Authentication
OAuth 2.0 & OpenID Connect

Robert M. Vunabandi
Motivation ! ! ! What are we trying to achieve?

- We want users to use our app
- Users are complicated
- Users are lazy af: they don’t want to create new accounts
- We don’t want to create our own security system (that’s too complicated)
- We want users to trust our app
SOLUTION:

OAuth 2.0 and OpenID Connect
What is OAuth 2.0? What is OpenID Connect?

From https://oauth.net/2/: 

**Oauth 2.0** “is the industry-standard protocol for authorization.”

**Oauth 2.0** “focuses on client developer simplicity while providing specific authorization flows for web applications, desktop applications, mobile phones, and living room devices.”

From http://openid.net/connect/: 

**OpenID Connect** “is a simple identity layer on top of the OAuth 2.0 protocol. It allows Clients to verify the identity of the End-User based on the authentication performed by an Authorization Server, as well as to obtain basic profile information about the End-User in an interoperable and REST-like manner.”
So, Basically

Oauth 2.0 = authorization

OpenID Connect = verification

Oauth 2.0 + OpenID Connect = auth-entification
THESE GUYS ARE JUST PROTOCOLS!

NOT FRAMEWORKS, NOT APIS, NOT LIBRARIES, NOT DEPENDENCIES, NOT APPLICATIONS, NOT SERVERS, NOT DATABASES, NOT LANGUAGES, NOT ALGORITHMS, . . .

THEY ARE JUST AGREED-UPON AND ENFORCED GUIDELINES FOR AUTHENTICATION
What is Happening When Using Them?

During a transaction that abides by OAuth 2.0 and OpenID Connect protocols, we have:

- **User (i.e. ResourceOwner):** The person using your app. This person owns the resource that your server (i.e. Client) needs.
- **Client:** Your application’s server. You application’s server gets a **ClientID** and **ClientSecret** from the **AuthorizationServer**.
- **AuthorizationServer:** The server that provides and verifies the given **ClientID** and **ClientSecret**.
- **ResourceServer:** The server that provides the resources that you need. This is often the same as the **AuthorizationServer**.
Grant Types

They specify what type of access you want to get from the resource server.

There are multiple grant types:

- AuthorizationCode
- Implicit Grant
- Resource Owner Credentials
- Client Credentials
- Refresh Token

We focus on AuthorizationCode with refreshToken
AuthorizationCode Grant Type and RefreshToken

This allows us to gain access to a very specific subset of data among the entire user’s data stored in the resource server’s database. The resource server will only give us what we get access to based on the AuthorizationCode Grant Type.

How? By using a key. This key is called AccessToken

User can revoke the AccessToken (i.e. make it useless) anytime. Also, this AccessToken expires (becomes useless) after a very specific amount of time or on a very specific date and time.

The RefreshToken is a revocable but unexpirable key generator (AccessToken generator). When user revokes the Client’s access, the RefreshToken becomes useless.
The **SIMPLE VERSION** of the **AuthorizationCode Workflow**

(this will actually be kinda wrong...)
1. 'Log user in!' through `/auth/resource`
2. redirect to authorization page.
3. requests authorization page
4. delivers login then authorization page
5. User logs in then authorizes app
6. returns a AccessToken specific to Client and User
7. sends AccessToken
8. sends AccessToken
9. sends UserInformation
10. '/auth/resource/callback' Saves needed information into database, 'logs user in', renders the redirected page

**this is actually kinda wrong guys**
The ACTUAL VERSION of the AuthorizationCode Workflow
1. '/auth/resource'
2. redirect to authorization with state and ClientID
3. requests authorization page
4. delivers login then authorization page
5. User logs in then authorizes app
6. returns a new AuthorizationCode and state specific to Client and User
7. sends AuthorizationCode and state
8. sends AuthorizationCode, ClientID, and ClientSecret
9. returns AccessToken specific to User
10. sends AccessToken
11. sends UserInformation
12. '/auth/resource/callback'
Saves needed information into database, 'logs user in', renders the redirected page
That is roughly what happens!
Oauth Workflow with Facebook
1. '/auth/facebook'
2. redirect to authorization with state and ClientID
3. requests authorization page
4. delivers login then authorization page
5. User logs in then authorizes app
6. returns a new AuthorizationCode and state specific to Client and User
7. sends AuthorizationCode and state
8. sends AuthorizationCode, ClientID, and ClientSecret
9. returns AccessToken specific to User
10. sends AccessToken
11. sends UserInformation
12. '/auth/resource/callback'
   Saves needed information into database, 'logs user in', renders the redirected page
Oauth Workflow with MIT OpenID Connect
User

Client

Authorization Server

Resource Server

1. `/auth/oidc`
2. redirect to authorization with state and ClientID
3. requests authorization page
4. delivers login then authorization page
5. User logs in then authorizes app
6. returns a new AuthorizationCode and state specific to Client and User
7. sends AuthorizationCode and state
8. sends AuthorizationCode, ClientID, and ClientSecret
9. returns AccessToken specific to User
10. sends AccessToken
11. sends UserInformation
12. `/auth/oidc/callback`
   Saves needed information into database, ‘logs user in’, renders the redirected page
What about the refreshToken Workflow?
1. `/auth/resource`

2. redirect to authorization with state and ClientID

3. requests authorization page

4. delivers login then authorization page

5. User logs in then authorizes app

6. returns a new AuthorizationCode and state specific to Client and User

7. sends AuthorizationCode and state

8. sends AuthorizationCode, ClientID, and ClientSecret

9. returns AccessToken and RefreshToken specific to User

10. sends AccessToken

11. sends UserInformation

12. `/auth/resource/callback`
   Saves needed information into database, 'logs user in', renders the redirected page
Demo

(i.e. sorta seeing this in Code)
Skeleton Code and Guide:
go.6148.io/mitopenid