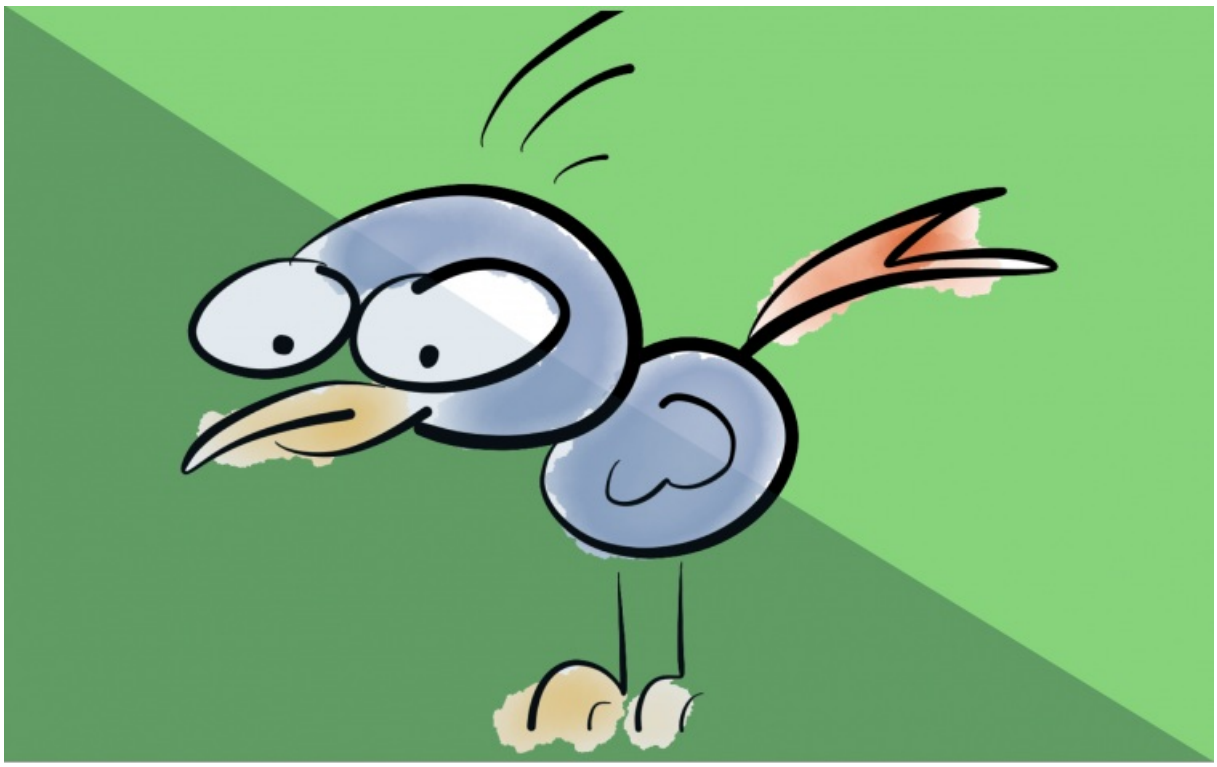
OAuth2 in 8 Steps



With <3 from SymfonyCasts

Chapter 1: Serious OAuth in 8 Steps

Hey guys and gals! In this tutorial, we're going to get serious with OAuth by building an app with some complex and real-life features, like Facebook authentication, dealing with refresh tokens and more. We'll need about 8 steps to turn a barebones starting app into a complex, OAuth machine:

1. Client Credentials: making API requests for our own account
2. Authorization Code: Getting a token for another user's account
3. Logging in via OAuth
4. OAuth with Facebook
5. OAuth in JavaScript with Google+
6. Handling Expired Tokens
7. Using Refresh Tokens
8. Tightening up Security

As we go through these, we'll give you any theory and background you need.

Tiny Crash Course in OAuth

For now, you just need to understand that OAuth is an Authorization Framework. In human-speak, it means that it defines the different ways two parties, like your cool web site and a *user* on your website, can exchange tokens securely. Each of these ways is known as a grant type and though they look different, each grant type will always deliver an access token.

OAuth Token

So what's this token? It's just a unique string tied to my account that gives *you* access to make API requests on my behalf. It's like a username and password all rolled into one. For example, if ABCD1234 is a valid token to my Facebook account, then an HTTP request that looks like this would post to my timeline:

```
POST /weaverryan/feed HTTP/1.1
Host: graph.facebook.com
Content-Type: application/x-www-form-urlencoded
Content-Length: length

access_token=ABCD1234&message=Hello
```

Exactly *how* you pass the access token in an API request is different between Facebook, Twitter, or any other API. But it's always there.

I *could* just give you my username and password, but a token is much better. If I give 10 apps access to my account, each app will have its own token, which means I can revoke access to some apps, but not others.

Tokens can have a limited scope, which is *huge*. Unlike a password which gives you access to do *anything* on my account, I can give you a token that lets you view my Facebook friends, but not post to my wall.

So OAuth is really just a big set of rules that describe how two parties can exchange tokens. If I create a website where I want to access my users' Facebook friends, exactly how does a user *give* me an access token?

Let's answer that question, along with the thrilling topic of token expiration, the hopeful story of refresh tokens, the inspirational tale of single-sign on and all kinds of other things.

Let's go!

Chapter 2: Client Credentials

Meet Brent. He's the hardworking, beard-growing, kale-munching type who has a coop of the nicest, smartest, and best egg-laying chickens this side o' the Mississippi! But feeding his chickens and doing other things around the farm has always taken a lot of time.

But great news! The brand new "Chicken Oversight Operations Platform", or COOP site has just launched! With COOP, you can login to the site and collect your chicken eggs, unlock the barn, and do all kinds of other things just by clicking a button.

Noticing that COOP has an API, Brent wonders if he could write a little script that would collect his eggs automatically. Yea, if he had a script that made an API request on his behalf, he could run it on a CRON job daily and sleep in!

So COOP is real, sort of. You can find this make-believe website by going to http://coop.apps.knpuniversity.com. Go ahead and create an account, and start controlling your virtual farm. It's the future!

Starting our Command-line Script

COOP's API is simple, with just a few endpoints, including the one we want for our little command-line script: eggs-collect.

I've already made a cron/ directory with a script called collect_eggs.php that'll get us started:

```
// collect_eggs.php
include __DIR__.'/vendor/autoload.php';
use Guzzle\Http\Client;

// create our http client (Guzzle)
$http = new Client('http://coop.apps.knpuniversity.com', array(
    'request.options' => array(
        'exceptions' => false,
    )
));
```

Tip

Code along with us! Click the Download button on this page to get the starting point of the project, and follow the readme to get things setup.

It doesn't do anything except create a Client object that's pointing at the COOP website. Since we'll need to make HTTP requests to the COOP API, we'll use a really nice PHP library called Guzzle. Don't worry if you've never used it, it's really easy.

Before we start, we need to use Composer to download Guzzle.

> This tutorial uses an old (version 3) version of Guzzle! It doesn't affect the tutorial, but if you decide to install it manually, be sure to install guzzle/guzzle.

Download Composer into the cron/ directory and then install the vendor libraries:

```
$ php composer.phar install
```

Tip

New to Composer? Do yourself a favor and master it for free: The Wonderful World of Composer.

Let's try making our first API request to /api/2/eggs-collect. The 2 is our COOP user ID, since we want to collect eggs from *our* farm. Your number will be different:

```
// collect_eggs.php
// ...

$request = $http->post('/api/2/eggs-collect');
$response = $request->send();
echo $response->getBody();

echo "\n\n";
```

Try it by executing the script from the command line:

```
$ php collect_eggs.php
```

Not surprisingly, this blows up!

```
{
  "error": "access_denied",
  "error_description": "an access token is required"
}
```

OAuth Applications

But before we think about getting a token, we need to create an application on COOP (http://coop.apps.knpuniversity.com/api - click "Your Applications" and then "Create your Application"). The application represents the external app or website that we want to build. In our case, it's the little command-line script. In OAuth-speak, it's this application that will actually ask for access to a user's COOP account.

Give it a name like "Brent's Lazy CRON Job", a description, and check only the box for "Collect Eggs from Your Chickens". These are "scopes", or basically the permissions that your app will have if a token is granted from COOP.

When we finish, we now have a Client ID and an auto-generated "Client Secret". These are a sort of username and password for the application. One tricky thing is that the terms "application" and "client" are used interchangeably in OAuth. And both are used to refer to the application we just registered and the actual app you're building, like the CRON script or your website. I'll try to clarify along the way.

Now, let's get an access token!

Client Credentials Grant Type

The first OAuth grant type is called Client Credentials, which is the simplest of all the types. It involves only two parties, the client and the server. For us, this is our command-line script and the COOP API.

Using this grant type, there is no "user", and the access token we get will only let us access resources under the control of the application. When we make API requests using this access token, it's almost like we're logging in as the *application* itself, not any individual user. I'll explain more in a second.

If you visit the application you created earlier, you'll see a nice "Generate a Token" link that when clicked will fetch one. Behind the scenes, this uses client credentials, which we'll see more closely in a second.

```
http://coop.apps.knpuniversity.com/token
    ?client_id=Your+Client+Name
    &client_secret=abcdefg
    &grant_type=client_credentials
```

But for now, we can celebrate by using this token immediately to take actions on behalf of the application!

Access Tokens in the API

Exactly how to do this depends on the API you're making requests to. One common method, and the one COOP uses, is to send it via an Authorization Bearer header.

```
GET /api/barn-unlock HTTP/1.1
Host: coop.apps.knpuniversity.com
Authorization: Bearer ACCESSTOKENHERE
```

Update the script to send this header:

```
// collect-eggs.php
// ...

$accessToken = 'abcd1234def67890';

$request = $http->post('/api/2/eggs-collect');
$request->addHeader('Authorization', 'Bearer '.$accessToken);
$response = $request->send();
echo $response->getBody();

echo "\n\n";
```

When we run the script again, start celebrating, because it works! And now we have enough eggs to make an omlette :)

```
{
  "action": "eggs-collect",
  "success": true,
  "message": "Hey look at that, 3 eggs have been collected!",
  "data": 3
}
```

Trying to Collect Someone Else's Eggs

Notice that this collects the eggs for *our* user becase we're including our user ID in the URL. What happens if we change id to be for a different user?

```
/api/3/eggs-collect
```

If you try it, it fails!

```
{
  "error": "access_denied",
  "error_description": "You do not have access to take this action"
}
```

Technically, with a token from client credentials, we're making API requests not on behalf of a user, but on behalf of an application. This makes client credentials perfect for making API calls that edit or get information about the application itself, like a count of how many users it has.

We decided to build COOP so that the application *also* has access to modify the user that created the application. That's why we *are* able to collect our user's eggs, but not our neighbor's.

Getting the Token via Client Credentials

Put the champagne away: we're not done yet. Typically, access tokens don't last forever. COOP tokens last for 24 hours, which means that tomorrow, our script will break.

Letting the website do the client-credentials work for us was nice for testing, but we need to do it ourselves inside the script. Every OAuth server has an API endpoint used to request access tokens. If we look at the COOP API Authentication docs, we can see the URL and the POST parameters it needs:

```
http://coop.apps.knpuniversity.com/token

Parameters:
    client_id
    client_secret
    grant_type
```

Let's update our script to first make *this* API request. Fill in the client_id, client_secret and grant_type POST parameters:

```
// collect-eggs.php
// ...

// run this code *before* requesting the eggs-collect endpoint
$request = $http->post('/token', null, array(
    'client_id'     => 'Brent\'s Lazy CRON Job',
    'client_secret' => 'a2e7f02def711095f83f2fb04ecbc0d3',
    'grant_type'    => 'client_credentials',
));

// make a request to the token url
$response = $request->send();
$responseBody = $response->getBody(true);
var_dump($responseBody);die;
// ...
```

With any luck, when you run it, you should see a JSON response with an access token and a few other details:

```
{
  "access_token": "fa3b4e29d8df9900816547b8e53f87034893d84c",
  "expires_in": 86400,
  "token_type": "Bearer",
  "scope": "chickens-feed"
}
```

Let's use *this* access token instead of the one we pasted in there:

```
// collect-eggs.php
// ...

// step1: request an access token
$request = $http->post('/token', null, array(
    'client_id'     => 'Brent\'s Lazy CRON Job',
    'client_secret' => 'a2e7f02def711095f83f2fb04ecbc0d3',
    'grant_type'    => 'client_credentials',
));

// make a request to the token url
$response = $request->send();
$responseBody = $response->getBody(true);
$responseArr = json_decode($responseBody, true);
$accessToken = $responseArr['access_token'];

// step2: use the token to make an API request
$request = $http->post('/api/2/eggs-collect');
$request->addHeader('Authorization', 'Bearer '.$accessToken);
$response = $request->send();
echo $response->getBody();

echo "\n\n";
```

Now, it still works *and* since we're getting a fresh token each time, we'll never have an expiration problem. Once Brent sets up a CRON job to run our script, he'll be sleeping in 'til noon!

Why, What and When: Client Credentials

Every grant type eventually uses the /token endpoint to get a token, but the details before that differ. Client Credentials is *a way* to get a token directly. One limitation is that it requires your client secret, which is ok now because our script is hidden away on some server.

But on the web, we won't be able to expose the client secret. And that's where the next two grant types become important.

Chapter 3: Authorization Code Grant Type

Suddenly, Brent is jolted awake at noon to the sound of farmer Scott driving his eggs to the market and screaming:

> Haha, Brent! My chickens lay way more eggs than yours!

But in reality, Brent *knows* that his chickens are way better egg-making hens than Scott's... but how to prove it?

Then it hits him! The COOP API has an endpoint to see how many eggs have been collected from a user's farm each day. Brent decides to create a new website that will use this endpoint to count how many total eggs a COOP user's farm has collected. He'll call it: Top Cluck! Fantasy Chicken League, or FCL for short. To call the /api/eggs-count endpoint on behalf of each user, the site will use OAuth to collect an access token for every farmer that signs up.

Once again, the question is: how can each user give FCL an access token that allows it to count eggs on their behalf?

Starting up FCL

Let's check out the FCL app, which Brent has already started building. It lives in the client/ directory of the code download. I'll use the built-in PHP web server to run this site:

```
$ cd client/web
$ php -S localhost:9000
```

Tip

Code along with us! Click the Download link on this page to get the starting point of the project.

That command starts a built-in PHP webserver, and it'll just sit right there until we're ready to turn it off. This project also uses Composer, so let's copy the composer.phar file we used earlier into this directory and use install to download some outside libraries the project uses.

Tip

If this doesn't work and PHP simply shows you its command-line options, check your PHP version. The built-in web server requires PHP 5.4 or higher.

Put the URL to the site into your browser and load it up. Welcome to Top Cluck! We already have a leaderboard and basic registration. Go ahead and create an account, which automatically logs us in.

The site is fully-functional, with database tables ready to keep track of how many eggs each farmer has collected. The only missing piece is OAuth: getting the access token for each user so that we can make an API request to count their eggs.

Redirecting to Authorize

Before TopCluck can make an API request to COOP to count my eggs, I need to authorize it. On the homepage, there's already an Authorize link, which just prints a message right now.

The code behind this URL lives in the src/OAuth2Demo/Client/Controllers/CoopOAuthController.php file. You don't even need to understand how this works, just know that whatever we do here, shows up:

```
// src/OAuth2Demo/Client/Controllers/CoopOAuthController.php
// ...

public function redirectToAuthorization(Request $request)
{
    die('Hallo world!');
}
```

The first step of the authorization code grant type is to redirect the user to a specific URL on COOP. From here the user will authorize our app. According to COOP's API Authentication page, we need to redirect the user to /authorize and send several query parameters.

In our code, let's start building the URL:

```
// src/OAuth2Demo/Client/Controllers/CoopOAuthController.php
// ...

public function redirectToAuthorization(Request $request)
{
    $url = 'http://coop.apps.knpuniversity.com/authorize?'.http_build_query(array(
        'response_type' => 'code',
        'client_id' => '?',
        'redirect_uri' => '?',
        'scope' => 'eggs-count profile'
    ));

    var_dump($url);die;
}
```

The response_type type is code because we're using the Authorization Code flow. The other valid value is token, which is for a grant type called implicit flow. We'll see that later.

For scopes, we're using profile and eggs-count so that once we're authorized, we can get some profile data about the COOP user and, of course, count their eggs.

To get a client_id, let's go to COOP and create a new application that represents TopCluck. The most important thing is to check the 2 boxes for profile information and collecting the egg count. I'll show you why in a second.

Tip

If there is already an application with the name you want, just choose something different and use that as your client_id.

Copy the client_id into our URL. Great! The last piece is the redirect_uri, which is a URL on our site that COOP will send the user to after granting or denying our application access. We're going to do all kinds of important things once that happens.

Let's set that URL to be /coop/oauth/handle, which is just another page that's printing a message. The code for this is right inside the same file, a little further down:

```
// src/OAuth2Demo/Client/Controllers/CoopOAuthController.php
// ...

public function receiveAuthorizationCode(Application $app, Request $request)
{
    // equivalent to $_GET['code']
    $code = $request->get('code');

    die('Implement this in CoopOAuthController::receiveAuthorizationCode()');
}
```

Instead of hardcoding the URL, I'll use the URL generator that's part of Silex:

```
public function redirectToAuthorization(Request $request)
{
    $redirectUrl = $this->generateUrl('coop_authorize_redirect', array(), true);

    $url = 'http://coop.apps.knpuniversity.com/authorize?'.http_build_query(array(
        'response_type' => 'code',
        'client_id' => 'TopCluck',
        'redirect_uri' => $redirectUrl,
        'scope' => 'eggs-count profile'
    ));
    // ...
}
```

However you make your URL, just make sure it's absolute. Ok, we've built our authorize URL to COOP, let's redirect the user

to it:

```
public function redirectToAuthorization(Request $request)
{
    // ...

    return $this->redirect($url);
}
```

That redirect() function is special to my app, so your code may differ. As long as you somehow redirect the user, you're good.

Tip

Since we're using Silex, the redirect() function is actually a shortcut I created to create a new RedirectResponse object.

Authorizing on COOP

Let's try it! Go back to the homepage and click the "Authorize" link. This takes us to our code, which then redirects us to COOP. We're already logged in, so it gets straight to asking us to authorize the app. Notice that the scopes that we included in the URL are clearly communicated. Let's authorize the app. Later, we'll see what happens if you don't.

When we click the authorization button, we're sent back to the redirect_uri on TopCluck! Nothing has really happened yet. COOP didn't set any cookies or anything else. But the URL *does* include a code query parameter.

Exchanging the Authorization Code for an Access Token

This query parameter is called the authorization code, and it's unique to this grant type. It's not an access token, which is really what we want, but it's the key to getting that. The authorization code is our temporary proof that the user said that our application can have an access token.

Let's start by copying the code from the collect_eggs.php script and pasting it here. Go ahead and change the client_id and client_secret to be from the new client or application we created for TopCluck:

```
// src/OAuth2Demo/Client/Controllers/CoopOAuthController.php
// ...

public function receiveAuthorizationCode(Application $app, Request $request)
{
    // equivalent to $_GET['code']
    $code = $request->get('code');

    $http = new Client('http://coop.apps.knpuniversity.com', array(
        'request.options' => array(
            'exceptions' => false,
        )
    ));

    $request = $http->post('/token', null, array(
        'client_id'     => 'TopCluck',
        'client_secret' => '2e2dfd645da38940b1ff694733cc6be6',
        'grant_type'    => 'authorization_code',
    ));

    // make a request to the token url
    $response = $request->send();
    $responseBody = $response->getBody(true);
    var_dump($responseBody);die;
}
```

If we look back at the COOP API Authentication docs, we'll see that /token has 2 other parameters that are used with the authorization grant type: code and redirect_uri. I'm already retrieving the code query parameter, so let's fill these in. Make sure to also change the grant_type to authorization_code like it describes in the docs. Finally, dump the $responseBody to see if this request works:

```
public function receiveAuthorizationCode(Application $app, Request $request)
{
    // equivalent to $_GET['code']
    $code = $request->get('code');
    // ...

    $request = $http->post('/token', null, array(
        'client_id'     => 'TopCluck',
        'client_secret' => '2e2dfd645da38940b1ff694733cc6be6',
        'grant_type'    => 'authorization_code',
        'code'          => $code,
        'redirect_uri'  => $this->generateUrl('coop_authorize_redirect', array(), true),
    ));

    // ...
}
```

The key to this flow is the code parameter. When COOP receives our request, it will check that the authorization code is valid. It also knows which user the code belongs to, so the access token it returns will let us make API requets on behalf of *that* user.

But what about the redirect_uri? This parameter is absolutely necessary for the API request to work, but isn't actually used by COOP. It's a security measure, and it *must* exactly equal the original redirect_uri that we used when we redirected the user.

Ok, let's try it! When we refresh, the API actually gives us an error:

```
{
    "error": "invalid_grant",
    "error_description": "The authorization code has expired"
}
```

The authorization code has a very short lifetime, typically measured in seconds. We normally exchange it immediately for an access token, so that's ok! Let's start the whole process from the homepage again.

Tip

Usually, an OAuth server will remember that a user already authorized an app and immediately redirect the user back to your app. COOP doesn't do this only to make things easier to understand.

This time, the API request to /token returns an access_token. Woot! Let's also set expires_in to a variable, which is the number of seconds until this access token expires:

```
public function receiveAuthorizationCode(Application $app, Request $request)
{
    // ...

    $request = $http->post('/token', null, array(
        'client_id'     => 'TopCluck',
        'client_secret' => '2e2dfd645da38940b1ff694733cc6be6',
        'grant_type'    => 'authorization_code',
        'code'          => $code,
        'redirect_uri'  => $this->generateUrl('coop_authorize_redirect', array(), true),
    ));

    // make a request to the token url
    $response = $request->send();
    $responseBody = $response->getBody(true);
    $responseArr = json_decode($responseBody, true);

    $accessToken = $responseArr['access_token'];
    $expiresIn = $responseArr['expires_in'];
}
```

Using the Access Token

Just like in our CRON script, let's use the access token to make an API request! One of the endpoints is /api/me, which returns information about the user that is tied to the access token. Let's make a GET request to this endpoint, setting the

access token on the Authorization header, just like we did before:

```
public function receiveAuthorizationCode(Application $app, Request $request)
{
    // ...

    $accessToken = $responseArr['access_token'];
    $expiresIn = $responseArr['expires_in'];

    $request = $http->get('/api/me');
    $request->addHeader('Authorization', 'Bearer '.$accessToken);
    $response = $request->send();
    echo ($response->getBody(true));die;
}
```

Try it by going back to the homepage and clicking "Authorize". Simply refreshing the page won't work here, as the authorization code will have already expired. With any luck, you'll see a JSON response with information about the user:

```
{
    "id": 2,
    "email": "brent@knpuniversity.com",
    "firstName": "Brent",
    "lastName": "Shaffer"
}
```

This works of course because we're sending an access token that is tied to Brent's account. This also works because when we redirect the user, we're asking for the profile scope.

And with that, we've seen the key parts of the authorization code grant type and how to use an access token in our application. But where should we store the token and what if the user denies our application access? We'll look at these next.

Chapter 4: Authorization Code: Saving the Token & Handling Failures

What if we want to make other API requests on behalf of Brent later? Where should we store the access token?

Saving the Access Token Somewhere

Some access tokens last an hour or two, and are well suited for storing in the session. Others are long-term tokens, for example facebook provides a 60-day token, and these make more sense to store in a database. Either way, storing the token will free us from having to ask the user to authorize again.

In our app, we're going to store it in the database:

```
// src/OAuth2Demo/Client/Controllers/CoopOAuthController.php

public function receiveAuthorizationCode(Application $app, Request $request)
{
    // ...
    $meData = json_decode($response->getBody(), true);

    $user = $this->getLoggedInUser();
    $user->coopAccessToken = $accessToken;
    $user->coopUserId = $meData['id'];
    $this->saveUser($user);

    // ...
}
```

This code is specific to my app, but the end result is that I've updated the coopAccessToken column on the user table for the currently-authenticated user. I'm also saving the coopUserId, which we'll need since most API calls have the user's ID in the URI.

Recording the Expires Time

We can also store the time when the token will expire. I'll create a DateTime object that represents the expiration time. We can check this later before trying to make API requests. If the token is expired, we'll need to send the user through the authorization process again:

```
public function receiveAuthorizationCode(Application $app, Request $request)
{
    // ...
    $expiresIn = $responseArr['expires_in'];
    $expiresAt = new \DateTime('+'.$expiresIn.' seconds');
    // ...

    $user = $this->getLoggedInUser();
    $user->coopAccessToken = $accessToken;
    $user->coopUserId = $meData['id'];
    $user->coopAccessExpiresAt = $expiresAt;
    $this->saveUser($user);

    // ...
}
```

Again, the code here is special to my app, but the end result is just to update a column in the database for the current user. When we try it, it runs and hits our die statement. But if you go to the homepage, the user drop-down shows us that the COOP user id was saved! Eggcellent...

When Authorization Fails

But what if the user declines to authorize our app? If this happens, an OAuth server will redirect the user back to our

redirect_uri. If we start from the homepage again but deny access on COOP, we can see this. But this time, the page explodes because our request to /token is *not* returning an access token. In fact, COOP hasn't included a code query parameter in the URL on the redirect.

This is what a canceled authorization looks like: no authorization code.

Unfortunately, we can't just assume that the user authorized our application. As we've seen when this happens, the code query parameter will be missing, but the OAuth server should include a few extra query parameters explaining what went wrong. These are commonly called error and error_description. Let's grab these and pass them into a template I've already prepared:

```
public function receiveAuthorizationCode(Application $app, Request $request)
{
    // equivalent to $_GET['code']
    $code = $request->get('code');

    if (!$code) {
        $error = $request->get('error');
        $errorDescription = $request->get('error_description');

        return $this->render('failed_authorization.twig', array(
            'response' => array(
                'error' => $error,
                'error_description' => $errorDescription
            )
        ));
    }

    // ...
}
```

When we try the flow again, we see a nicer message. You can really do whatever you want in your application, just make sure you're handling the possibility that the user will decline your app's request.

These errors should be documented by the OAuth server, but the standard set includes "temporarily_unavailable", "server_error", and "access_denied".

When Fetching the Access Token Fails

There's one other spot where things can fail: when requesting out to /token. What if the response doesn't have an access_token field? Under normal circumstances, this really shouldn't happen, but let's render a different error template in case it does. Don't worry about the variables I'm passing into the template, I'm just trying to pass enough information so that we can see what the problem was:

```
public function receiveAuthorizationCode(Application $app, Request $request)
{
    // ...
    $request = $http->post('/token', null, array(
        // ...
    ));

    $response = $request->send();
    $responseBody = $response->getBody(true);
    $responseArr = json_decode($responseBody, true);

    // if there is no access_token, we have a problem!!!
    if (!isset($responseArr['access_token'])) {
        return $this->render('failed_token_request.twig', array(
            'response' => $responseArr ? $responseArr : $response
        ));
    }
    // ...
}
```

Try the whole cycle again, but approve the app this time. It works the first time of course. But if you refresh, you'll see this error in action. The code parameter exists, but it's expired. So, the request to /token fails.

Redirecting after Success

Until now, we've had an ugly die statement at the bottom of the code that handles the OAuth redirect. What you'll actually want to do here is redirect to some other page. Our work is done for now, so we want to help the user to continue on our site:

```
public function receiveAuthorizationCode(Application $app, Request $request)
{
    // ...

    // redirect back to the homepage
    return $this->redirect($this->generateUrl('home'));
}
```

In our application, this code simply redirects us to the homepage. And just like that, we're done! This is the authorization grant type, which has 2 distinct steps to it:

1. First, redirect the user to the OAuth server using its /authorize endpoint, your application's client_id, a redirect_uri and the scopes you want permission for. The URL and how the parameters look may be different on other OAuth servers, but the idea will be the same.

2. After authorizing our app, the OAuth server redirects back to a URL on our site with a code query parameter. We can use this, along with our client_id and client_secret to make an API request to the /token endpoint. Now, we have an access token.

Let's finally use it to count some eggs!

Couting Eggs

On the homepage, we still have the "Authorize" button. But now that we have an access token for the user, we really don't need this anymore. The template that displays this page is at views/dashboard.twig, and I'm already passing a user variable here, which is the currently-authenticated user object. Let's hide the "Authorize" link if the user has a coopUserId stored in the database:

```
{# views/dashboard.twig #}
{# ... #}

{% if user.coopUserId %}

{% else %}
    <a class="btn btn-primary btn-lg" href="{{ path('coop_authorize_start') }}">Authorize</a>
{% endif %}
```

If we *do* have a coopUserId, let's add a link the user can click that will count their daily eggs. Don't worry if you're not familiar with the code here, we're just generating a URL to a new page that I've already setup:

```
{# views/dashboard.twig #}
{# ... #}

{% if user.coopUserId %}
    <a class="btn btn-primary btn-lg" href="{{ path('count_eggs') }}">Count Eggs</a>
{% else %}
    <a class="btn btn-primary btn-lg" href="{{ path('coop_authorize_start') }}">Authorize</a>
{% endif %}
```

When we refresh, we see the new link. Clicking it gives us another todo message. Open up src/OAuth2Demo/Client/Controllers/CountEggs.php, which is the code behind this new page.

Making the eggs-count API Request

Start by copying the /api/me code from CoopOAuthController, and changing the method from get() to post(), since the eggs-count endpoint requires POST:

```
// src/OAuth2Demo/Client/Controllers/CountEggs.php
// ...

class CountEggs extends BaseController
{
    // ...
    public function countEggs()
    {
        $http = new Client('http://coop.apps.knpuniversity.com', array(
            'request.options' => array(
                'exceptions' => false,
            )
        ));

        $request = $http->post('/api/me');
        $request->addHeader('Authorization', 'Bearer '.$accessToken);
        $response = $request->send();
        $meData = json_decode($response->getBody(), true);

        die('Implement this in CountEggs::countEggs');

        return $this->redirect($this->generateUrl('home'));
    }
}
```

The endpoint we want to hit now is /api/USER_ID/eggs-count. Fortunately, we've already saved the COOP user id and access token for the currently logged-in user to the database. Get that data by using our app's $this->getLoggedInUser() method and update the URL:

```
public function countEggs()
{
    $user = $this->getLoggedInUser();

    $http = new Client('http://coop.apps.knpuniversity.com', array(
        'request.options' => array(
            'exceptions' => false,
        )
    ));

    $request = $http->post('/api/'.$user->coopUserId.'/eggs-count');
    $request->addHeader('Authorization', 'Bearer '.$user->coopAccessToken);
    // ...
}
```

I'll add in some debug code so we can see if this is working:

```
public function countEggs()
{
    // ...

    $request = $http->post('/api/'.$user->coopUserId.'/eggs-count');
    $request->addHeader('Authorization', 'Bearer '.$user->coopAccessToken);
    $response = $request->send();
    echo ($response->getBody(true));die;
    // ...
}
```

When we refresh, you should see a nice JSON response. Yea, we're counting eggs! That'll show Farmer Scott!

Since the purpose of TopCluck is to keep track of how many eggs each farmer has collected each day, let's save the new count to the database. Like before, I've already done all the hard work, so that we can focus on just the OAuth pieces. Just call setTodaysEggCountForUser() and pass it the current user and the egg count. While we're here, we can remove the die statement and redirect the user back to the homepage once we're done:

```
public function countEggs()
{
    // ...

    $response = $request->send();
    $countEggsData = json_decode($response->getBody(), true);

    $eggCount = $countEggsData['data'];
    $this->setTodaysEggCountForUser($this->getLoggedInUser(), $eggCount);

    return $this->redirect($this->generateUrl('home'));
}
```

When we refresh, we should get redirected back to the homepage. But on the right, Farmer Brent's egg count isn't going up. Let's go to COOP and collect a few more eggs manually. Back on FCL, if we count our eggs again, we get the updated count. Sweet!

All the Things that can Go Wrong

The "Count Eggs" page we created works great, but we're not handling any of the things that might go wrong. First, we're hiding its link, but what if a user somehow ends up on the page without a coopUserId or coopAccessToken? Let's code for this case:

```
public function countEggs()
{
    $user = $this->getLoggedInUser();

    if (!$user->coopAccessToken || !$user->coopUserId) {
        throw new \Exception('Somehow you got here, but without a valid COOP access token! Re-authorize!');
    }

    // ...
}
```

I'm throwing an exception message, but we could also handle this differently, like by redirecting the user to the "Authorize" page to start the OAuth flow.

Another thing we can check for is whether or not the token has expired. This is possible because we stored the expiration data in the database. I've created an easy helper method to check for this. If this happens, let's redirect the user to re-authorize, just like if they had clicked the "Authorize" link:

```
public function countEggs()
{
    $user = $this->getLoggedInUser();

    if (!$user->coopAccessToken || !$user->coopUserId) {
        throw new \Exception('Somehow you got here, but without a valid COOP access token! Re-authorize!');
    }

    if ($user->hasCoopAccessTokenExpired()) {
        return $this->redirect($this->generateUrl('coop_authorize_start'));
    }

    // ...
}
```

Finally, what if the API request itself fails? A simple way to handle this might look like this:

```
public function countEggs()
{
    // ...

    $request = $http->post('/api/'.$user->coopUserId.'/eggs-count');
    $request->addHeader('Authorization', 'Bearer '.$user->coopAccessToken);
    $response = $request->send();

    if ($response->isError()) {
        throw new \Exception($response->getBody(true));
    }

    // ...
}
```

Of course, you may want to do something more sophisticated. The response could also have some error information on it, which you can play around with. For OAuth, this is important because the call *may* have failed because the access_token expired. What, I thought we just checked for that? Well, in the real world, there's no guarantee that the token won't expire before its scheduled time. Plus, the user may have decided to revoke your token -- what a bully. Be aware, and handle accordingly. Once again, the OAuth Server should provide information on the error in the error and error_description query string parameters.

You're now dangerous, so lets move on to let our farmers actualy log into FCL via COOP.

Chapter 5: User Login with OAuth

Now that it's possible for users to authorize TopCluck to count their COOP eggs, Brent's on his way to showing farmer Scott just whose eggs rule the roost.

Feeling fancy, he wants to make life even easier by letting users skip registration and just login via COOP. Afterall, every farmer who uses the site will already have a COOP account.

Since we've done all the authorization code work already, adding "Login with COOP" or "Login with Facebook" buttons is really easy.

Creating New TopCluck Users

Start back in CoopOAuthController.php, where we handled the exchange of the authorization code for the access token. Right now, this assumes that the user is already logged in and updates their account with the COOP details:

```
// src/OAuth2Demo/Client/Controllers/CoopOAuthController.php
// ...
public function receiveAuthorizationCode(Application $app, Request $request)
{
    // ...
    $meData = json_decode($response->getBody(), true);

    $user = $this->getLoggedInUser();
    $user->coopAccessToken = $accessToken;
    $user->coopUserId = $meData['id'];
    $this->saveUser($user);
    // ...
}
```

But instead, let's actively allow anonymous users to go through the authorization process. And when they do, let's create a *new* user in our database:

```
public function receiveAuthorizationCode(Application $app, Request $request)
{
    // ...

    $meData = json_decode($response->getBody(), true);

    if ($this->isUserLoggedIn()) {
        $user = $this->getLoggedInUser();
    } else {
        $user = $this->createUser(
            $meData['email'],
            // a blank password - this user hasn't created a password yet!
            '',
            $meData['firstName'],
            $meData['lastName']
        );
    }
    $user->coopAccessToken = $accessToken;
    $user->coopUserId = $meData['id'];
    $user->coopAccessExpiresAt = $expiresAt;
    $this->saveUser($user);
    // ...
}
```

Some of these functions are specific to my app, but it's simple: if the user isn't logged in, create and insert a new user record using the data from the /api/me endpoint.

Choosing a Password

Notice I'm giving the new user a blank password. Does that mean someone could login as the user by entering a blank

password? That would be a huge security hole!

The problem is that the user isn't choosing a password. In fact, they're opt'ing to *not* have one and to use their COOP account instead. So one way or another, it should *not* be possible to login to this account using *any* password. Normally, my passwords are encoded before being saved, like all passwords should be. You can't see it here, but when the password is set to a blank string, I'm skipping the encoding process and actually setting the password in the database to be blank. If someone *does* try to login using a blank password, it'll be encoded first and won't match what's in the database.

As long as you find some way to prevent anyone from logging in as the user via a password, you're in good shape! You could also have the user choose a password right now or have an area to do that in their profile. I'll mention the first approach in a second.

Finally, let's log the user into this new account:

```
public function receiveAuthorizationCode(Application $app, Request $request)
{
    // ...

    if ($this->isUserLoggedIn()) {
        $user = $this->getLoggedInUser();
    } else {
        $user = $this->createUser(
            $meData['email'],
            // a blank password - this user hasn't created a password yet!
            '',
            $meData['firstName'],
            $meData['lastName']
        );

        $this->loginUser($user);
    }

    // ...
}
```

We still need to handle a few edge-cases, but this creates the user, logs them in, and then still updates them with the COOP details.

Adding the Login with COOP Link

Let's try it out! Log out and then head over to the login page. Here, we'll add a "Login with COOP" link. The template that renders this page is at views/user/login.twig:

```
{# views/user/login.twig #}

<div class="form-group">
    <div class="col-lg-10 col-lg-offset-2">
        <button type="submit" class="btn btn-primary">Login!</button>
        OR
        <a href="{{ path('coop_authorize_start') }}"
            class="btn btn-default">Login with COOP</a>
    </div>
</div>
```

The URL for the link is the same as the "Authorize" button on the homepage. If you're already logged in, we'll just update your account. But if you're not, we'll create a new account and log you in. It's that simple!

Let's also completely reset the database, which you can do just by deleting the data/topcluck.sqlite file inside the client/ directory:

```
$ $ rm data/topcluck.sqlite
```

When we try it out, we're redirected to COOP, sent back to TopCluck, and are suddenly logged in. If we look at our user details, we can see we're logged in as Brent, with COOP User ID 2.

Handling Existing Users

There's one big hole in our logic. If I logout and go through the process again, it blows up! This time, it tries to create a *second* new user for Brent instead of using the one from before. Let's fix that. For organization, I'm going to create a new private function called findOrCreateUser() in this same class. If we can find a user with this COOP User ID, then we can just log the user into that account. If not, we'll keep creating a new one:

```
public function receiveAuthorizationCode(Application $app, Request $request)
{
    // ...

    if ($this->isUserLoggedIn()) {
        $user = $this->getLoggedInUser();
    } else {
        $user = $this->findOrCreateUser($meData);

        $this->loginUser($user);
    }

    // ...
}

private function findOrCreateUser(array $meData)
{
    if ($user = $this->findUserByCOOPId($meData['id'])) {
        // this is an existing user. Yay!
        return $user;
    }

    $user = $this->createUser(
        $meData['email'],
        // a blank password - this user hasn't created a password yet!
        '',
        $meData['firstName'],
        $meData['lastName']
    );

    return $user;
}
```

Try the process again. No error this time - we find the existing user and use it instead of creating a new one.

Duplicate Emails

There is one other edge-case. What if we *don't* find any users with this COOP user id, but there *is* already a user with this email? This might be because the user registered on TopCluck, but hasn't gone through the COOP authorization process.

Pretty easily, we can do another lookup by email:

```
private function findOrCreateUser(array $meData)
{
    if ($user = $this->findUserByCOOPId($meData['id'])) {
        // this is an existing user. Yay!
        return $user;
    }

    if ($user = $this->findUserByEmail($meData['email'])) {
        // we match by email
        // we have to think if we should trust this. Is it possible to
        // register at COOP with someone else's email?
        return $user;
    }

    $user = $this->createUser(
        $meData['email'],
        // a blank password - this user hasn't created a password yet!
        '',
        $meData['firstName'],
        $meData['lastName']
    );

    return $user;
}
```

Cool. But be careful. Is it easy to fake someone else's email address on COOP? If so, I could register with someone else's email there and then use this to login to that user's TopCluck account. With something other than COOP's own user id, you need to think about whether or not it's possible that you're getting falsified information. If you're not sure, it might be safe to break the process here and force the user to type in their TopCluck password for this account before linking them. That's a bit more work, but we do it here on KnpUniversity.com.

Finishing Registration

When you *do* have a new user, instead of just creating the account, you may want to show them a finish registration form. This would let them choose a password and fill out any other fields you want.

We've got more OAuth-focused things that we need to get to, so we'll leave this to you. But the key is simple: store at least the coopAccessToken, coopUserId and token expiration in the session and redirect to a registration form with fields like email, password and anything else you need. You could also store the email in the session and use it to prepopulate the form, or even make another API request to /api/me to get it. When they finally submit a valid form, just create your user then. It's really just like any registration form, except that you'll also save the COOP access token, user id, and expiration when you create your user.

Chapter 6: OAuth with Facebook

Now that Brent has TopCluck up and running he can finally challenge Farmer Scott to an all out egg counting brawl. Brent knows that if they are both tracking egg collections on TopCluck Farmer Scott will be proven wrong and everyone will see how awesome his hens really are.

But what fun is winning if no one else gets to see? So Brent hatches another idea: having users share their chicken-laying progress on Facebook.

Fortunately, Facebook uses OAuth 2.0 for their API, so we're already dangerous. And like a lot of sites, they even have a PHP library to help us work with it. Installing it via Composer is easy. In fact, I already added it to our composer.json, so the library is downloaded and ready to go:

```
{
    "require": {
        ...
        "facebook/php-sdk": "~3.2.3"
    }
}
```

The library has a simple example, but it's easier to see it integrated in a real application.

The FacebookOAuthController

We're going to integrate with Facebook using the same authorization code grant type we just used with COOP. So it shouldn't be any surprise that we need the same 2 pages as before: 1 that redirects to Facebook and 1 that handles things after Facebook redirects back to us.

In fact, if you open up FacebookOAuthController.php, you'll see that I've started us with a setup that looks exactly like we had with COOP.

Starting the Redirect

Let's start by adding a link on the homepage to "Connect with Facebook":

```
{# views/dashboard.twig #}
{# ... #}

<div class="panel panel-default">
    <div class="panel-body">
        Share your progress on Facebook!
        <a href="{{ path('facebook_authorize_start') }}">Connect with Facebook</a>
    </div>
</div>
```

When we click this, we hit the code in the first function. Just like before, our job is to redirect to the authorize URL on Facebook. If we dig a little bit on Google, we can see this is /dialog/oauth. We could start building this by hand, but the Facebook SDK can help us out.

If we look at their simple usage example of the PHP SDK, we can see how to create the Facebook object. Copy this into the code for our page:

```
// src/OAuth2Demo/Client/Controllers/FacebookOAuthController.php
// ...

public function redirectToAuthorization()
{
    $config = array(
      'appId' => 'YOUR_APP_ID',
      'secret' => 'YOUR_APP_SECRET',
      'allowSignedRequest' => false
    );

    $facebook = new \Facebook($config);

    die('Todo: Redirect to Facebook');
}
```

We don't need the require part because we're using Composer, which takes care of this for us.

Just like with COOP, we need to register our application with Facebook to get our client id and secret.

Creating your Facebook Application

Head over to developers.facebook.com and create a new application. Give it a name and choose your favorite category. Immediately, we have a App ID and App Secret. Let's paste these into our code:

```
public function redirectToAuthorization()
{
    $config = array(
      'appId' => '1386038978283XXX',
      'secret' => '9ec32a48f1ad1988e0d4b9e80a17dXXX',
      'allowSignedRequest' => false
    );

    $facebook = new \Facebook($config);

    die('Todo: Redirect to Facebook');
}
```

Redirecting the User

Now, to get the authorize URL, we can use the getLoginUrl() function on the SDK. Remember that this URL always has 3 important things on it: the client ID, the redirect URI back to our site and the list of scopes we need. The object already has our client ID, so lets pass the redirect URI and scopes here. For Facebook, these are called redirect_uri and scope:

```
public function redirectToAuthorization()
{
    // ...

    $redirectUrl = $this->generateUrl(
        'facebook_authorize_redirect',
        array(),
        true
    );

    $url = $facebook->getLoginUrl(array(
        'redirect_uri' => $redirectUrl,
        'scope' => array('publish_actions', 'email')
    ));

    die('Todo: Redirect to Facebook');
}
```

To know which scopes you need, you have to check with the API you're using. If we google about Facebook API scopes, we find a page that explains all of them. We'll ultimately want to be able to get basic user information *and* post to a user's timeline. These are email and publish_actions.

Finally, let's redirect the user to this URL. The flow should feel completely familiar by now:

```
public function redirectToAuthorization()
{
    // ...
    $url = $facebook->getLoginUrl(array(
        'redirect_uri' => $redirectUrl,
        'scope' => array('publish_actions', 'email')
    ));

    return $this->redirect($url);
}
```

Registering the Redirect URI

When we try it out, we *do* go to Facebook's /dialog/oauth with the client_id, redirect_uri and scope parameters. But we get an error:

```
Given URL is not allowed by the Application configuration. One or more
of the given URLs is not allowed by the App's settings. It must match
the Website URL or Canvas URL, or the domain must be a subdomain of one
of the App's domains.
```

It's complaining about the redirect URL we're sending. For added security, OAuth servers allow, and sometimes require you to configure your redirect URL in your application. Go back to our application and click Settings and then "Add Platform". Choose "Website" and then fill in the URL of your site.

Tip

Facebook likes to change their interface, so this may look different someday soon! But one way or another, you're looking for a way to register your redirect URL.

And just like that, when we try it again, it works. Facebook made us do that so that no other sites can try to use our app id and have Facebook redirect back to some other domain. COOP's application settings also have this ability, but it wasn't required, so we skipped it. But, it's always better to fill this in.

At the authorize URL, Facebook describes the scopes that we're asking for, including the ability to post. One nice thing about Facebook is that we can choose to grant this scope, but make any posts show only to us. That's a great way to test things.

Getting the Access Token

When we finish, we're redirected back to our second page, which still has the original todo message. But we have a code query parameter, and we know that it can be exchanged for an access token.

Start by creating a private function that creates the Facebook object, and use it in both functions:

```
public function redirectToAuthorization()
{
    $facebook = $this->createFacebook();
    // ... the rest of the original function
}

public function receiveAuthorizationCode(Application $app, Request $request)
{
    $facebook = $this->createFacebook();

    die('Todo: Handle after Facebook redirects to us');
}

private function createFacebook()
{
    $config = array(
      'appId' => '1386038978283XXX',
      'secret' => '9ec32a48f1ad1988e0d4b9e80a17dXXX',
      'allowSignedRequest' => false
    );

    return new \Facebook($config);
}
```

OAuth tells us that our next step is to make an API request to the token endpoint to exchange our authorization code for an access token. That's absolutely right, and it can be done with the help of the SDK:

```
public function receiveAuthorizationCode(Application $app, Request $request)
{
    $facebook = $this->createFacebook();

    $userId = $facebook->getUser();
    var_dump($userId);die;

    die('Todo: Handle after Facebook redirects to us');
}
```

When we try the process again, we get a valid-looking user id. So, what just happened?

The getUser() method does a whole lot more than it looks like. It actually looks for the code query parameter and makes the API request to get the access token automatically! This is awesome, but it's also magic! If you can keep in mind how OAuth works and what's happening behind the scenes at each step, you'll be in great shape when something goes wrong.

Handling Failure

Just like with COOP, we need to handle failure. If we're missing the authorization code or something else goes wrong behind the scenes, the getUser() method will return 0. Let's use that to render the error template:

```
public function receiveAuthorizationCode(Application $app, Request $request)
{
    // ...
    $userId = $facebook->getUser();

    if (!$userId) {
        return $this->render('failed_authorization.twig', array(
            'response' => $request->query->all()
        ));
    }
    // ...
}
```

When something *does* go wrong, Facebook will redirect back to us with information about what went wrong on the standard error and error_description query parameters. Because they're following this OAuth standard, we can easily find error details and even decide what to do next. For example, if the error is set to access_denied, then it means the user denied our authorization request. In our app, I'm just passing all of the query parameters into a template that will display them.

To try this, we first need to go to Facebook and remove the app from our account. Unlike COOP, most OAuth servers remember if you authorized an app and don't ask you again.

On TopCluck, click "Connect with Facebook" again but "Cancel" the authorization request. After the redirect, we see the error, error_description and error_reason query parameters. But instead of seeing the error template, our valid userId is printed out as if it were successful. What just happened?

Our OAuth flow *did* fail. But even still, the Facebook object looks and finds a valid access token that it stored in the session from the last, successful authorization. That's nice, but it's unexpected. Just remember that getUser() tries many things: like exchanging the authorization code for an access token or simply finding an access token that it already stored in the session.

To see the error page, clear out your session cookie to reset everything. Log back in, then connect with Facebook but deny the request again. Oh Cluck! Error page! Without any session data to fall back on, the Facebook object doesn't have an access token and so can't make an API request to get the user id.

Saving the Facebook User ID

In CoopOAuthController, once we have the access token, our next step was to store some details in the database for the user, like the COOP user id, access token and expiration date.

For Facebook, I want to do something similar, but let's *only* store the Facebook user id. We can do this without any more work because the getUser() function gives us that id:

```
public function receiveAuthorizationCode(Application $app, Request $request)
{
    $facebook = $this->createFacebook();
    $userId = $facebook->getUser();
    // ...

    $user = $this->getLoggedInUser();
    $user->facebookUserId = $userId;
    $this->saveUser($user);

    return $this->redirect($this->generateUrl('home'));
}
```

And of course, let's redirect back to the homepage after finishing. Try the whole cycle out - this time approving our application's authorization request. We now know that a lot is happening behind the scenes.

First, the Facebook object exchanges the authorization code for an access token and saves it in the session. This all happens when we call getUser(). Next, we save the Facebook user ID into the database and redirect to the homepage. Clicking the "User Info" box shows us the Facebook ID.

Store the Access Token in the Database?

So why aren't we storing the access token or expiration? Actually, this is up to you. The Facebook object is automatically storing the access token in the session. So, everything is easy right now.

But on the user's next session, the access token will be gone and we'll need to re-ask the user to authorize. If you want to avoid this, you could store the Facebook access token in the database. In a second, I'll show you how you'd use that access token. Of course, these tokens don't last forever, so eventually you'll need to re-authorize them or use a Using Refresh Tokens, the topic of an upcoming chapter!

Chapter 7: Facebook: Using the API, Logging in and Failure

Sharing on your Wall

If the current user has a Facebook ID, let's replace the "Connect with Facebook" link with one called "Share" that will post to their timeline:

```
{# views/dashboard.twig #}

<div class="panel-body">
    {% if user.facebookUserId %}
        Share how many eggs you've collected today on Facebook!
        <a href="{{ path('facebook_share_place') }}" class="btn btn-info">Share</a>
    {% else %}
        Share your status on Facebook!
        <a href="{{ path('facebook_authorize_start') }}">Connect with Facebook</a>
    {% endif %}
</div>
```

The URL I'm generating here is pointing to a function called shareProgressOnFacebook in FacebookOAuthController:

```
// src/OAuth2Demo/Client/Controllers/FacebookOAuthController.php
// ...

public function shareProgressOnFacebook()
{
    die('Todo: Use Facebook\'s API to post to someone\'s feed');

    return $this->redirect($this->generateUrl('home'));
}
```

Click the link to see the message in my die statement being printed.

Using the Facebook API

To post to someone's timeline, we'll use Facebook's API. Like with any API that uses OAuth, we just need to know the URL, the HTTP method, any data we need to send, and how the access token should be attached to the request.

With some quick googling, we see that we need to make a POST request to /[USER_ID]/feed and send message and access_token POST data.

We could *absolutely* do this manually, using the nice Guzzle library from before. But since we're using the Facebook SDK, it's even easier.

Use the createFacebook method from before to get our Facebook object and then use its api method. This takes 3 arguments: the API URL, the HTTP method, and any parameters we need to send:

```
public function shareProgressOnFacebook()
{
    $facebook = $this->createFacebook();

    $facebook->api(
        '/'.$facebook->getUser().'/feed',
        'POST',
        array(
            'message' => 'TEST',
        )
    );

    die('Todo: Use Facebook\'s API to post to someone\'s feed');
    // ...
}
```

The handy $facebook->getUser() method gives us the right USER_ID for the URL. The only missing piece is the access_token parameter, which we can leave out because the Facebook class adds that automatically for us. Again, that's really cool - just don't lose sight of how things are really working behind the scenes.

Let's set the return value to a variable and dump it:

```
$result = $facebook->api(
    '/'.$facebook->getUser().'/feed',
    'POST',
    array(
        'message' => 'TEST',
    )
);
var_dump($result);die;
```

Refresh the page to try it out. It prints out an array with an id and a long number string. The response from api is specific to what you're trying to do. In this case, this is the ID of the new post it made. When I go to my Facebook page, there's my egg-citing post!

Remember that one of the reasons this works is that our authorization URL included the scope publish_actions. Had we *not* done that, this request would fail.

Tip

With Facebook and other OAuth servers, users are able to approve *some* of the scopes requested by your application but deny others. So code defensively - API requests may fail!

Let's make the message more realistic by putting in my egg count and finish the flow by redirecting back to the homepage:

```
public function shareProgressOnFacebook()
{
    $facebook = $this->createFacebook();
    $eggCount = $this->getTodaysEggCountForUser($this->getLoggedInUser());

    $facebook->api(
        '/'.$facebook->getUser().'/feed',
        'POST',
        array(
            'message' => sprintf('Woh my chickens have laid %s eggs today!', $eggCount),
        )
    );

    return $this->redirect($this->generateUrl('home'));
}
```

Refresh to try it all again. Check Facebook to see that we're bragging about our egg-laying hens' progress!

Handling Failure and Re-Authorizing

Of course, the API request may fail, especially in the world of OAuth where the access token might be expired. If any API request fails, the Facebook class will throw a FacebookApiException. That's great, because we can wrap the API call in a try-catch block:

```
try {
    $facebook->api(
        '/'.$facebook->getUser().'/feed',
        'POST',
        array(
            'message' => sprintf('Woh my chickens have laid %s eggs today!', $eggCount),
        )
    );
} catch (\FacebookApiException $e) {
    // it failed!
}
```

If you want to get information about the error, the exception object has a few useful methods, like getResult(), which gives you the raw API error response or getType() and getCode(). Facebook has a helpful page called Using the Graph API that talks

about the API and also the errors you might get back. If getType() returns OAuthException, or if the code is 190 or 102, the error is probably related to OAuth and we should try re-authorizing them:

```
try {
    $facebook->api(
        '/'.$facebook->getUser().'/feed',
        'POST',
        array(
            'message' => sprintf('Woh my chickens have laid %s eggs today!', $eggCount),
        )
    );
} catch (\FacebookApiException $e) {
    // https://developers.facebook.com/docs/graph-api/using-graph-api/#errors
    if ($e->getType() == 'OAuthException' || in_array($e->getCode(), array(190, 102))) {
        // our token is bad - reauthorize to get a new token
        return $this->redirect($this->generateUrl('facebook_authorize_start'));
    }

    // it failed for some odd reason...
    throw $e;
}
```

There's even another page that talks about handling expired tokens in more detail. If this seems a little unclear, that's probably because Facebook's error documentation is a little fuzzy.

If it's any other error, I'll just throw the original exception. You could even render some custom error page.

With any API that uses OAuth, if you can be smart enough to detect when API requests fail due to an expired access token, you can give your users a better experience by having them re-authorize your application instead of just failing.

Re-trying an API Request

Depending on the error, you might also want to re-try the request. Let's refactor the API call into a new private method called makeApiRequest():

```
public function shareProgressOnFacebook()
{
    $eggCount = $this->getTodaysEggCountForUser($this->getLoggedInUser());
    $facebook = $this->createFacebook();

    $ret = $this->makeApiRequest(
        $facebook,
        '/'.$facebook->getUser().'/feed',
        'POST',
        array(
            'message' => sprintf('Woh my chickens have laid %s eggs today!', $eggCount),
        )
    );

    // if makeApiRequest returns a redirect, do it! The user needs to re-authorize
    if ($ret instanceof RedirectResponse) {
        return $ret;
    }

    return $this->redirect($this->generateUrl('home'));
}

private function makeApiRequest(\Facebook $facebook, $url, $method, $parameters)
{
    try {
        return $facebook->api($url, $method, $parameters);
    } catch (\FacebookApiException $e) {
        // https://developers.facebook.com/docs/graph-api/using-graph-api/#errors
        if ($e->getType() == 'OAuthException' || in_array($e->getCode(), array(190, 102))) {
            // our token is bad - reauthorize to get a new token
            return $this->redirect($this->generateUrl('facebook_authorize_start'));
        }

        // it failed for some odd reason...
        throw $e;
    }
}
```

This method does the exact same thing as before. The if statement checks to see if makeApiRequest() needs us to redirect the user back to the authorize URL.

But if we add a new $retry argument, we could run the request 1 more time if it fails:

```
private function makeApiRequest(\Facebook $facebook, $url, $method, $parameters, $retry = true)
{
    try {
        return $facebook->api($url, $method, $parameters);
    } catch (\FacebookApiException $e) {
        // ... the check for an expired token

        // re-try one time
        if ($retry) {
            return $this->makeApiRequest($facebook, $url, $method, false);
        }

        // it failed for some odd reason...
        throw $e;
    }
}
```

Of course, this is really only interesting if we expect Facebook to have a decent number of temporary failures. But the big idea is that you should do your best to figure out *why* a failure has happened and re-try if it makes sense.

Tip

If you're using the Guzzle library to make API requests (which the Facebook class does *not* use), it has built-in support for re-trying a request if it fails. See Guzzle Retry Subscriber (for Guzzle version 4).

This is especially useful in the world of OAuth. We *didn't* store the Facebook access token in the database. But if we had, we

could use it right now and re-try the request again:

```
private function makeApiRequest(\Facebook $facebook, $url, $method, $parameters, $retry = true)
{
    try {
        return $facebook->api($url, $method, $parameters);
    } catch (\FacebookApiException $e) {
        if ($e->getType() == 'OAuthException' || in_array($e->getCode(), array(190, 102))) {
            if ($retry) {
                $user = $this->getLoggedInUser();
                // this is fake code - we don't have a facebookAccessToken
                // property in our example project
                $facebook->setAccessToken($user->facebookAccessToken);

                return $this->makeApiRequest($facebook, $url, $method, false);
            }

            // ... the same redirect code
        }

        // ... the same throw code
    }
}
```

So if the access token were missing from the session and the one in the database hasn't expired, this will make everything work perfectly smooth. Since this is fake code, let's remove all the retry code for now:

```
private function makeApiRequest(\Facebook $facebook, $url, $method, $parameters)
{
    try {
        return $facebook->api($url, $method, $parameters);
    } catch (\FacebookApiException $e) {
        if ($e->getType() == 'OAuthException' || in_array($e->getCode(), array(190, 102))) {
            // our token is bad - reauthorize to get a new token
            return $this->redirect($this->generateUrl('facebook_authorize_start'));
        }

        // it failed for some odd reason...
        throw $e;
    }
}
```

Logging in with Facebook

Finally, let's make it so the farmers can login with their Facebook account. Let's start by adding a link on the login page. Just like with "Login with COOP", the URL is to the page that starts the Facebook authorization process:

```
{# views/user/login.twig #}
{# ... #}

<button type="submit" class="btn btn-primary">Login!</button>
OR
<div class="btn-group">
    <a href="{{ path('coop_authorize_start') }}" class="btn btn-default">
        Login with COOP
    </a>
    <a href="{{ path('facebook_authorize_start') }}" class="btn btn-default">
        Login with Facebook
    </a>
</div>
```

Logging in with Facebook is going to work *exactly* like logging in with COOP. In fact, let's just copy all the related code from CoopOAuthController into our FacebookOAuthController:

```
// src/OAuth2Demo/Client/Controllers/FacebookOAuthController.php
// ...

public function receiveAuthorizationCode(Application $app, Request $request)
{
    $facebook = $this->createFacebook();
    $userId = $facebook->getUser();
    // ...

    if ($this->isUserLoggedIn()) {
        $user = $this->getLoggedInUser();
    } else {
        $user = $this->findOrCreateUser($json);

        $this->loginUser($user);
    }

    $user->facebookUserId = $userId;
    $this->saveUser($user);
    // ...
}

private function findOrCreateUser(array $meData)
{
    if ($user = $this->findUserByCOOPId($meData['id'])) {
        return $user;
    }

    if ($user = $this->findUserByEmail($meData['email'])) {
        return $user;
    }

    $user = $this->createUser(
        $meData['email'],
        '',
        $meData['firstName'],
        $meData['lastName']
    );

    return $user;
}
```

But to create a user, we need some basic information, like email, first name and last name. With COOP, we made an API request to get this information. Let's do the same thing for Facebook, using the really important endpoint /me. And knowing that things can fail, let's make sure to wrap it in a try-catch block:

```
public function receiveAuthorizationCode(Application $app, Request $request)
{
    // ...

    try {
        $json = $facebook->api('/me?fields=email,first_name,last_name');
    } catch (\FacebookApiException $e) {
        return $this->render('failed_token_request.twig', array('response' => $e->getMessage()));
    }
    var_dump($json);die;
    // ...
}
```

Tip

Due to recent Facebook API changes, you now need to add ?fields= to explicitly ask for which fields you want.

At this point, we *should* have a valid access token, so if the request fails, something is very strange. That's why I'm showing an error page instead of redirecting them to re-authorize.

I'm dumping the result of the API request, so let's logout and try the process. But first, reset the database so that it doesn't find our existing user:

```
$ rm data/topcluck.sqlite
```

When we login with Facebook, we hit the dump, which holds a lot of nice information about the user:

```
array (size=12)
  'id' => string '100002910877036' (length=15)
  'name' => string '...' (length=17)
  'first_name' => string '...' (length=10)
  'last_name' => string '...' (length=6)
  ...
```

We're allowed to ask for this information because when we redirect the user for authorization, we're asking for the email scope. Let's update the findOrCreateUser() method to use this data.

First, change findUserByCOOPId() to findUserByFacebookId(), which is a shortcut method in my app to find a user by the facebookUserId() column:

```
private function findOrCreateUser(array $meData)
{
    if ($user = $this->findUserByFacebookId($meData['id'])) {
        // this is an existing user. Yay!
        return $user;
    }
    // ...
}
```

Next, change the firstName and lastName keys to match Facebook's API response:

```
private function findOrCreateUser(array $meData)
{
    // ...

    $user = $this->createUser(
        $meData['email'],
        // a blank password - this user hasn't created a password yet!
        '',
        $meData['first_name'],
        $meData['last_name']
    );

    return $user;
}
```

It's that easy! Go back to the login page and try the whole process. When it finishes, we can click on the "User Info" section to see that we're logged in as a new user.

And that's it! Since Facebook uses OAuth, working with it is almost exactly like working with COOP. The biggest differene is that Facebook has a PHP SDK, which makes life easier, but hides some of the OAuth magic that's happening behind the scenes. But now that you truly understand things, that's no problem for you!

Chapter 8: Implicit Grant Type with Google+

With Facebook integration done, Brent can use it to brag about his major egg-collecting success on Facebook for all his farmer friends to see .... including farmer Scott.

Now he wants to go a step further and let people invite their Google+ connections to signup for a TopCluck account. To make it rural hipster, he wants to do this entirely on the frontend with JavaScript. The user will click a button to authorize their Google+ account, see a list of their connections, and select which ones to invite - all without any page reloads.

The Implicit Grant Type

So far we've seen 2 different grant types, or strategies for exchanging the access token. These were Client Credentials and Authorization Code. Unfortunately, neither works inside JavaScript. The problem is that both involve making a request to the OAuth server using your client secret. As the name suggests, that string is a secret. So, printing it inside an HTML page and using it in JavaScript would be a terrible idea.

Instead, we need to look at one more grant type called Implicit. It's a lot like Authorization Code, but simpler.

JavaScript OAuth with Google+

To integrate with Google+, let's start by finding their JavaScript Quick Start, which is a little example app. If we follow the Google+ Sign-In button, we can get some actual details on how Google+ sign in works.

Now that we know a lot about OAuth, the "Choosing a sign-in flow" is really interesting. This is a great example of how the OAuth grant types will look slightly different depending on the server.

Pure server-side flow

First, look at the Pure server-side flow. If you look closely, the steps are describing the authorization code grant type. The redirect is done via JavaScript, but with all the familiar parameters like scope, response_type and client_id. After the redirect, the server checks for a code query parameter and uses a Google PHP SDK to get an access token.

Hybrid server-side flow

Next, go back and look at the Hybrid server-side flow. This is another version of the authorization code grant type, which has 2 major differences.

First, instead of redirecting the user, we use a little Google+ JavaScript library and some markup. When the user clicks the sign in link, it doesn't redirect the user. Instead, it opens a popup, which asks the user to authorize your app.

The second big difference is how we get the authorization code. After the user authorizes our application, the popup closes. Instead of redirecting the user to a URL on our site, a JavaScript function is called and passed the code. We then send this via AJAX to a page on our server, which exchanges it for an access token.

This approach *still* involves the server, but the work of getting the code is delegated to JavaScript. In reality, it's just another version of the authorization code grant type.

Client-side Flow

Finally, let's look at the Client-side Flow, which is where everything happens in JavaScript. There are 3 variants of this type, but they're all basically the same. When we press the "Click me" demo button, we get a popup asking for authorization. And immediately after approving, some JavaScript on the page shows us the access_token and some other details. This happens completely without the server.

Creating the Google Application

Like everything, our first step is to create an application so that we have a client ID and client secret. Click to go to the Developers Console and create a new project.

Next, click APIs and auth and make sure the "Google+ API" is set to ON.

Finally, click "Credentials" on the left and click the "Create New Client ID" button. Keep "Web Application" selected and fill in your domain name. Since we won't be using the Authorization Code grant type and redirecting the user, we only really need to worry about the JavaScript origins. Google makes us fill these in for security purposes - a topic we'll cover later.

When we're finished, we have a brand new Client ID and secret. Keep these handy!

Including the JavaScript SDK

The implicit OAuth flow can be done without any tools, but Google makes our life a lot easier by giving us a JavaScript SDK. Copy the script into our layout:

```
{# views/base.twig #}

<script src="http://code.jquery.com/jquery-2.0.3.min.js"></script>
<script src="{{ app.request.basePath }}/js/bootstrap.min.js"></script>

<script type="text/javascript">
    (function () {
        var po = document.createElement('script');
        po.type = 'text/javascript';
        po.async = true;
        po.src = 'https://apis.google.com/js/client:plusone.js';
        var s = document.getElementsByTagName('script')[0];
        s.parentNode.insertBefore(po, s);
    })();
</script>

{# ... #}
```

This exposes a global gapi object we'll use in a second.

Initiate the Sign-in Flow

Let's add a "Connect with Google+" button on the homepage and attach a jQuery click event listener to it:

```
{# views/dashboard.twig #}

<!-- ... -->
<a href="#" class="btn btn-lg btn-info js-google-signin">Connect with Google+</a>
<!-- ... -->

{% block javascripts %}
    {{ parent() }}

    <script>
        jQuery(document).ready(function() {
            $('.js-google-signin').on('click', function(e) {
                // prevent the click from going to #
                e.preventDefault();
            });
        });
    </script>
    {# Put any JavaScript here #}
{% endblock %}
```

We can start the authentication process by using the signIn method of the gapi.authentication JavaScript object:

```
jQuery(document).ready(function() {
    $('.js-google-signin').on('click', function(e) {
        // prevent the click from going to #
        e.preventDefault();

        gapi.auth.signIn();
    });
});
```

When we try it, nothing happens. In fact, there's a JavaScript error:

```
cookiepolicy is a required field.  See
https://developers.google.com/+/web/signin/#button_attr_cookiepolicy
for more information.
```

What we're trying to do here is *similar* to the step in the Authorization Code grant type where we originally redirect the user to the OAuth server. There are details we need to send to Google+, like our client id and the scopes we want.

In fact, the gapi.auth object has nice documentation and the signIn method there shows us the common parameters we need:

```
// just the example copied from https://developers.google.com/+/web/api/javascript#gapiauthsigninparameters
function initiateSignIn() {
  var myParams = {
    'clientid' : 'xxxxxxxxxxxxx..apps.googleusercontent.com',
    'cookiepolicy' : 'single_host_origin',
    'callback' : 'mySignInCallback',
    'scope' : 'https://www.googleapis.com/auth/plus.login',
    'requestvisibleactions' : 'http://schemas.google.com/AddActivity'
    // Additional parameters
  };
  gapi.auth.signIn(myParams);
}
```

Let's copy these into our JavaScript. Update the clientid but keep the scope as it will let us access the user's social graph. The requestvisibleactions parameter relates to posting activities - you can leave it, but we won't need to worry about it:

```
    jQuery(document).ready(function() {
        $('.js-google-signin').on('click', function(e) {
            // prevent the click from going to #
            e.preventDefault();

            var myParams = {
                'clientid': '104029852624-a72k7hnbrrqo02j5ofre9tel76ui172i.apps.googleusercontent.com',
                'cookiepolicy': 'single_host_origin',
                'callback': 'mySignInCallback',
                'scope': 'https://www.googleapis.com/auth/plus.login',
                'requestvisibleactions': 'http://schemas.google.com/AddActivity'
            };
            gapi.auth.signIn(myParams);
        });
    });
```

The cookiepolicy tells the SDK to set cookie data that's only accessible by our host name. This is a necessary detail just to make sure the data being passed around can't be read by anyone else.

All of these parameters are explained nicely on the documentation page.

Let's try it again! Now we get the popup which asks us to authorize the app. And when we approve, we get a JavaScript error:

```
Callback function named "mySignInCallback" not found
```

That's actually great! Instead of redirecting the user back to a URL on our site, Google passes us the OAuth details by calling a JavaScript function. Calling the JavaScript function here serves the same purpose as a browser redirect: it hands off authorization data from the server to the client. This isn't special to the Implicit flow - the Hybrid server-side flow we looked at earlier is an example of an Authorization Code grant type that does this part in JavaScript as well.

Step 5 of the docs show us how the function might look. Let's create our mySignInCallback function and dump the auth information.

```
function mySignInCallback(authResult) {
    console.log(authResult);
}
```

Refresh and try it again! Awesome, we see it print out an object with an access_token. This is the big difference between the Implicit flow and the Authorization Code grant types. With Authorization Code, this step returns a code, which we then still need to exchange for an access token by making an API request. But with Implicit, the access token is given to us immediately.

Choosing Authorization Code versus Implicit

Remember that whether we're redirecting the user or using this popup method, we can *choose* to use the Authorization Code or Implicit grant type. In fact, the JavaScript object contains both the token *and* an authorization code. So we can either choose to use the token in JavaScript, or do a little more work to send the code to our server via AJAX and exchange that for a token.

Instead of sending us both, other OAuth servers let you choose between the code and the token.

Remember the response_type parameter we used with Coop? We set it to code, which is why we got back a code query parameter on the redirect. But we could also set it to token. And if we did, the redirect would have contained a token parameter instead of the code.

The response_type is how we tell the OAuth server which grant type we want to use. Even Facebook has a response_type parameter on its login URL, which has the same 2 values.

Authorization Code versus Implicit

So why would anyone choose Authorization Code over Implicit since it has an extra step? The big answer is security, which we'll talk about more in the next chapter. Another disadvantage, which is also related to security, is that the Implicit grant type can't give you a refresh token.

Chapter 9: Finishing the Login Callback

Finish the login callback function by copying the example from Step 5 of the docs and tweaking the code to use jQuery:

```
function mySignInCallback(authResult) {
    if (authResult['status']['signed_in']) {
        // Update the app to reflect a signed in user
        $('.js-google-signin').hide();
    } else {
        // Possible error values:
        //   "user_signed_out" - User is signed-out
        //   "access_denied" - User denied access to your app
        //   "immediate_failed" - Could not automatically log in the user
        console.log('Sign-in state: ' + authResult['error']);
    }
}
```

When we refresh and try again, the sign in button disappears, proving that authentication was successful!

Using the API

Just like with the Facebook PHP SDK, the Google JavaScript SDK now has an access token that it's storing. This means we can start making API calls. I'll copy in a function that uses the API to get a list of all of the people in my circles and print their smiling faces:

```
// views/dashboard.twig
function loadCirclesPeople() {
    var request = gapi.client.plus.people.list({
        'userId': 'me',
        'collection': 'visible'
    });
    request.execute(function (people) {
        var $people = $('#google-plus-people');
        $people.empty();
        for (var personIndex in people.items) {
            var person = people.items[personIndex];
            $people.append('<img src="' + person.image.url + '">');
        }
    });
}
```

This looks for a div with the id google-plus-farmers, so let's add that to our page:

```
{# views/dashboard.twig #}

<!-- ... -->
<a href="#" class="btn btn-lg btn-info js-google-signin">Connect with Google+</a>
<div id="google-plus-farmers"></div>
<!-- ... -->
```

Let's call this function automatically after we authenticate. This code loads the Google+ part of the SDK and calls our function.:

```
function mySignInCallback(authResult) {
    if (authResult['status']['signed_in']) {
        // ...

        // loads the gapi.client.plus JavaScript object
        gapi.client.load('plus','v1', function() {
            loadCirclesPeople();
        });
    } else {
        // ...
    }
}
```

Ok, let's try it! When we refresh and sign in, we get a beautiful box of farmers in our circle! In my console, if we click on the AJAX call that was made, we can see that an access token was sent on the Authorization: Bearer header. OAuth is happening behind the scenes!

Page parameters

Our ultimate goal is for the user to be able to choose from the people in their circles and invite them to join TopCluck. With all the OAuth stuff behind us, this is just a matter of writing some JavaScript and figuring out exactly how to use the Google+ API to accomplish this. We'll leave this to you!

But there's one more small thing that's bothering me. When we click to sign in, the sign-in function is called twice, which means loadCirclesPeople is called twice and 2 API requests are made to Google.

Regardless of why this happens, we could of course avoid the double-calls by using a simple variable:

```
var isSignedIn = false;
function mySignInCallback(authResult) {
    if (authResult['status']['signed_in']) {
        if (isSignedIn) {
            return;
        }
        isSignedIn = true;

        // ...
    } else {
        // ...
    }
}
```

But the reason this is happening is more interesting. Rememember how the Facebook SDK stores the access token details in the session? The Google JavaScript SDK stores those details in a cookie. This means that since we've already authorized with Google+, we should *still* be signed in if we refresh. The callback function is called twice since we were already authenticated *and* we clicked to authenticate again.

If we already authorized during this session, we can avoid making the user click the Connect button by moving the signIn parameters to meta tags. This is actually what Step 4 of the example does. Let's copy these meta tags into our layout and update it with our client id. We can also add the callback parameter here:

```
{# views/base.twig #}
{# ... #}

<meta name="description" content="">
<meta name="viewport" content="width=device-width">

<meta name="google-signin-clientid" content="104029852624-a72k7hnbrrqo02j5ofre9tel76ui172i.apps.googleusercontent.com" />
<meta name="google-signin-scope" content="https://www.googleapis.com/auth/plus.login" />
<meta name="google-signin-requestvisibleactions" content="http://schemas.google.com/AddActivity" />
<meta name="google-signin-cookiepolicy" content="single_host_origin" />
<meta name="google-signin-callback" content="mySignInCallback" />
{# ... #}
```

Google calls this page-level configuration. One big advantage is that if we already have an access token stored in a cookie, it will execute the callback function on page load. Now that we have these, remove the params entirely:

```
// views/dashboard.twig
$('.js-google-signin').on('click', function(e) {
    // prevent the click from going to #
    e.preventDefault();

    gapi.auth.signIn();
});
```

Refresh the page. Instantly, the Sign in button disappears and our circles show up. Whether we're managing the access token on the server or in JavaScript, we can make it persist throughout a session. This isn't always clear, since the Facebook and Google SDK's do a lot automatically for us. Just keep thinking about how OAuth works and you'll be in great shape.

In this chapter, we saw how you can choose between the authorization code or implicit grant type when starting the authorization process. And although it has nothing to do with grant types, we also saw how the authorization process can be done by redirecting the user, like we saw in past chapters, *or* by opening a popup and communicating with JavaScript. Which method you'll use will largely depend on the OAuth server and what it supports most easily.

But if you need a *pure* JavaScript solution that never touches the server, then you need the implicit grant type. Even if you can keep much of the flow in JavaScript, the authorization code *still* needs a server so that it can use the client secret to exchange the code for the token.

Chapter 10: Using Refresh Tokens

Brent has a big problem. A user can already log in to TopCluck and click a link that uses the COOP API to count the number of eggs collected that day. But that's manual, and if a farmer forgets, his egg count will show up as zero.

Intead, he wants to write a CRON JOB that automatically counts the eggs for every user each day. The problem is that each COOP access token expires after 24 hours. And since we can't redirect and re-authorize the user from a CRON job, when a token expires, we can't count eggs.

Refresh Tokens

Fortunately, OAuth comes with an awesome idea called refresh tokens. If you have a refresh token, you can use it to get a new access token. Not all OAuth servers support refresh tokens. Facebook, for example, allows you to get long-lived access tokens, with an expiration of 60 days. But those are really just access tokens, and when they expire, you'll need to send the user back through the login flow.

Why do refresh tokens exist? If an attacker steals an access token, there is only a short window they can use it before it expires. If an attacker gains a refresh token, it is useless to them without the client's credentials, as you'll see. Having two keys instead of one is a method often used in security to make it harder for attackers to compromise a system.

Fortunately, COOP *does* support refresh tokens. Open up the CoopOAuthController where we make the API request to /token. Let's dump this response and go through the process:

```
// src/OAuth2Demo/Client/Controllers/CoopOAuthController.php
public function receiveAuthorizationCode(Application $app, Request $request)
{
    // ...

    $request = $http->post('/token', null, array(
        // ...
    ));

    $response = $request->send();
    $responseBody = $response->getBody(true);
    $responseArr = json_decode($responseBody, true);

    var_dump($responseArr);die;
    // ...
}
```

Ah hah! The response has an access_token *and* a refresh_token. Let's store the refresh token to a column on the user so we can re-use it later:

```
public function receiveAuthorizationCode(Application $app, Request $request)
{
    // ...

    // after the /token request
    $accessToken = $responseArr['access_token'];
    $expiresIn = $responseArr['expires_in'];
    $expiresAt = new \DateTime('+'.$expiresIn.' seconds');
    $refreshToken = $responseArr['refresh_token'];

    // ...
    $user->coopRefreshToken = $refreshToken;
    $this->saveUser($user);
    // ...
}
```

Tip

In order to get a refresh token, you *may* need to pass an extra parameter (e.g. offline) when redirecting the user to

authorize.

No Refresh Tokens in the Implicit Grant Type

Even if an OAuth server supports refresh tokens, you won't be given one if you use the implicit flow. To see what I mean, change the response_type parameter on our COOP authorize URL to token and add a die statement right at the top of the code that handles the redirect:

```
public function redirectToAuthorization(Request $request)
{
    $redirectUrl = $this->generateUrl('coop_authorize_redirect', array(), true);

    $url = 'http://coop.apps.knpuniversity.com/authorize?'.http_build_query(array(
        'response_type' => 'token',
        'client_id' => 'TopCluck',
        'redirect_uri' => $redirectUrl,
        'scope' => 'eggs-count profile'
    ));

    return $this->redirect($url);
}

public function receiveAuthorizationCode(Application $app, Request $request)
{
    die;
    // ...
}
```

When we try the process again, COOP redirects us back with a URL that contains an access token instead of the authorization code:

```
http://localhost:9000/coop/oauth/handle#
    access_token=eaf215f677bea1562026df05ecca202163a6c69f
    &expires_in=86400
    &token_type=Bearer
    &scope=eggs-count+profile
```

Since this is how the implicit flow works, this no surprise. But notice that there's no refresh token. That's one major disadvantage of using the implicit grant type.

Using the Refresh Token

Let's undo our change and go back to asking for an authorization code.

We can't see it visually, but when we try the whole process, the user record in the database now has a coopRefreshToken saved to it.

I've already started the little script for the CRON job, which you can see at data/refresh_tokens.php. What we want to do here is use the COOP API to count and save each user's daily eggs.

But first, we need to make sure that everyone has a non-expired access token. Let's use a method called getExpiringTokens() that I've already prepared. This queries the database and returns details for all users whose coopAccessExpiresAt value is today or earlier:

```
// data/refresh_tokens.php
$app = require __DIR__.'/../bootstrap.php';
use Guzzle\Http\Client;

// create our http client (Guzzle)
$http = new Client('http://coop.apps.knpuniversity.com', array(
    'request.options' => array(
        'exceptions' => false,
    )
));

// refresh all tokens expiring today or earlier
/** @var \OAuth2Demo\Client\Storage\Connection $conn */
$conn = $app['connection'];

$expiringTokens = $conn->getExpiringTokens();
```

Tip

In the background, this is just running a query similar to this:

```
SELECT * FROM users WHERE coopAccessExpiresAt < '2014-XX-YY';
```

Next, let's iterate over each expiring token. To get a refresh token, we'll make an API request to the very-familiar /token endpoint. In fact, I'll start by copying the Guzzle API call from CoopOAuthController:

```
// data/refresh_tokens.php
// ...

$expiringTokens = $conn->getExpiringTokens();

foreach ($expiringTokens as $userInfo) {

    $request = $http->post('/token', null, array(
        'client_id'     => 'TopCluck',
        'client_secret' => '2e2dfd645da38940b1ff694733cc6be6',
        'grant_type'    => 'authorization_code',
        'code'          => $code,
        'redirect_uri'  => $this->generateUrl('coop_authorize_redirect', array(), true),
    ));

    // make a request to the token url
    $response = $request->send();
    $responseBody = $response->getBody(true);
    var_dump($responseBody);die;
    $responseArr = json_decode($responseBody, true);

}
```

Of course, we don't have a $code variable, but we *do* have the user's refresh token. Change grant_type to be refresh_token and replace the code parameter with the refresh_token. We can also remove the redirect_uri, which isn't needed with this grant type:

```
$request = $http->post('/token', null, array(
    'client_id'     => 'TopCluck',
    'client_secret' => '2e2dfd645da38940b1ff694733cc6be6',
    'grant_type'    => 'refresh_token',
    'refresh_token' => $userInfo['coopRefreshToken'],
));
```

Let's try out the API call! Tweak the getExpiringTokens() method temporarily. We don't actually have any users with expiring tokens, but this change will return any tokens expiring in the next month, which should be everyone:

```
$expiringTokens = $conn->getExpiringTokens(new \DateTime('+1 month'));

foreach ($expiringTokens as $userInfo) {
    // ...

    $response = $request->send();
    $responseBody = $response->getBody(true);
    var_dump($responseBody);die;
    $responseArr = json_decode($responseBody, true);
}
```

Now, try it by executing the script from the command line:

```
$ $ php data/refresh_token.php
```

With any luck, we should see a familiar-looking JSON response:

```
{
    "access_token": "1729a2fc9e6d6da2d2cb877c5bf3239fd2c57d0d",
    "expires_in": 86400,
    "token_type": "Bearer",
    "scope": "eggs-count profile",
    "refresh_token": "f6ecef2bf0d16d7c13a983616b30d72ca915ab65"
}
```

Perfect! Now we just need to update the user with the new coopAccessToken, coopExpiresAt and coopRefreshToken. Again, we can copy or re-use some code from CoopOAuthController, since this is the same response from there. The saveNewTokens() method is a shortcut to update the user record with this data:

```
// data/refresh_tokens.php
// ...

foreach ($expiringTokens as $userInfo) {
    // ...

    $accessToken = $responseArr['access_token'];
    $expiresIn = $responseArr['expires_in'];
    $expiresAt = new \DateTime('+'.$expiresIn.' seconds');
    $refreshToken = $responseArr['refresh_token'];

    $conn->saveNewTokens(
        $userInfo['email'],
        $accessToken,
        $expiresAt,
        $refreshToken
    );
}
```

Tip

In the background, this is just running an UPDATE query against this user to update the access token, expiration and refresh token columns.

Let's add a little message so we can see what's going on:

```
$conn->saveNewTokens(
    $userInfo['email'],
    $accessToken,
    $expiresAt,
    $refreshToken
);
// ...

echo sprintf(
    "Refreshing token for user %s: now expires %s\n\n",
    $userInfo['email'],
    $expiresAt->format('Y-m-d H:i:s')
);
```

But when we try it now, the script blows up! Since we're still dumping the raw response, above the exception we can see the message "Invalid refresh token". The problem is that when we used the refresh token a second ago, the COOP API gave us a new one and invalidated the old one. We weren't saving it yet, so now we're stuck and need to re-authorize the user.

Tip

An OAuth server may or may not invalidate the refresh token after using it - that's totally up to the server.

Go back to the site, log out, and log back in with COOP. This will get a new refresh token for the user. And since we're saving the new refresh token, in our script each time, we can run it over and over again without any issues.

And now that we've refreshed everyone's access tokens, we could loop through each user and send an API request to count their eggs. The code for that would look almost exactly like code in the CountEggs.php file, so we'll leave that to you.

Nothing lasts Forever

Of course, nothing lasts forever, and even the refresh token will eventually expire. These tokens commonly last for 14-60 days, and afterwards, you have no choice but to ask the user to re-authorize your application.

Tip

A refresh token *could* last forever - it's up to the OAuth server. However, it's still possible that the user revokes access in the future.

This means that unless your OAuth server has some sort of key that lasts forever, our CRON job will eventually *not* be able to count the eggs for all of our farmers. We may need to send them an email to re-authorize or be ok that these inactive users aren't updated anymore.

Chapter 11: Security

Since TopCluck is handling a lot of access tokens for Brent's farmer friends, he wants to make sure it's secure. Nothing would be worse than for the access tokens of the TopCluck farmers to get stolen - allowing some city slicker to take control of the good people's farms!

Exchanging tokens in a secure way isn't easy because there are a lot of opportunities for a hacker to be listening to the requests or doing some other clever thing.

CSRF Protection with the state Parameter

Open up CoopOAuthController so that we can squash a really common attack. In the authorize redirect URL, add a state parameter and set its value to something that's only known to the session for *this user*. We can do that by generating a random string and storing it in the session:

```
// src/OAuth2Demo/Client/Controllers/CoopOAuthController.php
public function redirectToAuthorization(Request $request)
{
    $redirectUrl = $this->generateUrl('coop_authorize_redirect', array(), true);

    $state = md5(uniqid(mt_rand(), true));
    $request->getSession()->set('oauth.state', $state);
    $url = 'http://coop.apps.knpuniversity.com/authorize?'.http_build_query(array(
        'response_type' => 'code',
        'client_id' => 'TopCluck',
        'redirect_uri' => $redirectUrl,
        'scope' => 'eggs-count profile',
        'state' => $state
    ));

    return $this->redirect($url);
}
```

Let's also add a die statement in the receiveAuthorizationCode() function that's executed after COOP redirects back to us:

```
public function receiveAuthorizationCode(Application $app, Request $request)
{
    die;
    // ...
}
```

Log out and click to login via COOP. Of course, when we redirect to COOP, the new state parameter is there. Interestingly, after we authorize, COOP redirects back to us and *also* includes that exact state parameter.

In receiveAuthorizationCode(), we just need to make sure that state matches the string that we set in the session exactly. If it doesn't, let's render an error page: this could be an attack:

```
public function receiveAuthorizationCode(Application $app, Request $request)
{
    if ($request->get('state') !== $request->getSession()->get('oauth.state')) {
        return $this->render(
            'failed_authorization.twig',
            array('response' => array(
                'error_description' => 'Your session has expired. Please try again.'
            ))
        );
    }

    // ...
}
```

Using the state parameter is just like using a CSRF token with a form: it prevents XSS attacks.

When we log in now, it all still works perfectly.

Imagine I start the authorization process, but use a browser plugin to prevent COOP from redirecting me back to TopCluck. Then, I post the redirect URL with my valid authorization code to a forum somewhere, maybe embedded in an image tag. Assuming you're logged into TopCluck, when you view this page, the image tag will make a request to TopCluck, which exchanges the authorization code for an access token in the background.

So what? Well, CoopOAuthController would end up saving your coopUserId to the attacker's TopCluck account. This means when the attacker logs into TopCluck using COOP, they'll be logged in as *you*!

So, *always* use a state parameter. Fortunately, when you work with something like Facebook's SDK, this happens automatically. We didn't realize it, but it was generating a state parameter, saving it to the session, and checking it when we exchanged the authorization code for the access token. That's pretty nice.

Registering the Redirect URI

Head over to COOP and check out our application there. One field we left blank was the Redirect URI. Let's fill it in now with a made-up URL.

Try logging in again. This time, we immediately get an error from COOP:

> The redirect URI provided is missing or does not match

The redirect URI is a security measure that guarantees that nobody can use your client ID, which is public, to authorize users and redirect with the authorization code or access token back to *their* site. Many OAuth servers require this to be filled in. In fact, we saw that with Facebook earlier

I'll re-edit the application and put in our exact redirect_uri value. When we try to login in now, it works.

Most OAuth servers will require this value. Sometimes, the URL we put here must match the redirect_uri parameter *exactly*. Other times, it's a fuzzy match. This is up to the OAuth server you're using, but exact matching is much more difficult to fake.

In a client-side environment where the code or token is passed via JavaScript, the OAuth server may just ask you for your hostname or a list of JavaScript origins. These function the same way: to prevent JavaScript on some other hostname from using your client id.

The Insecurity of Implicit

The implicit grant type is the least secure grant type because the access token can be read by other JavaScript on your page and could be a victim of XSS attacks. If you decide to use implicit, you must be *extra careful* in preventing the attacks on the pages where access tokens are used in JavaScript.

This is another example of why registering an exact redirect URI is important. If an attacker locates just one XSS vulnerability on your site, they could manipulate the redirect URI to point there, and use it to steal access tokens. It's also even more important to validate your state parameter.

If it's at all possible to use the authorization code grant type instead, this is much better because even if there was a man in the middle or piece of JavaScript reading your authorization code, the client secret is still needed to turn that into an access token.

One interesting thing about the implicit grant type is that the access token is passed back as a URL fragment instead of a query parameter:

```
http://localhost:9000/coop/oauth/handle?code=abcd123
http://localhost:9000/coop/oauth/handle#access_token=wxyz5678
```

We didn't see this with Google+ because it was all being handled in the background for us. But this is really important because anything after the hash in a URL isn't actually sent when your browser requests a page. The JavaScript on your page can read this, but since it's not sent over the web, anyone listening between the user and the server won't be able to intercept it. That's not as important with the code, because the man-in-the-middle would still need the client secret to do anything with it.

Https

An important piece of OAuth security is using SSL. This means all requests to an OAuth server should be done using

HTTPS. The reason is that the access_token, is always sent in plain text. That's true when the OAuth server first gives us the access token and on *every single* API request we make back afterwards. This makes using OAuth APIs much more convenient for us developers, but if those requests aren't encrypted, you're asking for a fox in your hen house.

And when you make those calls over HTTPS, make sure you actually verify the SSL certificate. Your HTTP library will do this for you, but it will also give you the option to skip verification. This is tempting when developing locally or if you get an error like:

> Peer certificate cannot be authenticated with known CA certificates

But don't disable verification! That's like keeping the door open on your chicken coop! Turning off SSL Verification is the same as sending the access token unencrypted. Don't manually turn this off and you'll be okay.

Interestingly, *your* site doesn't technically need to use HTTPS. When the user is redirected back with the auth code, it's ok if someone intercepts this, since they won't also have your client secret.

But any time you have a logged in user, you should really use HTTPS. Without it, your user's session could be stolen by someone else on the same network! And all your hard work making your OAuth implementation secure will go to waste.

Authentication with OAuth

In our tutorial, we allow people to log in with COOP and Facebook. But this isn't the purpose of OAuth. Usually, we think that the only way for us to get an access token is for *that user* to give it to us directly via the authorization process. So when we're given an access token for Brent's account, we think "This must be Brent, let's log him into his TopCluck account".

With this authorization code grant type and the state parameter, this is safe. But suppose instead that we decide to use the implicit flow in JavaScript. After success, we'll send the new access_token via AJAX to the TopCluck server and authenticate the user by looking up the coopUserId associated with the token.

Now, what if some other site also allows you to authorize your COOP account with them. They now also have an access token for your COOP account. If they're nasty, or if your access_token gets stolen, someone could pass it directly to our AJAX endpoint and become authenticated on TopCluck in your account.

That's right - any site that has an access token to your Coop or Facebook account could use it to log into any other site that has this flawed login mechanism.

The moral is this: since OAuth is not meant for authentication, you need to be extra careful when you do this. Most importantly, stay away from the implicit grant type for authenticating users, as we have done in this tutorial.

The End

Our hero Brent's life is a lot better than when we started. Thanks to his CRON script, his chickens are getting fed everyday. And with the TopCluck site, he's well on his way to victory over farmer Scott *and* sharing his glory all over Facebook. All of this was possible by getting a deep understanding of OAuth, which unleashed us to do all kinds of interesting integrations with third-party sites. I know that you will have just as much success as Brent!

See you next time!