OAuth 2.0 Simple Example

Hey folks. I know that there are many of these pages out there that try to explain how OAuth 2.0 works, but I still spent the better part of the day figuring it all out so I thought that this document was warranted. This is also written for future me.

The best page that I found was Google's OpenID Connect page. It went about 90% of the way there although better examples and more details about the final JWT payload would have been good. I also used the JWT IO home page and their cool web-tool. More on that later.

If you want to see a working example, see the end of this page. Or you can:

Click here to login with Google

Background

Ok. Before we get into the details it's important to remember what we are trying to do here. OAuth allows us to use the authentication from a OAuth provider (like Google) instead of forcing a user to provide username and password into your site. It suffers from the issue that often when you log into Google (or other providers), they provide information like email-address to the sites you are trying to authenticate with, sigh, but I guess people are cool with that or something.

Registering Your Application

Before you start coding, you will need to register your application with the OAuth provider. In our case, we go to the following credentials list in Google. You will need a Google-id for this. On Google you do something like the following.

- Create an application
- Add content to the consent screen like title and logo
- Add credentials, specifically an OAuth 2.0 client ID
- Choose the "Web application" type and give it a name
- Enter the URIs that are allowed to be redirect-URIs

Google then gives you a client-id and secret that you will need to record and use in your web and server code. For this example, Google gave us:

```
Here is your client ID
139281538940-arh29cscgqk2vic01ackiphugqe6m2lr.apps.googleusercontent.com
Here is your client secret
nvDSpUI4R6iga0xfcvO7-V-s
```

Full OAuth 2.0 Authentication Process

The full OAuth process looks like:

1. User goes to your web-server to login and is presented with a username and password *and* a "Login with Google" button.
2. The user clicks on the button which takes them to Google with a callback-URL.
3. If the user has not done so already, Google presents some sort of "Do you want to login to this site..." type of information and asks for user confirmation.
4. The user either confirms or denies the auth request.
5. Google redirects the user back to your web-server with an auth-code using the callback-URL provided.
6. If approved, your web-server code needs to call directly to Google to validate the auth-code, not using the user's browser from a CGI script or controller.
7. Google returns an access-token and JWT token information.
8. You can then decode the JWT token.
9. The JWT token can then be verified against Google's published signatures, and you can do other things with the access-token over time.

Let's break these steps in the process down. We will be using Google API URLs in this example but I hope that Yahoo and other providers are at least similar.

User Goes to Your Web Server

You have some sort of login page on your web-server ("/login.html") which asks for a username and password. It also has a link or a clickable image to login using the user's Google account. The URL to Google is something like:

```
https://accounts.google.com/o/oauth2/v2/auth
```

It also includes the following URL parameters and is URL-encoded:

- client_id=... - Given to you when you register your application with the OAuth provider.
- response_type=code - I think this is a constant.
- scope=openid%20email - In Google's docs they say that this is a constant.
- redirect_uri=... - URI that you configure with Google which is an approved location for redirecting back to. This (like all params) must be URL encoded. See the example below.
- state=... - Secure string that your server makes which should be stored in the user's session. This gets returned in the callback after the user logs in using the OAuth provider and should be validated at that time.
- login_hint=email-address - optional, if already known
- openid.realm=256stuff.com - optional, your domain name, not sure how used
- hd=256stuff.com - optional, the hosted domain, not sure how different from realm. It should be your domain.

Here's an example URL with all of the parameters. It has some whitespace for readability:

```
https://accounts.google.com/o/oauth2/v2/auth
   ?client_id=139281538940-arh29cscgqk2vic01ackiphugqe6m2lr.apps.googleusercontent.com
   &response_type=code
   &scope=openid%20email
```

```
&redirect_uri=http%3A%2F%2F256stuff.com%2Fgray%2Fdocs%2Foauth2.0%2FcomeBack.cgi
&state=this-should-be-some-generated-secret-token
```

For more details, see Google's docs on the authentication URI parameters.

User Clicks on the Login With Google Button

Click.

Google Presents Some Sort of Confirmation Page

The OAuth provider will present the user with a page something like:

> 256stuff.com OAuth Sample would like to ...

It is important to note that if the user is already logged in and already confirmed the particular application, the OAuth provider will not show any confirmation page but will redirect immediately back with the approval information. Pretty cool. If you want to removed a connected application, you can do it in your Connected apps & sites section on the OAuth provider.

User Denies the Auth Request and Google Redirects To the redirect_uri

If you deny the request, the OAuth provider will redirect you back to the redirect URI provided in the request (redirect_uri). Here's the full redirect URI of the deny result with added whitespace:

```
http://256stuff.com/gray/docs/oauth2.0/comeBack.cgi
   ?error=access_denied
   &state=this-should-be-some-generated-secret-token#
```

Your server should then say something like:

> Well why did you click on the "Login With Google" button if you were just going to deny it fool?!

User Approves the Auth Request and Google Redirects To the redirect_uri

If you approve the request (good dog), the OAuth provider will redirect you back to the redirect URI provided in the request (redirect_uri). Here's something like the full redirect URI of the approval result with added whitespace:

```
http://256stuff.com/gray/docs/oauth2.0/comeBack.cgi
   ?state=this-should-be-some-generated-secret-token
   &code=4/Ei4UjaDc5rNnV2U8Ie8MJVFm-zIQs3ysoQ
   &authuser=0
   &prompt=consent
   &session_state=1c0e1b49853dba76fe6098d4feb1d4c..2062#
```

The fields in the request seem to be as follows:

- state - the state parameter that you generated and sent in the OAuth request URL
- code - authentication code that we will verify in the call below
- authuser - not sure about this
- prompt - whether or not the user actually confirmed the auth request or did they do it already ('content' or 'none')
- session_state - some sort of Google session information

If Approved, Your Web-Server Needs to Validate the Response

This is designed to not involve the user's browser since you are using the client-secret here. The request is made using a CGI script or in a controller.

After checking for approval, your web-server code should then validate that the state parameter from the redirect is the same that was stored in the user's session. Checking the state is important to protect against Cross-Site Request Forgery attacks. Your web-server code then needs to take the code parameter and call back to OAuth provider to validate it and turn it into an access-token. You web-server code should send a POST to the URL:

```
https://www.googleapis.com/oauth2/v4/token
```

The fields in the request are as follows:

- code=... - code parameter returned in the redirect URL params
- client_id=... - same as in the client-id above
- client_secret=... - client secret provided when you registered your app with the OAuth provider
- redirect_uri=... - same URI in the request above
- grant_type=authorization_code - standard, see the docs below

This is a POST request made to a secure URL by your web-server code and *not* through the user's browser. The request should look something like the following, again minus the whitespace in the params section. Notice that as always, the parameters must be URL encoded.

```
POST /oauth2/v4/token HTTP/1.1
Host: www.googleapis.com
Content-Type: application/x-www-form-urlencoded

code=4%2FEi4UjaDc5rNnV2U8Ie8MJVFm-zIQs3ysoQ
&client_id=139281538940-arh29cscgqk2vic01ackiphugqe6m2lr.apps.googleusercontent.com
&client_secret=nvDSpUI4R6iga0xfcvO7-V-s
&redirect_uri=http%3A%2F%2F256stuff.com%2Fgray%2Fdocs%2Foauth2.0%2FcomeBack.cgi
&grant_type=authorization_code
```

For more details, see Google's docs on the request parameters.

Google Returns Access-Token and JWT Information

If denied, a 404 or other status-code is returned. For example, this happens when you make 2 requests for the same code. If confirmed, the response to this API call should be a block of information in JSON format that either validates or denies the code parameter.


```
{
 "access_token": "ya29.QQIBibTwvKkE39hY8mdkT_mXZoRh7Ub9cK9hNsqrxem4QJ6sQa36VHfyuBe",
 "token_type": "Bearer",
 "expires_in": 3600,
 "id_token": "eyJhSUzI1Ni---Y0ZDQEifQ.eyJpc3ov2FjY29---iOjEyODYwMzZ9.C8gS1YGqjJ3K---g6e8bYJrEEDYiceu6KCLJKCH
}
```


The access-token and id-token have both been truncated in the above example. The id-token is especially long since it is an encoded block. The fields in the response are described as:

- access_token - A token that can be sent to a OAuth provider API
- token_type - Identifies the type of token returned. At this time, this field always has the value Bearer.
- expires_in - The remaining lifetime of the access token in seconds.
- id_token - A JWT that contains identity information about the user that is digitally signed by the OAuth provider.

Decode the JWT Token

The id-token is actually a large block of 3 Base64 encoded chunks separated by periods ('.'). For more documentation, I had to refer to the JWT IO code to see what they were doing.

The first block when decoded is a JSON block header that looks something like the following. I've added some whitespace.

```
{"alg":"RS256",
 "kid":"ce30d9f163852843c9a94ce1c1d711e4464d4391"}
```

The fields in this header block are described as:

- alg - Algorithm name used to sign the payload.
- kid - Key ID hint used to locate the proper key to check against the signature bytes.

The 2nd block from the id-token is a base64 encoded JSON payload that looks something like the following. I've added whitespace for readability:

```
{"iss":"https://accounts.google.com",
 "at_hash":"eXUxmt_D_iV52SrEg",
 "aud":"139281538940-arh29cscgqk2vic01ackiphugqe6m2lr.apps.googleusercontent.com",
 "sub":"113381163725",
 "email_verified":true,
 "azp":"139238940-arcscq2ic0ihugem2lr.apps.googleusercontent.com",
 "email":"some@email.address",
 "iat":1449282920,
 "exp":1449286520}
```

The fields in this payload block are probably variable by OAuth provider. For Google they are:

- iss - The Issuer Identifier of the response.
- at_hash - Access token hash which provides validation of the access token.
- aud - Identifies the audience that this ID token is intended for. This should be our client_id.
- sub - An identifier for the user, unique among all Google accounts and never reused.
- email_verified - True if the user's e-mail address has been verified; otherwise false.
- azp - The client_id of the authorized presenter.
- email - The user's email address.
- iat - The time the ID token was issued, represented in Unix time (integer seconds).
- exp - The time the ID token expires, represented in Unix time (integer seconds). This time should not have passed.

For more information about these fields, see Google's documentation of them.

The 3rd block is base64 encoded signature bytes that can be used to compare with the signature generated by the algorithm specified in the header against the full payload JSON string. You will need to download the keys from Google which can be found by looking at the jwks_uri field in the OpenID configuration document which right now (12/2015) points to v3 certs. Google's docs say that these keys are changed "infrequently" like "once per day" (sic) so they can be cached but not for long. For more info see Google's discovery document.

Try It Out

If you want to try it:

Click here to login with Google

It will say that you are going to be sharing your email with me. All I can do is promise that I'm not going to do anything with it aside from showing you the auth results. It won't show up in my web-logs either. If you are not convinced (and why would you be), you should create a throw-away gmail account.

Hope this helps someone else.