30

Social Media Integration Sharing and Authentication

AS SOCIAL MEDIA CONTINUES TO BECOME A CORNERSTONE OF OUR INTERACTIONS, more and more web applications benefit from the integration of these platforms. While each platform implements these integrations in a different fashion (using their own web services, API calls, etc.), there are common trends and approaches especially around the aspect of authentication that have taken hold. In this chapter we will explore those common technologies such as OAuth as well as how to perform a few common tasks with the popular social media platform Instagram. Note that while in this chapter we will focus on Instagram, the concepts and techniques described will apply to most all social media platforms.

Web Service Authentication with OAuth

OAuth is an open-source standard for authorization, allowing users to log into other websites using the credentials of the website they already have. Many blogs provide the ability to log into the site using, for example, Instagram credentials. OAuth presently comes in two similar yet incompatible versions, 1.0 and 2.0, with 2.0 being the most commonly implemented. We will be discussing the implementation of OAuth 2.0 in this chapter, however many of the concepts are similar for OAuth 1.0.

To understand OAuth we need to define the various entities or roles that interact in the authentication process, which are as follows:

- Resource: The thing you want access to. Often in OAuth schemes a single authorization is for access to multiple resources. For example, a user may wish to grant access to a third party to view the follower list and post messages to their Twitter account (where each is a separate resource).
- Owner: The owner of the resources in question.
- Resource Server: The website/server that controls the resources.

- Authorization Server: The website/server that is responsible for granting authorization to the resources.
- Client: The application that would like access to the owner's resource(s).

Ultimately the goal is for the Authorization Server to give a Client access to one or more Resources owned by the Owner, hosted on the Resource Server. In more visual terms, this process looks somewhat like Figure 30.1.

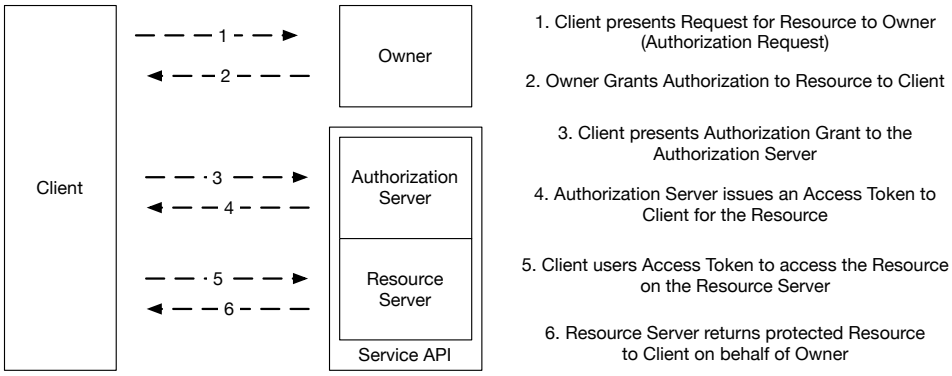

Figure 30.1.

Note the diagram in Figure 30.1 is not necessarily the exact request flow between these various entities. Depending on the nature of the authorization given by the user, the flow of the requests may also change. Ultimately in all cases however, it is the goal of OAuth to provide to the Client an access token that allows it to access the Resources the Owner has granted access to.

There is obviously much more to OAuth than simply this diagram, so let's take a moment to introduce a few more details. For starters, in order for a Client to participate in an OAuth transaction it first must be known to the Authorization Server. That is to say, the client must present credentials of its own to the Authorization Server alongside any Authorization Grant. The Authorization Server then evaluates the request to the resources in question not only by the grant provided by the Owner but also by the requesting Client's credentials. Each Client must register in advance with the service in question (e.g., go to the developer section of Twitter and register your application), and is granted a Client ID (public information) and Client Secret (private to the Client) to use in the OAuth process.

Once registered, how the actual flow of the OAuth process works is going to largely depend on the nature of the Grant being authorized by the Owner to the Client. OAuth 2.0 supports four distinct grant types:

- Authorization Code: Used for server-side applications (such as a PHP application) that needs access to a resource.
- Implicit: Used for mobile applications, or applications that run entirely on a device controlled by the Owner (such as a purely JavaScript application or an iPhone application).
- Resource Owner Password Credential: Only used for trusted Clients, such as Clients controlled by the Resource Owner.
- Client Credentials: Used with Application's API access.

For the purposes of this chapter we will only discuss the Authorization Code and Implicit grant types, as they are by far the most common you will encounter implementing OAuth in a typical web application.

Authorization Code Grants

Let's start with the most common OAuth implementation using an Authorization Code grant type. At a high level, this flow can be described as shown in Figure 30.2 (excluding subsequent requests actually using the provided access token).

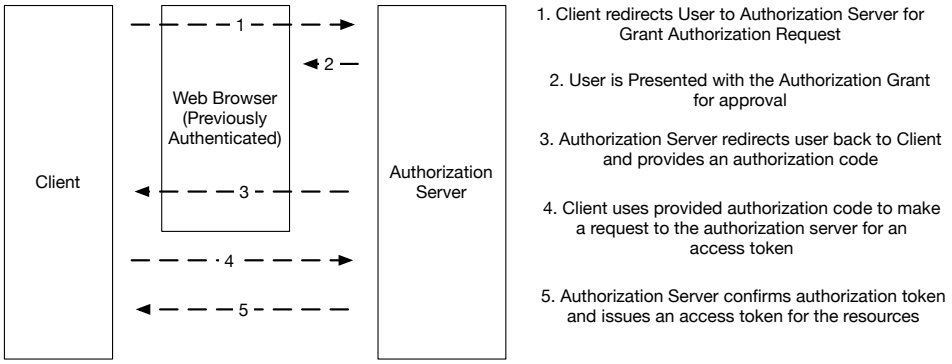

Figure 30.2.

In an authorization code workflow, used in server-side applications (such as those written in PHP), the Client redirects the user to the Authorization Server of a third party that the client would like access to, providing its client ID. The Authorization Server receives the request and presents the Owner with an authorization form outlining the resources being requested by the client and requesting they approve the authorization grant. If the Owner is not yet authenticated with the Authorization Server separately (e.g., not logged into Twitter), the Owner must do this first before being presented with this form. Once the Owner has have approved the grant, the Authorization Server of the third party constructs an Authorization Code and redirects the Owner back to the Client with that code.

With the Authorization Code now in the possession of the server-side application, a second HTTP request is made by that application back to the Authorization Server which includes the application's client ID, client secret, and the Authorization Code received. After all of these things have been verified by the Authorization Server, the application receives via this request an Access Token which can be used to perform requests against the desired resources and should be saved.

It is worth noting that the term Access Token can be a bit misleading. Generally speaking, the data returned from an Authorization Server which constitutes an Access Token is in reality a data structure that contains among other things two separate tokens. The first is the actual Access Token and the second is a Refresh Token. Typically speaking, Access Tokens do not work indefinitely but rather only are valid for a period of time before the authorization process has to happen again. Since repeatedly asking the user for permission to a resource would be cumbersome, the server-side application can transparently request a new Access Token using the Refresh Token without further user input. The lifetime of an Access Token can always been determined in these cases by examination of the expiration value contained within the token result and, if necessary, refreshing the Access Token using the Refresh Token.

Implicit Grants

Sometimes you would like to use OAuth in ways where a server-side backend is not available (such as a mobile application or JavaScript application). In these cases there is a separate workflow to grant authorization known as an Implicit Grant. Unlike the Authorization Code grant, which authenticates the requesting application's credentials, an Implicit grant relies solely on the URI of the application to verify identity. Implicit Grants also do not support Refresh Tokens. The flow of an Implicit Grant is illustrated in Figure 30.3.

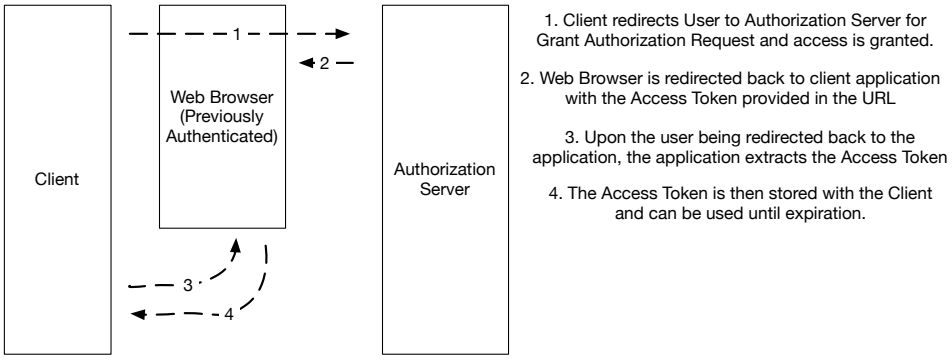

Figure 30.3.

For the sake of simplicity we have not included every minor step in our Implicit Grant workflow, while still maintaining an accurate representation of the steps. As with the Authorization Code grant, the Implicit Grant workflow begins with the user being redirected to the Authorization Server to grant the client authorization to the desired resources. However, where in an Authorization Code grant this is followed by the redirection of the user back to the client with an Authorization Code (requiring a second request to get an Access Token), in an Implicit Grant an Access Token is immediately returned. This Access Token can then be used by the client immediately. As previously mentioned however, such Access Tokens cannot be automatically refreshed by the use of a Refresh Token and must be re-authorized by the user upon every expiration.

In the development of web applications, and specifically when dealing with integrations with social media, Authorization Code and Implicit Grants are the two primary mechanisms available in the OAuth 2.0 specification you will work with. There are other workflows as well however they are out of scope for this chapter. In the next section, we will begin to put our knowledge of OAuth to use by implementing a simple Instagram web client.

Building an Instagram Web Client

The first Social Media integration we will explore is Instagram. Instagram implements an OAuth 2.0 authentication mechanism and we will be using an Authorization Code grant to allow users to use our simple Instagram feed browser. As is the case with any OAuth 2.0 implementation our application must first be registered with the party controlling the resources we want to access so we will also start there.

To get started integrating Instagram we must first create a developer account and register our new application. This can be done by going to http://instagram.com/developer and clicking on Manage Clients. Once in the Manage Client interface, a new client may be created by clicking Register New Client and filling out the brief form describing your application. From a technical perspective the most important portion of this form will be the list of valid redirect URIs, so special care should be taken. These URIs are the valid web addresses your client can request Instagram to redirect back to providing the Authorization Code described in the previous section. Typically in a web application there will only be one valid redirect URI per environment, however having one redirect URI for a production system and one for a development or staging system is not uncommon.

Once you have registered your client with Instagram you will be provided the client ID and client Secret values you will need to perform a successful Authorization Code grant. Hold on to these values as they will be foundational to building our application.

For our Instagram application we will have five separate PHP scripts. The first of these scripts will be a simple configuration file containing the necessary values we'll need to implement our OAuth client. Then we will have two scripts which will actually implement our OAuth client and two more scripts to provide the functionality of the Instagram client we are building. To facilitate the ability to perform various HTTP requests easily we will also be using the Guzzle HTTP Client package available via Composer and the Bootstrap CSS framework for easy UI creation.

Since it is a requirement for every other script we will construct, let's start with the simple PHP script to store the settings for our application and call it `settings.php`:

```
<?php

return [
    'client_id' => 'xxx',
    'client_secret' => 'xxx',
    'redirect_uri' => 'http://' . $_SERVER['HTTP_HOST'] . '/complete-
oauth.php',
    'scopes' => [
        'likes',
        'basic',
        'public_content'
    ]
];
```

This script contains all the values we will need to perform a successful OAuth authentication request against the Instagram servers. It is designed to be included by another PHP script using the `include_once` directive. It includes the Client ID and Client Secret values taken from the client we registered on http://instagram.com/developer, a redirect URI that will calculate based on the HTTP hostname which will be used to complete the OAuth procedure, and an array of scope values.

The scope key of our settings is an important array, as it defines what things we are asking the user to access of theirs when we perform our OAuth authentication. Fundamentally these can be thought of as unique string constants specific to the platform you are authenticating against. Each constant represents a different piece of functionality, data, or other resource you are requesting the user give you authorization to use in your application. Instagram provides a number of these scope values, all of which are documented in the Instagram API documentation. Please see https://www.instagram.com/developer/authorization/ for complete documentation of available scopes that a client may ask authorization to use. In our case, we are asking for the `likes`, `basic`, and `public_content` scopes, which will give us the ability to access basic account information, the ability to access any public profile information or media available to the user, and the ability to like/unlike content on the user's behalf.

The OAuth Login Page

The first step in our Instagram client is to present the user with a page to allow them to login via OAuth, granting us access to their Instagram account. To do this, we must construct a URL and attach it to a link on our page so that when it is clicked by the user it takes them to the appropriate page on the Instagram Authorization Server to grant us access to the resources we desire. Referring to the Instagram API documentation (https://www.instagram.com/developer/authentication/), we want to direct the user to https://api.instagram.com/oauth/authorize/ and pass the necessary details of the authorization

code request as `GET` parameters. Our login page and entrance point into our application therefore is as follows (named `index.php`):

```
<?php

require_once __DIR__ . '/../vendor/autoload.php';

$settings = include_once 'settings.php';

$authParams = [
    'client_id' => $settings['client_id'],
    'client_secret' => $settings['client_secret'],
    'response_type' => 'code',
    'redirect_uri' => $settings['redirect_uri'],
    'scope' => implode(' ', $settings['scopes'])
];

$loginUrl = 'https://api.instagram.com/oauth/authorize?' .
http_build_query($authParams);

?>
<html>
    <head>
        <title>PMWD - Chapter 30 - Instagram Demo</title>
        <link rel="stylesheet"
href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css"
integrity="sha384-
1q8mTJOASx8j1Au+a5WDVnPi2lkFfwwEAa8hDDdjZlpLegxhjVME1fgjWPGmkzs7"
crossorigin="anonymous">
        <link rel="stylesheet"
href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap-theme.min.css"
integrity="sha384-
fLW2N01lMqjakBkx3l/M9EahuwpSfeNvV63J5ezn3uZzapT0u7EYsXMjQV+0En5r"
crossorigin="anonymous">
        <script
src="//maxcdn.bootstrapcdn.com/bootstrap/3.3.6/js/bootstrap.min.js"
integrity="sha384-
0mSbJDEHialfmuBBQP6A4Qrprq5OVfW37PRR3j5ELqxss1yVqOtnepnHVP9aJ7xS"
crossorigin="anonymous"></script>
    </head>
    <body>
        <div class="container">
            <h1>PMWD - Chapter 30 (Instagram Demo)</h1>
            <div class="row">
                <div class="col-md-4 col-md-offset-4">
                    <div class="panel panel-default">
```

```
                        <div class="panel-heading">
                            <h3 class="panel-title">Login with Instagram</h3>
                        </div>
                        <div class="panel-body">
                            <a href="<?=$loginUrl?>" class="btn btn-block btn-
primary">Login with Instagram</a>
                        </div>
                    </div>
                </div>
            </div>
        </div>
    </body>
</html>
```

Ignoring the HTML used to actually render the login form, we start our PHP script by including `autoload.php`. This include statement is used to include the `autoload.php` script generated by Composer and provides our application access to in this case the Guzzle HTTP client. While not absolutely necessary for this specific page, for the sake of simplicity all scripts in this example will include it.

Next, we load our settings array from the `settings.php` script described earlier and assign it to the `$settings` variable using an `include_once` statement. Finally, we construct an array of key/value pairs representing the various `GET` parameters we need to pass along to Instagram when the user clicks Login and then append them as `GET` parameters to the Instagram Authorization URL using the `http_build_query()` PHP function. This useful function constructs the correct HTTP query string from an associative array that we can simply append to the Instagram Authorization URL.

Once we have built the authorization URL we need to start the OAuth authentication on Instagram's servers, we simply attach that URL to a Login with Instagram button that will begin the authentication process when clicked.

The next step in the OAuth process is handled by Instagram itself, which will first ask the user to log into their Instagram account then request authorization from them based on the scope(s) we requested access to in our initial authorization URL `GET` parameters. Assuming the user authorizes the request, Instagram will return the user to the specified redirect URI with a `GET` parameter code. This parameter is our Authorization Code needed to obtain an Access Token.

Completing The OAuth Authorization Grant

The next time we see the user after they click our Login button described above will be in the `complete-oauth.php` script, which is the script we have instructed Instagram to redirect the user back to after they have authorized our application's request. When the user returns Instagram will also have provided HTTP GET parameter code, used to retrieve the Access Token from Instagram as shown:

```
<?php
```

```
use GuzzleHttp\Client;
use GuzzleHttp\Exception\ClientException;
require_once __DIR__ . '/../vendor/autoload.php';

$settings = include_once 'settings.php';

if(!isset($_GET['code'])) {
    header("Location: index.php");
    exit;
}

$client = new Client();

try {
    $response = $client->post('https://api.instagram.com/oauth/access_token',
[
        'form_params' => [
            'client_id' => $settings['client_id'],
            'client_secret' => $settings['client_secret'],
            'grant_type' => 'authorization_code',
            'redirect_uri' => $settings['redirect_uri'],
            'code' => $_GET['code']
        ]
    ]);
} catch(ClientException $e) {
    if($e->getCode() == 400) {
        $errorResponse = json_decode($e->getResponse()->getBody(), true);
        die("Authentication Error: {$errorResponse['error_message']}");
    }

    throw $e;
}

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

$_SESSION['access_token'] = $result;

header("Location: feed.php");
exit;
```

After the previously explained setting up of our `$settings array` and including our Composer autoloader, the first thing we do is make sure that we have been provided the code

HTTP `GET` parameter as expected. Assuming we have, we then create a Guzzle HTTP Client instance and use it to perform a HTTP `POST` request back to the Instagram Authorization Server. During this request we transmit all of the necessary data for our OAuth request using the `form_params` option key. This includes our client ID and secret, the grant type, our redirection URI and finally the Authorization Code we were just given as part of this request.

Once the request occurs Instagram will authenticate all of these passed pieces of data to verify the authorization request and return an Access Token as a response. In the event the authorization fails Instagram will instead return a HTTP 400 error which translates into an exception thrown by our HTTP client. Thus, we catch this specific exception and display the error message for diagnostic purposes. Upon success of the request, we will be returned a JSON document containing an Access Token, Refresh Token, and other relevant metadata such as the expiration time for the Access Token. In a typical web application, this result would be stored in some sort of persistent storage associated with the user (i.e. database), but for our purposes we simply store it in the session for the user. Now that we have been issued valid credentials to interact with Instagram on behalf of the user, we can now redirect them to the entry point for our actual application and display a feed of content from their Instagram account. This is done in the `feed.php` script, which we will introduce next.

Displaying an Instagram Feed

The `feed.php` script is the entry point to the actual functionality provided by our example Instagram application. It assumes that the user has already granted us authorization to their account in the steps previously discussed. Being an example, our Instagram application accomplishes the following basic tasks:

- Load the most recent media available in the user's public feed, unless provided a tag to search against.
- Allow the user to like a given post.

As this script is a little more complicated than those previously discussed we will break it up into two parts. The first part is the logic we execute before we render the interface, and the second will be the logic needed to properly render our interface.

Let's take a look at the first part of our `feed.php` script now:

```
<?php

require_once __DIR__ . '/../vendor/autoload.php';

use GuzzleHttp\Client;

if(!isset($_SESSION['access_token']) || empty($_SESSION['access_token'])) {
    header("Location: index.php");
    exit;
```

```
}

$requestUri = "https://api.instagram.com/v1/users/self/media/recent";
$recentPhotos = [];
$tag = '';

if(isset($_GET['tagQuery']) && !empty($_GET['tagQuery'])) {
    $tag = urlencode($_GET['tagQuery']);
    $requestUri = "https://api.instagram.com/v1/tags/$tag/media/recent";
}

$client = new Client();

$response = $client->get($requestUri, [
    'query' => [
        'access_token' => $_SESSION['access_token']['access_token'],
        'count' => 50
    ]
]);

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

if(is_array($results)) {
    $recentPhotos = array_chunk($results['data'], 4);
}


?>
```

We start our script by performing a few sanity checks to make sure we have an Access Token available and redirecting the user back to the login page to get one if we don't. Next, we initialize a few variables. The `$requestUri` variable (which is the URI we will perform our request to get media from the Instagram API), the `$recentPhotos` variable (which will store the media from our request), and the `$tag` variable which will store the tag we are retrieving media for.

By default we want to return the most recent media available, unless a specific tag was specified. Thus, we initialize our `$requestUri` for that endpoint and then check to see if the `tagQuery` HTTP `GET` parameter was provided. If it wasn't we will perform the request for the most recent media and if it was we will perform a request against a different URI in the Instagram API to return only that media which has the specified tag. This works because for our purposes in either case the result from the Instagram API is similar enough we can build a single interface to render the result.

Once we have determined our API endpoint we perform a HTTP `GET` request against it and pass along the number of results to return (the count parameter) and our OAuth Access Token. Note when we received the Access Token from Instagram it was in the form of a JSON document containing not only the Access Token but also other values such as a Refresh Token. We only want to provide the actual Access Token itself back to Instagram to perform the request.

The result of the request will be a JSON document containing a collection of media posts that match the criteria of our request so we simply convert that document into a PHP array using `json_decode()`. We then break that array into chunks of 4, which will be helpful when rendering the media in a moment. Ultimately we have our `$recentPhotos` array which will containing a list of sub-arrays, each containing up to four actual media posts to render.

Next, we will take this array and render it using the Bootstrap CSS framework in an appealing way. Here's the second half of our `feed.php` script which does so:

```
<html>
    <head>
        <title>PMWD - Chapter 30 - Instagram Demo</title>
        <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css"
integrity="sha384-
1q8mTJOASx8j1Au+a5WDVnPi2lkFfwwEAa8hDDdjZlpLegxhjVME1fgjWPGmkzs7"
crossorigin="anonymous">
        <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap-
theme.min.css" integrity="sha384-
fLW2N01lMqjakBkx3l/M9EahuwpSfeNvV63J5ezn3uZzapT0u7EYsXMjQV+0En5r"
crossorigin="anonymous">
        <script
src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/js/bootstrap.min.js"
integrity="sha384-
0mSbJDEHialfmuBBQP6A4Qrprq5OVfW37PRR3j5ELqxss1yVqOtnepnHVP9aJ7xS"
crossorigin="anonymous"></script>
        <script src="//code.jquery.com/jquery-1.12.0.min.js"></script>

        <script>
                        $(document).ready(function() {
                                $('.like-button').on('click', function(e) {
                                        e.preventDefault();

                                        var media_id =
$(e.target).data('media-id');

                                        $.get('like.php?media_id=' + media_id,
function(data) {
                                                if(data.success) {
```

```
                                                    $(e.target).remove();
                                                }
                                            });
                                    });
                            });
        </script>
    </head>
    <body>
        <div class="container">
            <h1>Instagram Recent Photos</h1>
            <div class="row">
                <div class="col-md-12">
                    <form class="form-horizontal" method="GET"
action="feed.php">
                        <fieldset class="form-group">
                            <div class="col-xs-9 input-group">
                                <input type="text" class="form-control"
id="tagQuery" name="tagQuery" placeholder="Search for a tag...."
value="<?=$tag?>"/>
                                <span class="input-group-btn">
                                    <button type="submit" class="btn btn-
primary"><i class="glyphicon glyphicon-search"></i> Search</button>
                                </span>
                            </div>
                        </fieldset>
                    </form>
                </div>
            </div>
            <div class="row">
                <?php foreach($recentPhotos as $photoRow): ?>
                    <div class="row">
                        <?php foreach($photoRow as $photo): ?>
                        <div class="col-md-3">
                            <div class="card">
                                <div class="card-block">
                                    <h4 class="card-
title"><?=substr($photo['caption']['text'], 0, 30)?></h4>
                                    <h6 class="card-subtitle text-
muted"><?=substr($photo['caption']['text'], 30, 30)?></h6>
                                </div>
                                <img class="card-img-top"
src="<?=$photo['images']['thumbnail']['url']?>"
alt="<?=$photo['caption']['text']?>">
                                <div class="card-block">
```

```
                                    <?php foreach($photo['tags'] as $tag): ?>
                                        <a href="feed.php?tagQuery=<?=$tag?>"
class="card-link">#<?=$tag?></a>
                                    <?php endforeach?>
                                </div>
                                <div class="card-footer text-right">
                                    <?php if(!$photo['user_has_liked']): ?>
                                        <a data-media-id="<?=$photo['id']?>"
href="#" class="btn btn-xs btn-primary like-button"><i class="glyphicon
glyphicon-thumbs-up"></i> Like</a>
                                    <?php endif; ?>
                                </div>
                            </div>
                        </div>
                        <?php endforeach; ?>
                    </div>
                <?php endforeach; ?>
            </div>
        </div>
    </body>
</html>
```

The output of the `feed.php` script is the heart of our example application, which renders either a specific tag or the most recent photos from the given user's Instagram feed. This includes a search bar at the top of the page to search for a tag, and then using Bootstrap's grid UI framework, each photo result. In addition to rendering the photo itself and its caption we also render a Like button for any photo rendered the user has not liked yet. This like button is powered by a jQuery script that, when clicked, makes an AJAX call back to our application (specifically the `like.php` script we will discuss next) to flag the photo as "liked" using the Instagram API. If the user has previously liked the photo in question we simply do not display the Like button.

Liking Photos on Instagram

To implement the ability to like a photo on Instagram, as we just discussed in the proceeding section the example application uses an AJAX request to the `like.php` script of our application. This script is as shown:

```
<?php

require_once __DIR__ . '/../vendor/autoload.php';

use GuzzleHttp\Client;

header("Content-Type: application/json");
```

```
if(!isset($_SESSION['access_token']) || empty($_SESSION['access_token'])) {
    header("Location: index.php");
    exit;
}

if(!isset($_GET['media_id']) || empty($_GET['media_id'])) {
    echo json_encode([
        'success' => false
    ]);
    return;
}

$media_id = $_GET['media_id'];

$requestUri = "https://api.instagram.com/v1/media/{$media_id}/likes";

$client = new Client();

$response = $client->post($requestUri, [
    'form_params' => [
        'access_token' => $_SESSION['access_token']['access_token']
    ]
]);

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

echo json_encode([
    'success' => true
]);
```

At this point, the fundamentals which occur at the top of each script in our example application should be straightforward. Unlike the other scripts in our application, however, this one is designed to be called via AJAX and thus its output is JSON instead of HTML. In this case, we expect to be provided an Instagram media ID (the unique identifier Instagram assigns to every post) as an HTTP `GET` parameter and then use Instagram's API to like the photo on behalf of the user (via the Access Token for them). Upon success, we indicate it to the calling AJAX script with a simple return value in JSON format.

Conclusion

This concludes our exploration of OAuth through the Instagram API. However by no means have we provided a comprehensive catalog of all of the API calls available. We

leave it as an exercise for the reader to explore all of the APIs, using the techniques and tools we've provided to authenticate your application on behalf of the user and perform HTTP API requests. A complete documentation of the Instagram API is available in the Instagram Developer portal, http://www.instagram.com/developer/, which is also where you register your application for OAuth access.

While great lengths were taken to explain each step in the OAuth process, it is worth pointing out that in a professional application such efforts are not necessary. Social Media platforms such as Facebook and Google, for example, both provide PHP SDKs for interacting with their various web services that simplify the authentication process considerably. For those platforms that do not provide a specific SDK for PHP, such as Twitter, oftentimes there still have been written well-maintained open-source SDKs such as TwitterOAuth (http://twitteroauth.com/) that accomplish the same task.

While it is certainly important to understand how OAuth and social media API integrations work, it is strongly recommended that once you have mastered them you rely on the wide variety of pre-existing tools rather than rebuild them from scratch.