Learning Objectives: OAuth

Learners will be able to…

- **understand the basics of OAuth 2.0**
- **explain the authentication process including token verification**
- **use a third-party API that requires OAuth (Github)**
- **implement single sign on using OpenID Connect (Google)**
- **list the basic requirements for a safe web API**

info

Make Sure You Know

This assignment assumes a basic knowledge of JavaScript, HTML and CSS.

Limitations

We will not explain simple programming concepts, it is assumed that the user is familiar with these concepts.

Security

SSL/TLS

If data is not encrypted then in certain situations (for example on a public wi-fi network) packets can be intercepted, data viewed and passwords compromised. These are called “man-in-the-middle” attacks because someone is intercepting the data as it passes from the client to the server. For non-public information, HTTPS should be used to provide an extra layer of security where all data exchanged between a client and a server is encrypted.

SSL (Secure Socket Layer) was the original security protocol used for HTTPS but it had many vulnerabilities and was succeeded by TLS (Transport Layer Security) in 1999. TLS 1.3 is the version currently used. It’s often referred to as SSL/TLS because TLS is very similar to SSL.

How does the interaction work?

1. The client initiates a conversation, known as a handshake process, with a server by sending a “hello message”, a list of TLS versions it supports (currently can be 1.2 and 1.3) and a cryptographic algorithm.
2. The server responds with it’s own “hello message” that includes the selected TLS algorithm and it’s digital certificate along with a public key.
3. When the client receives this information from the server it verifies server’s digital certificate and sends the server a secret key encrypted using the public key provided by the server.
4. The server decrypts the secret key using its private key. After this step the server and client have a way to exchange encrypted data.

HTTPS vulnerabilities

- There have been incidences where the root SSL (Secure Socket Layer) certificates were swapped by hackers and that gave them the ability to decrypt communication between the client and the server.
- In the past some companies required their workers to install certificates that would allow corporate data inspection of all Internet traffic flowing in and out of the organization.
- There are governments in some countries that have required that local Internet Service Providers make it mandatory for their customers to install government issued root certificates. These certificates make it possible for the government to decrypt all their citizens’ Internet traffic.

Defense in Depth, Layered security

Defense in Depth and **Layered Security** refer to putting in place a security system that is layered and provides multiple overlapping protection mechanisms. The following are some of the systems used.

- Firewalls - Web Application Firewalls (WAF) use knowledge about known attack methods and IP denylists to protect against known bad agents.
- Detection - It's important to detect intruders that have managed to breach early defenses. Some methods for doing so are Intrusion Detection Systems (IDS) and honeypots. Honeypots are used to detect malware and analyze the methods it is using.
- Authentication and access control - These are the most important parts of ensuring API security. The quality of the authentication method is the key here and we will cover OAuth 2.0, the industry standard.

OAuth 2.0

The need for a more secure system - OAuth 2.0

Basic authentication requires sending the login and password with each request. This system is vulnerable because it provides hackers with more opportunities to extract a user's credentials. The solution to this is to minimize the numbers of the requests where credentials are sent.

With OAuth 2 (short for Open Authorization) authentication of user credentials is conducted by an authorization server and the client gets a token that specifies their authorization status. Using tokens means that credentials do not need to be passed each time a request is made. Having an authorization server separated from other components allows the application of more secure layers to it.

OAuth 2 allows Single Sign On (SSO), which means the user does not need to share their password at all, a third-party authorization server may be used (e.g., Google or Facebook).

OA
Roles
- **Resou
Owner**
The use
enterpr
owns th
resourc

- **Resou
Server**
The ser
hosting
protect
resourc
returni
resourc
when th
request
accomp
by an a
tokens.

- **Client**
The clie
the

applica
that wa
access t
user's
resourc
client c
server,
desktop
another
device.

-
Author
Server
The ser
that is
respons
for
authen
the reso
owner
obtaini
authori
and the
issuing
access t

The OAuth work flow

This is the basic work flow for the protocol flow, there may be variations.

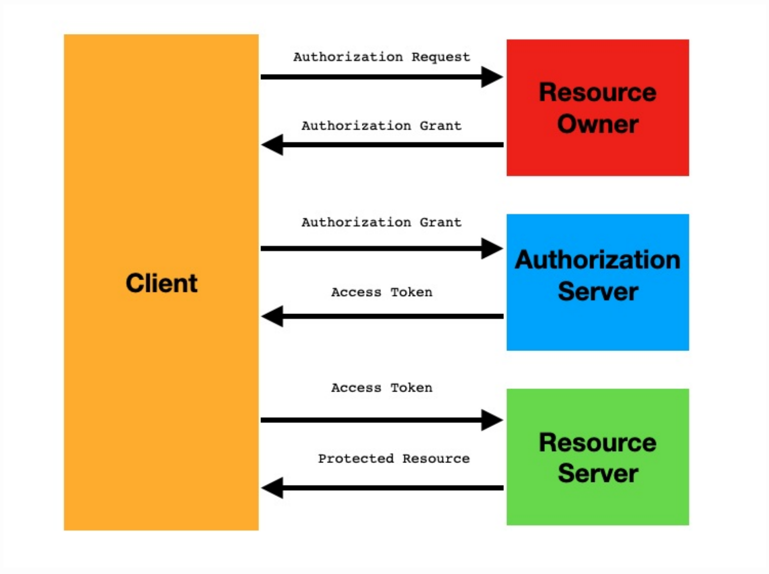


Diagram showing the exchanges in oauth

Here are the steps depicted above:

1. The client requests authorization to access resources for a user.
2. If the request is authorized, an authorization grant is sent back.
3. The client requests an access token from the authorization server by presenting authentication of its own identity, and the authorization grant.
4. If application identity authentication is successful and the authorization grant is valid, an access token is issued. Authorization is complete.
5. The application can now request a resource from the resource server by presenting the access token for authentication.
6. Provided that the access token is valid, the requested resource is returned.

OAuth Example - GitHub

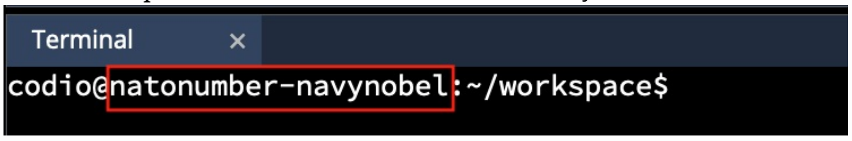
info

Installing the node modules

Install the node packages required for this example, run the commands below in the terminal.

```
cd examples/github-client
npm i
```

Follow these steps to try out the OAuth - GitHub example

1. Create the `redirectUri` (your Codio URL) by replacing `trapezejerome-vitalconvert` with the two words you see in your terminal between the `@` and `:`. You'll need this for the OAuth App.
   In the example below it would be `natonumber-navynobel`.
   

   ```
   codio@natonumber-navynobel:~/workspace$
   ```

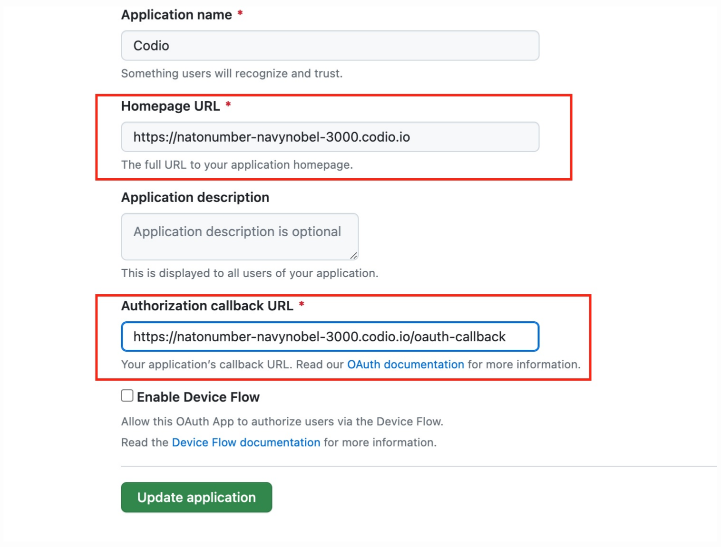
2. Create an OAuth App (if you don't have a Github account, you will have to create one first).

info

## OAuth App

Edit the following fields in the App, substituting your information as you did for the `redirectUri` in step 1.
Homepage URL:
`https://natonumber-navynobel-3000.codio.io`

Authorization Callback URL:
`https://natonumber-navynobel-3000.codio.io/oauth-callback`



Application name *

Codio

Something users will recognize and trust.

Homepage URL *

https://natonumber-navynobel-3000.codio.io

The full URL to your application homepage.

Application description

Application description is optional

This is displayed to all users of your application.

Authorization callback URL *

https://natonumber-navynobel-3000.codio.io/oauth-callback

Your application's callback URL. Read our OAuth documentation for more information.

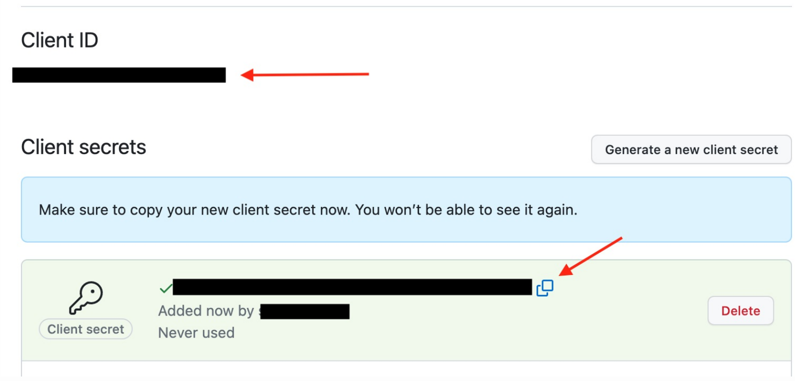
Enable Device Flow

Allow this OAuth App to authorize users via the Device Flow.

Read the Device Flow documentation for more information.

Edit the the Homepage URL and Authorization callback URL

3. Once you have completed creating your OAuth App, you can find it again at this URL, when you are logged into your account.
   - Copy your **Client ID** and **Client Secret** information from the GitHub App into app.js.



Client ID

[redacted]

Client secrets

Make sure to copy your new client secret now. You won't be able to see it again.

✓ [redacted]

Added now by [redacted]

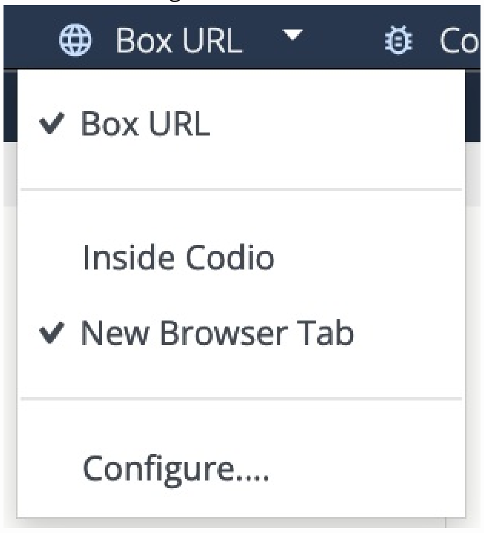
Never used

Starting the App

To start the app enter the following in the terminal, you must be in the examples/github-client directory. To stop the app use Ctrl-C in the terminal.

```
nodejs app.js
```

- Click the triangle next to the Box URL tab and select **New Browser Tab**.



- Click the **Box URL** menu item to connect and you will see that you have correctly connected to your OAuth App.
- In the new tab click **Log in using GitHub** and you will see that you have authenticated properly and can view information about your GitHub account.
- Note that the authorization token expires after 5 minutes, if you click on **BoxURL** after that interval you will no longer connect.

Viewing the code

The file is the main page and handles things once there is an authentication token, the file contains the logic to handle the authentication process, and the file is where all the setup is done.

OAuth Example - Google

info

Installing the node modules

Install the node packages required for this example, run the commands below in the terminal.

```
cd examples/google-openid-connect
npm i
```

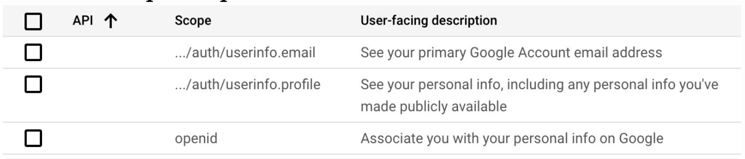
Follow these steps to try out the OAuth - Google example

1. Copy the `redirectUri` and paste it You may need to scroll down in the second file.
2. Setup OAuth 2.0 credentials, go to this website: https://console.developers.google.com/apis/credentials and follow the instructions below.

info

## Setup OAuth 2.0 credentials

- Create a **New Project**, give it a name and leave the organization field blank
- Select **OAuth Consent screen**.
  - Choose External for **User Type** and click **Create**.
- In **App Information**:
  - Enter App Name and your email address.
- Under **Authorized domain**:
  - Click **Add Domain** and Enter `codio.io`, your email again and then **Save and Continue**.
- Under **Developer contact information**
  - Enter your email address.
- On the next screen click **Add or Remove Scopes**.
  - Select the Scopes depicted below:



☐	API ↑	Scope	User-facing description
☐		.../auth/userinfo.email	See your primary Google Account email address
☐		.../auth/userinfo.profile	See your personal info, including any personal info you've made publicly available
☐		openid	Associate you with your personal info on Google

  - Click **Update** and **Save AND Continue**.
- Add **Test users**, you can just use your own email account. Note: You have to click **Add** twice.
- Click **Save and Continue**
- Click **Back TO Dashboard**
- Select **Credentials** from the left and then +**Create Credentials**.
- Create an **OAuth client ID**.
  - Application type should be **Web Application**.
- For the **Authorized JavaScript origins** enter your URL, it will be a variation of this:
  - `https://natonumber-navynobel-3000.codio.io`
- For the **Authorized redirect URI's** it will be the same one you edited in app.js earlier, a variation of this: `https://natonumber-navynobel-3000.codio.io/oauth-callback`
- Click **Create**.

- Now you can copy the **Client** and **Secret** ids to enter .

Starting the App

To start the app enter the following in the terminal, you must be in the examples/google-openid-connect directory. To stop the app use Ctrl-C in the terminal.

```
nodejs app.js
```

Click on **Box URL** in the menu and you will see your Google credentials.

Viewing the code

The file is the main page and handles things once there is an authentication token, the file contains the logic to handle the authentication process, the file contains a function that checks if the current token is valid, and the file is where all the setup is done.

▼ **What is a nonce?**

A nonce is a randomly generated string of characters.

Authorization Grants and Access Tokens

Authorization Grant Work Flows

An authorization grant is a credential representing the authorization given by the resource owner to the client (third-party application) that can be exchanged for an **Access Token**.

Here are four types of authorization grants in OAuth 2.0:

Authorization code grant: This grant type is obtained by using an authorization server to negotiate access between a client and a resource owner. The authorization server authenticates the resource owner and obtains authorization. The advantage of the Authorization code grant is that the access token is passed directly to the client and not through the resource owner. Credentials are only passed to and from the authorization server and the client and the resource owner never exchange them.

Implicit grant: This is a simplified version of the **Authorization code grant** type grant and is optimized for in-browser usage with JavaScript or a similar programming language. In this flow, the client is issued an access token directly. The authorization server does not authenticate the client, that is done by the resource owner. This method is more efficient as it doesn't require as many trips to obtain an access token but it is less secure and therefore not recommended - the **Authorization code grant** is preferred.

Resource owner password credentials grant: In this grant type the resource owner provides their credentials directly to the application and an access token is returned. This work flow should only be used with applications that are trusted because the user is supplying their password directly. **This is a legacy grant type and is not recommended because of the lack of security for the credentials.**

Client credentials grant: This grant work flow is most often used when the client is also the resource owner or has already been authoriz**ed by an authorization server. An example of this might be an application that is using its own API to make changes to its settings.

Device Grant: This grant provides devices without a browser, or with limited input capability, with a way to obtain an access token. This type of grant is commonly used by Apple TV Apps and other devices that stream video.

Access Tokens

Access tokens are passed as credentials in requests to access protected resources. The tokens specify the scope of access as well as the duration of the permissions for that access. The resource server enforces the limitations specified by the token.

Refresh Tokens

Refresh tokens are credentials used to obtain **access tokens** and are distributed by the authorization server. When an **access token** expires or becomes invalid a **refresh token** can be used to get a new one.