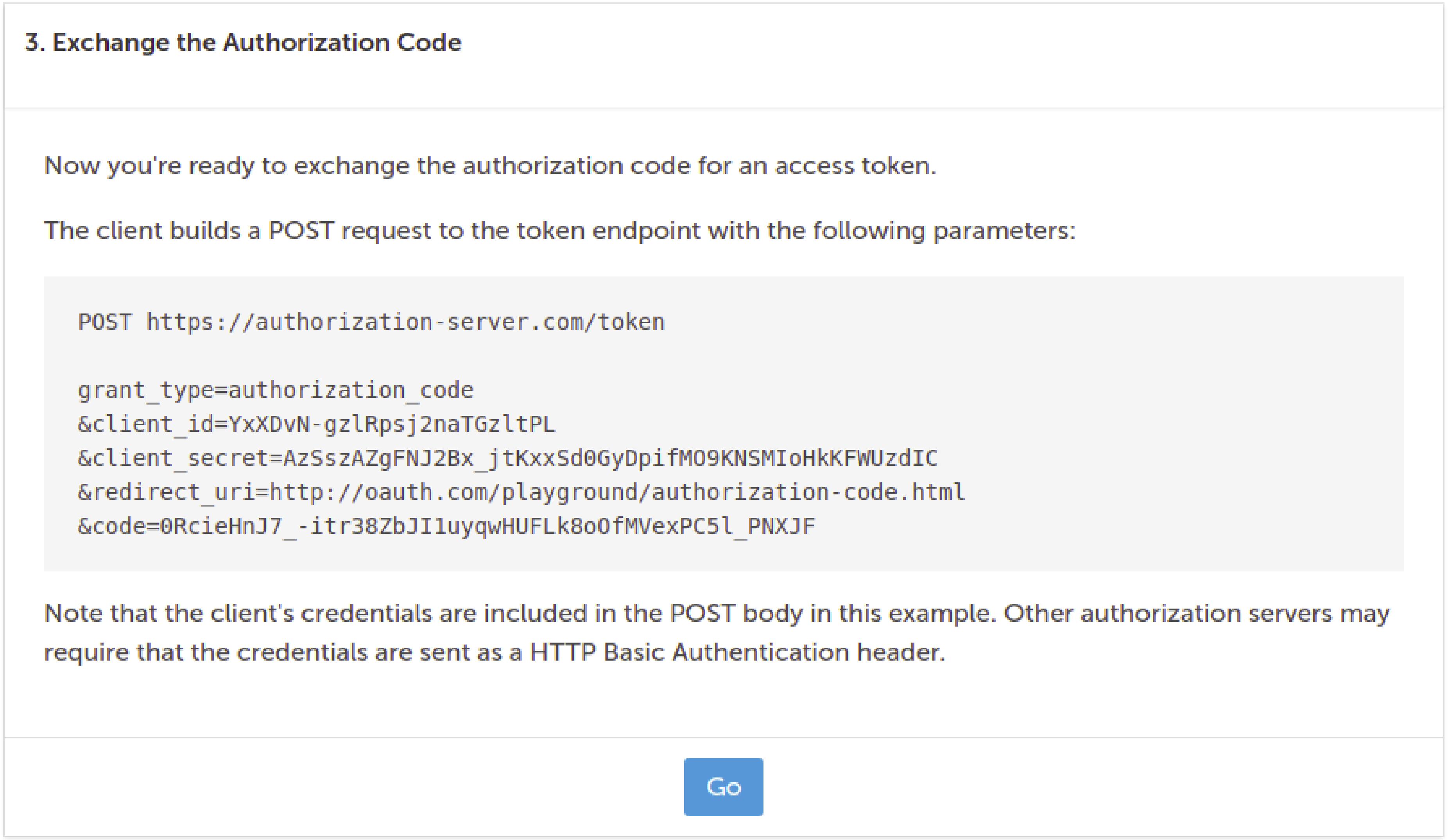
The user was redirected back to the client, and you'll notice a few additional query parameters in the URL:

```
?state=hN1ToIBvTL8JoNwR&code=0RcieHnJ7_-itr38ZbJI1uyqwHUFLk8oOfMVexPC5l_PNXJF
```

You need to first verify that the `state` parameter matches the value stored in this user's session so that you protect against CSRF attacks.

Depending on how you've stored the `state` parameter (in a cookie, session, or some other way), verify that it matches the state that you originally included in step 1. Previously, we had stored the state in a cookie for this demo.

Does the state stored by the client (`hN1ToIBvTL8JoNwR`) match the state in the redirect (`hN1ToIBvTL8JoNwR`)?



3. Exchange the Authorization Code

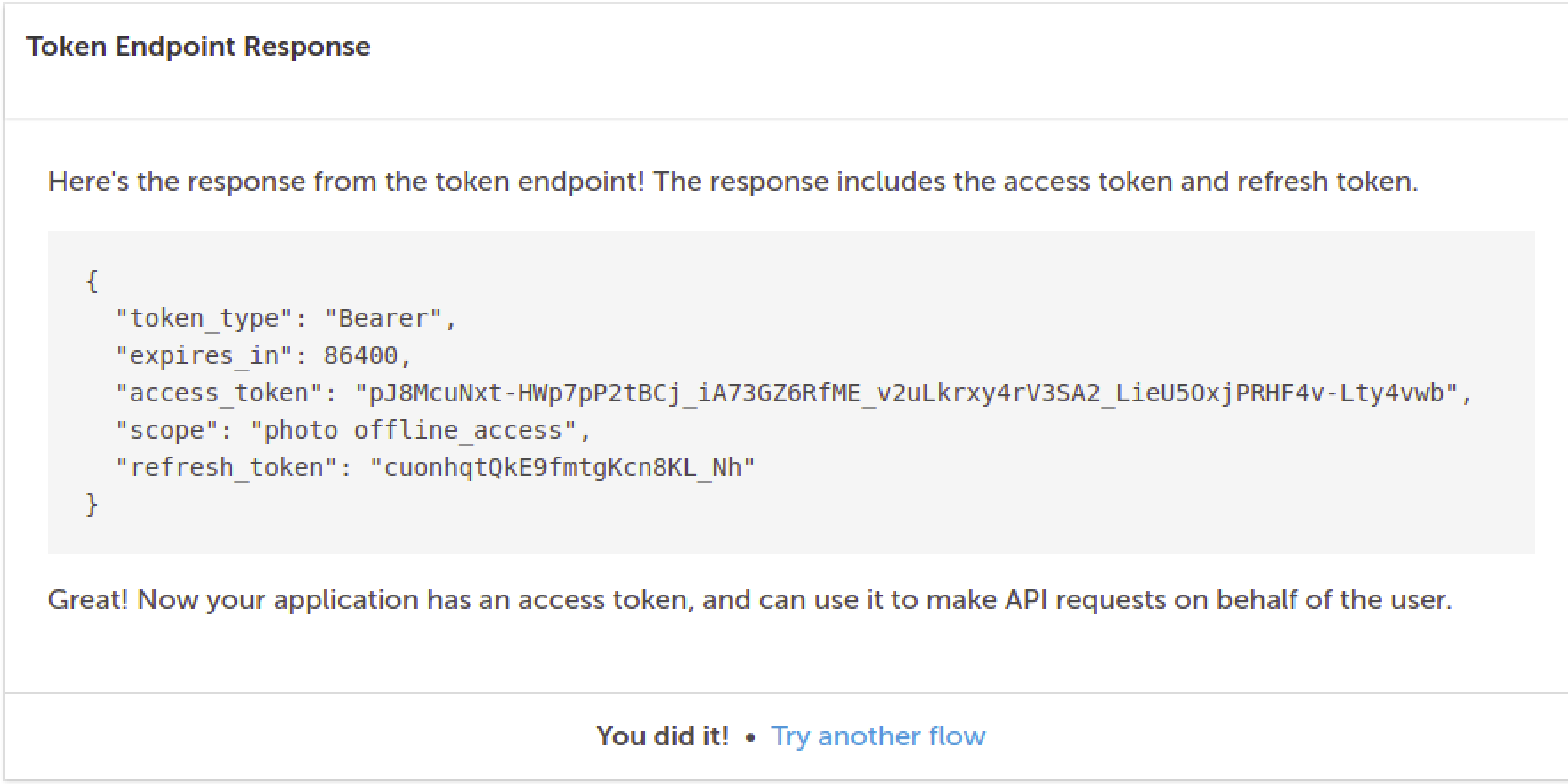
Now you're ready to exchange the authorization code for an access token.

The client builds a POST request to the token endpoint with the following parameters:

```
POST https://authorization-server.com/token

grant_type=authorization_code
&client_id=YxXDvN-gzlRpsj2naTGzltPL
&client_secret=AzSszAZgFNJ2Bx_jtKxxSd0GyDpifMO9KNSMIoHkKFWUzdIC
&redirect_uri=http://oauth.com/playground/authorization-code.html
&code=0RcieHnJ7_-itr38ZbJI1uyqwHUFLk8oOfMVexPC5l_PNXJF
```

Note that the client's credentials are included in the POST body in this example. Other authorization servers may require that the credentials are sent as a HTTP Basic Authentication header.



Token Endpoint Response

Here's the response from the token endpoint! The response includes the access token and refresh token.

```
{
  "token_type": "Bearer",
  "expires_in": 86400,
  "access_token": "pJ8McuNxt-HWp7pP2tBCj_iA73GZ6RfME_v2uLkrxy4rV3SA2_LieU5OxjPRHF4v-Lty4vwb",
  "scope": "photo offline_access",
  "refresh_token": "cuonhqtQkE9fmtgKcn8KL_Nh"
}
```

Great! Now your application has an access token, and can use it to make API requests on behalf of the user.

You did it! • Try another flow

What's Next?

OK, you can an authorization code or authorization token.

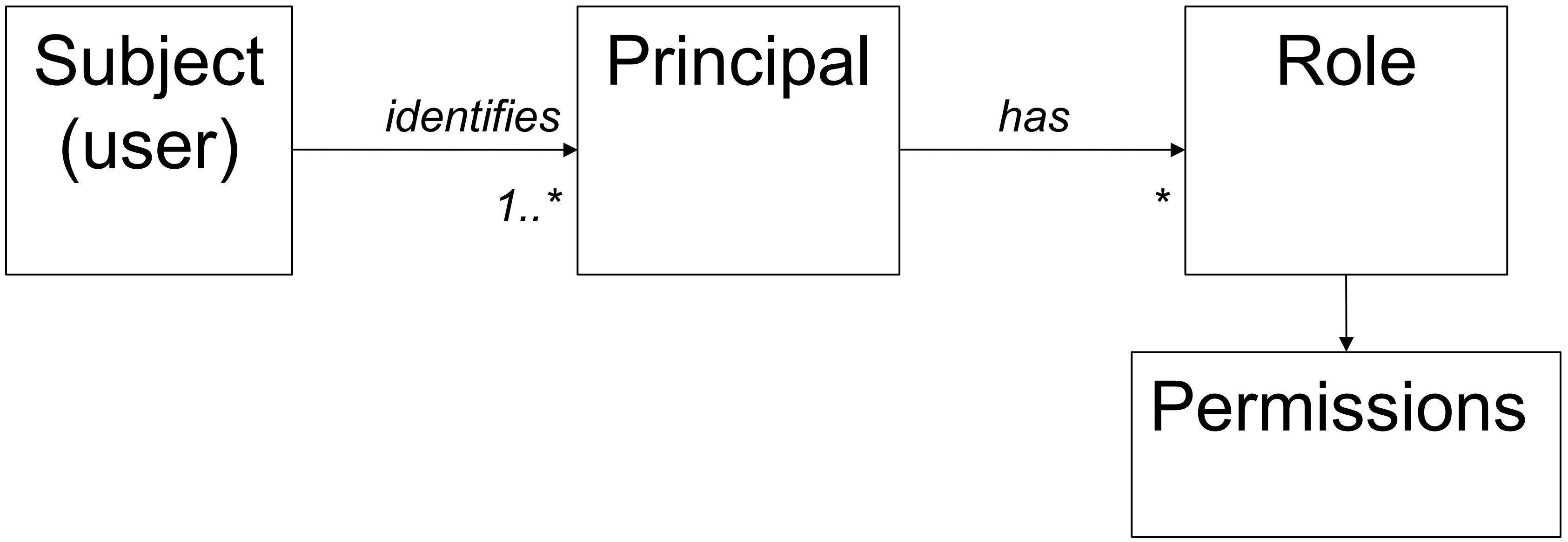
How do you use it?

Role Based Authorization

Permissions are based on the *roles* a user possesses.

A user may have many roles.

Example: “joe” has roles “voter” and “administrator”

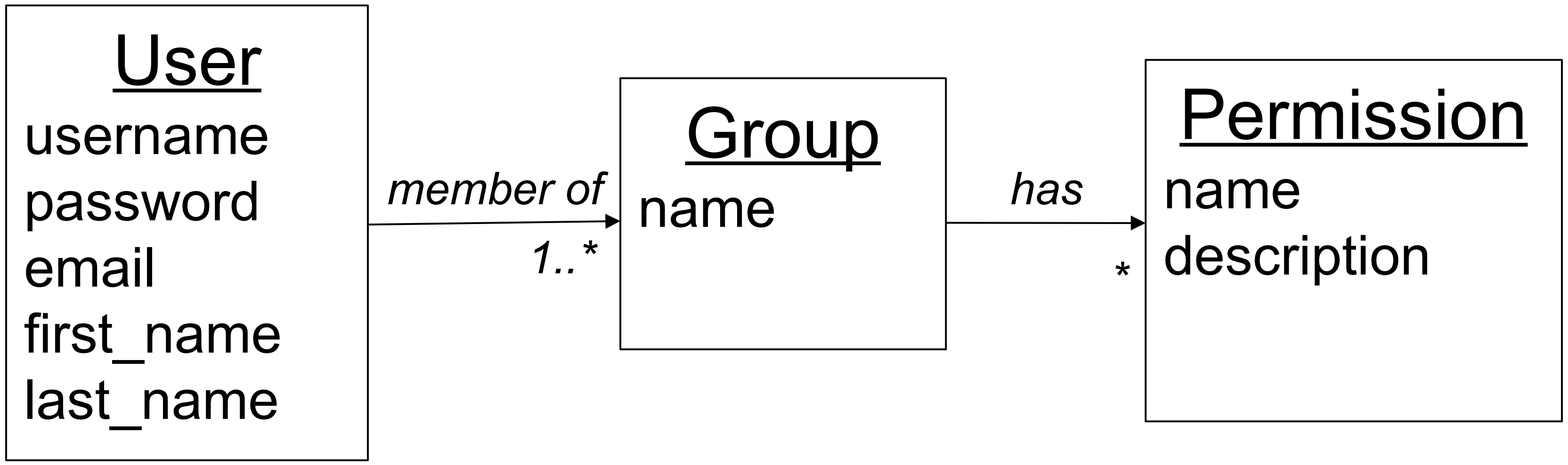


How Django Does It

User - identifies a user, authenticate using one of many *backends.*

Group - User is assigned one or more groups. Each group has some Permissions.

Permission - key-value pair (anything you like) used in code to enforce *authorization*



Checking Authorization in Code

```
from django.contrib.auth
           import authenticate, login

def dumb_login_view(request):
  # authenticate first
  user = authenticate(request, "hacker", "Hack!")
  login(request, user)

  if user.is_authenticated:
     # allow any logged in user to do something

  if user.has_perm('blog.can_create'):
     # allow user to create a blog entry
```

Checking Auth in Views

The `request` object has reference to current user.

```
def blog_index(request):
    if not request.user.is_authenticated:
        return redirect('login')

    if request.user.has_perm(
             'blog.can_post_comment')
        # include form for posting comments
```

Use Decorators on Views

Decorators reduce risk of errors, create cleaner code

```
from django.contrib.auth.decorators
   import login_required, permission_required

@login_required
def blog_index(request):
    """show index of todos for this user"""

@permission_required('blog.can_create')
def add(request):
    """post a new blog entry"""
```

Decorators in urls.py

You can add decorators in urls.py. I think using decorators in views is more readable & avoids errors.

```
urlpatterns = [

   path('blog/', login_required(views.index)),
   ...
```

Define Your Own Decorators

If none of Django's decorators do what you want...

https://docs.djangoproject.com/en/3.0/topics/auth/default/

```
def kasetsart_email(user) -> bool:
    return user.email.endswith('@ku.ac.th')

@user_passes_test( kasetsart_email )
def vote(request, question_id):
    # only users at KU can vote
```

Mixins for Class-based Views

"Mixin" means to combine or "mix in" behavior from several different classes.

```
from django.contrib.auth.mixins
              import LoginRequiredMixin

class PollsIndex(LoginRequiredMixin, ListView):
    template_name = 'polls/index.html'
    ...
```

Authorization Checks in Templates

Templates can use the **`user`** and **`perms`** objects.

```
{% if user.is_authenticated %}
    Hello, {{ user.username }}
{% else %}
    Please <a href="{% url 'login'%}">Login</a>
{% endif %}

{# same as user.has_perm('blog.post_entry') #}
{% if perms.blog.post_entry %}
   You can post a blog entry
{% endif %}
```

Where to Apply Authorization?

1. In templates. Gives web page the desired appearance and page flow, but can be by-passed. Don't rely on it.

2. In views. Requests are always passed to a view, so this is fairly secure. Prefer decorators or Mixins instead of checks in code.

3. In models? In some frameworks, you can configure required permissions directly into model classes. Apparently not in Django.

4. In url mapping (urls.py).