OAuth 2.0

OAuth is an **authorization protocol**. It is commonly used to get user authorization so that an app (the codebase using the API) has permission to access the user's data. In other words, it allows the app code (as a "client") to access user data from another site (the server or "resource server"). As this is a protocol, OAuth simply outlines a ***way*** to authorize one site to get information from another. By itself, OAuth is not a set piece of code that will do something for you, but APIs use OAuth as a standard to implement authorization. (The user is the "resource owner" because he or she owns the data—the account data—on the resource server.)

The resource server will implement code which follows this protocol and returns data as outlined in the protocol. On your end (the "client"), you will implement code which makes requests and processes result data as outlined in the protocol.

Note that OAuth is **not an *authentication* protocol**. Authentication and authorization are not the same thing. Authorization only refers to giving a client app the permission to access data. Authentication is about proving that the user is who he says he is. (Both may occur in the process of granting permission, but the differences between both should be understood.)

Authorization vs Authentication

The following are a few good links which try to explain the difference between **authorization** and **authentication**:

- https://security.stackexchange.com/questions/119225/oauth2-and-authentication
- http://stackoverflow.com/questions/33702826/oauth-authorization-vs-authentication

Read about the OAuth 2.0 protocol

You can read about the OAuth 2.0 spec on the official website (https://oauth.net/2/), but you will probably find the following link much easier to understand. The link leads to the part pertaining to websites:

https://aaronparecki.com/oauth-2-simplified/#web-server-apps

The basic parts of the protocol, as a summary of the above link, are as follows:

1. The user will be redirected to the resource server, sending the following parameters as well:
    - **response_type**: code
      This means that you expect to receive an authorization code.
    - **client_id**: *[your_client_id]*
      This is your API key for the service you want to be authorized with.
    - **redirect_uri**: *[url_in_your_site]*

When authorization is complete, the server/API server (e.g. LinkedIn site) redirects back to a page in your site (this URL) for the next steps of authorization.

- **scope**: *[value(s)]*
  One or more values can be added. The value names are specified by the API you're connecting to. These values indicate which parts of the user's account the site wants to access.

- **state**: *[random_alphanumeric_string]*
  This is a random alphanumeric string of your choosing which you will use later to verify after receiving authorization code. A "state" value is returned after this request. You need to check that the before "state" value equals the after "state" value. Different values are indication of a possible cross-site request forgery (CSRF) attack.

2. The user will log in to the resource server and "Allow" or "Deny" the request for account access.
3. The user is redirected back to the client site (to the **redirect_uri** specified above).
   a. If the user granted access, you should receive the following values:
      - **code**: *[the_returned_authorization_code]*
      - **state**: *[your_state_value]*
   b. If the user denied access, you'll probably receive some sort of error values (see the API for the exact values, if any).
4. Next request the access token. The client should POST the request, passing the following values:
   - **grant_type**: authorization_code
   - **code**: *[the_authorization_code_you_received]*
   - **redirect_uri**: *[same_redirect_uri_in_original_link]*
   - **client_id**: *[your_client_id]*
   - **client_secret**: *[your_client_secret]*
5. If successful, you should receive the following values:
   - **access_token**
   - **expires_in**: *[the_time_in_secs_when_the_token_expires]*

For security, if using Node.js store the client ID and secret (and API keys) as environment variables.

A basic example with OAuth (extra)

Note that for most APIs nowadays, it's better if you use HTTPS (for localhost it doesn't matter but if online, you **do** need to use HTTPS).

This is a very basic, incomplete example based on the previous Trakt example. The code is extremely simple without too many checks, but is just an illustration for how to connect to Trakt while giving you a rough skeleton so that you can convert this into your own class for easier integration. Note that OAuth is only required if any of the API calls needs to be user-specific. Some API calls are not user-specific and do not require the OAuth access token for the request (e.g. list trending shows).

Note that it's better to get a general idea about the OAuth steps before taking a look at the code. In practice, you should check to see if the API has libraries available to make API integration easier, but it's still useful to understand how you'd go about it from scratch in the event there are no libraries available.

We'll be looking at the following code which was written to make some basic OAuth-related API calls. This was originally all written in *index.js* to make it easier to read in one go, but you can split the functions into a ***/modules/trakt/oauth.js*** file.

```
const qs = require("querystring");
const traktUrl = "https://api.trakt.tv";

function startAuthorizing() {
  var params = {
    response_type: "code",
    client_id: process.env.TRAKT_CLIENT_ID,
    redirect_uri: process.env.TRAKT_REDIRECT_URI,
    state: process.env.TRAKT_STATE
  };
  let formattedParams = qs.stringify(params);
  let url = `${traktUrl}/oauth/authorize?${formattedParams}`;
  return url; //return Trakt's authorizing page URL
}

async function getAccessToken(code) {
  var params = {
    code: code,
    client_id: process.env.TRAKT_CLIENT_ID,
    client_secret: process.env.TRAKT_CLIENT_SECRET,
    redirect_uri: process.env.TRAKT_REDIRECT_URI,
    grant_type: "authorization_code"
  };
  let response = await fetch(
    `${traktUrl}/oauth/token`,
    {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify(params)
    }
  );
  accessToken = await response.json();
  return accessToken;
}

module.exports = {
  startAuthorizing,
  getAccessToken
};
```

Now for the page routes in ***index.js*** using the above functions:

```
let accessToken;

…

app.get("/page-requiring-oauth", async (request, response) => {
```

```
  if (accessToken) {
    //replace <user-slug> with your slug username (from your general account settings)
    let userData = await trakt.getUserData("<user-slug>", accessToken); //function to display
secured data (unimplemented)
    console.log(userData);
  } else {
    let authUrl = traktAuth.startAuthorizing(); //if no accessToken, kickstart the OAuth
process
    response.redirect(authUrl);
  }
});
app.get("/authorize", async (request, response) => {
  if (request.query.code && !accessToken) {
    token = await traktAuth.getAccessToken(request.query.code); //if there is code, get access
token
    if (token.access_token) {
      accessToken = token.access_token;
      response.redirect("/page-requiring-oauth");
    }
  } else {
    //kickstart authorizing
    console.log("start again");
  }
});
```

Note that "<user-slug>" has been coloured a bright red. This is a string which needs to be changed to your slug value.

There is a global variable in *index.js*, **`accessToken`**, used to store the access token when you get it.

There are functions which handle the OAuth flow:

- **`startAuthorizing()`**
  - Return the authorization page/login URL to get the user authorization. Redirecting to the returned URL kickstarts the whole OAuth process.
- **`getAccessToken(code)`**
  - Once you receive the authorization code, you need to send it to get the access token required for user-based API requests.

Let's break this down further.

For OAuth authentication (also see ***OAuth2*** PDF), you'll need to first request an authorization code, which is required to request an access token. In the documentation, requesting an authorization code has the following requirements:

GET
https://api.trakt.tv/oauth/authorize?response_type=code&client_id=<client_id>&redirect_uri=<redirect_uri>&state=<state_value>

Parameters:

- response_type (should always be "code")
- client_id
- redirect_uri

- state

The above means that you need to make a GET request to the given URL with the listed parameters (i.e. go to the page denoted by the URL). The **state** is a **random arbitrary alphanumeric value** used to check if there were man-in-the-middle attacks. If the values are correct, making the request should lead to Trakt's login page which asks if you want to authorize the app to access your info. Allowing the access will redirect back to your site's page with the authorization code and previously sent "state" returned in the query string. Using the authorization code, you can then request an access token. Note that in the code above, there was no check to see if the sent state value and received state value matched. You can add this yourself.

You can write this as a function. In *oauth.js*:

```
function startAuthorizing() {
  var params = {
    response_type: "code",
    client_id: process.env.TRAKT_CLIENT_ID,
    redirect_uri: process.env.TRAKT_REDIRECT_URI,
    state: process.env.TRAKT_STATE
  };
  let formattedParams = qs.stringify(params);
  let url = `${traktUrl}/oauth/authorize?${formattedParams}`;
  return url; //return Trakt's authorizing page URL
}
```

Note that we're using an object to store query string parameters then using the `querystring` module's `stringify()` method to convert to a query string. If you write it out manually without using `stringify()`, that would still be fine. One reason for using `stringify()` is to help avoid typos (it's easier to make a mistake if the string is long).

We need to go to Trakt's login for user authorization. The URL we returned has all the request parameters in the query string, so we can just use a redirect (a regular page load is essentially a GET).

In *index.js*:

```
app.get("/page-requiring-oauth", async (request, response) => {
  if (accessToken) {
    let userData = await trakt.getUserData("jarvis-8529324", accessToken); //function to
display secured data
    console.log(userData);
  } else {
    let authUrl = traktAuth.startAuthorizing(); //if no accessToken, kickstart the OAuth
process
    response.redirect(authUrl);
  }
});
```

In the documentation, to request an access token you need to make a request to:

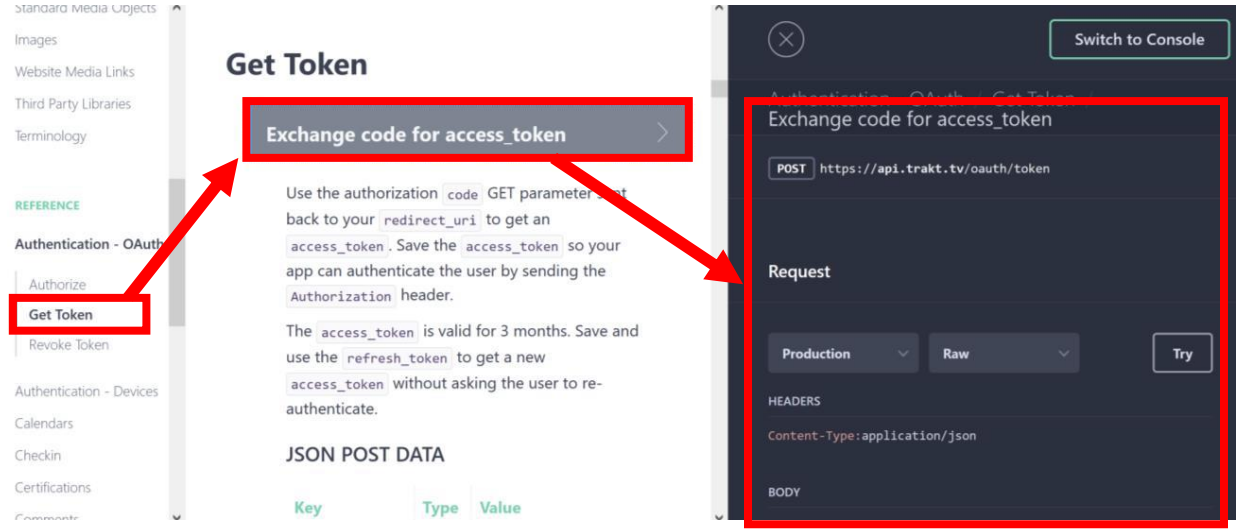
POST
https://api.trakt.tv/oauth/token

Headers:
• Content-Type: application/json

Body: (in JSON format)

Key	Type	Value
`code *`	string	Authorization code.
`client_id *`	string	Get this from your app settings.
`client_secret *`	string	Get this from your app settings.
`redirect_uri *`	string	URI specified in your app settings.
`grant_type *`	string	`authorization_code`

Body means that this needs to be passed as your HTTP request's content.



In the API documentation (at least in Trakt's API), after clicking on the API request reference, you have to click on the second box to view the info in the third panel. The third panel contains all the information you need to be able to make an API request. It'll list what type of HTTP request you need to make (e.g. GET, POST, PUT) along with any headers and request content (body) you should or *can* add to your request.

Let's take a look at the function that we've written for this:

```
async function getAccessToken(code) {
  var params = {
    code: code,
    client_id: process.env.TRAKT_CLIENT_ID,
    client_secret: process.env.TRAKT_CLIENT_SECRET,
    redirect_uri: process.env.TRAKT_REDIRECT_URI,
```

```
    grant_type: "authorization_code"
  };
  let response = await fetch(
    `${traktUrl}/oauth/token`,
    {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify(params)
    }
  );
  accessToken = await response.json();
  return accessToken;
}
```

The highlighted text is the part which actually creates and executes the request. Here, we're using Fetch again to make the HTTP request.

In the documentation, we can see that the body content of the request should be in JSON format, so we need to make sure that we don't forget the **`Content-Type`** header (in the documentation).
We also use a JSON object (`params`) to store body data then set the body data using the `body` key (bolded line in the code above). We need to use JSON.stringify() so that we don't get parsing errors due to newline/tab characters.

Upon success, store the access token so we can use it in subsequent calls. I've kept it simple here by using just a basic variable.

```
app.get("/authorize", async (request, response) => {
  if (request.query.code && !accessToken) {
    token = await traktAuth.getAccessToken(request.query.code); //if there is code, get access
token
    if (token.access_token) {
      accessToken = token.access_token;
      response.redirect("/page-requiring-oauth");
    }
  } else {
    //kickstart authorizing
    console.log("start again");
  }
});
```

Once you've got the access token, you can use it for making requests requiring user authorization. For example, if you want your app to retrieve a user's private profile (to display to that user in your app), you can implement getting the user profile in ***trakt/api.js***:

```
async function getUserData(slug, accessToken) {
  let reqUrl = `${trakt}/users/${slug}`;
  var response = await fetch(
    reqUrl,
    {
      method: "GET",
      headers: {
```

```
        "Content-Type": "application/json",
        "trakt-api-version": "2",
        "trakt-api-key": process.env.TRAKT_CLIENT_ID,
        "Authorization": `Bearer ${accessToken}`,
      }
    }
  );
  return await response.json();
}
```

For the "Authorization" header, there must be a space between the word "Bearer" and the access token value. This is the required format for a bearer token header. Without the space, it will not work.

To test this all out, you can take a look at the example with oauth (but you must change the values in the .env file and the slug in ***index.js***.

For more persistent storage across page loads, you can use session variables (e.g. you'd use session variables for a login): https://www.section.io/engineering-education/session-management-in-nodejs-using-expressjs-and-express-session/