Implementing Riot Sign On (web tutorial)

This tutorial will get you through making first contact with Riot Sign On, including serving a Sign In link, authenticating the user, and processing the user's tokens, but you'll need a few things before continuing:

- Riot Sign On — hosted at https://auth.riotgames.com. Use this to authenticate in the tutorial.
    - **/authorize** — endpoint for obtaining an authorization code
    - **/token** — endpoint to exchange authorization codes for access, identity, and refresh tokens
    - **/jwks.json** — endpoint to grab JSON Web Keys for verifying the authenticity of access and identity tokens
    - (optional) **/userinfo** — endpoint to use your access token to obtain user information
- For the tutorial, add **127.0.0.1 local.example.com** to your /etc/hosts file
- Register a client — You will need it so you have credentials to use with Riot Sign On
    - the client will need to be registered to allow **http://local.example.com:3000/oauth2-callback** as a **redirect_uri**.
- Service must be secured with **https**

This tutorial will be giving most examples as Node.js code samples, and assumes that you have access to standard npm libraries, such as request.

Required details for client registration

Fill in the blanks or amend where necessary. Client name, logo uri, privacy policy uri and terms of service uri can all be localized, please provide localizations in the form <field>#<locale>, e.g. privacy policy url#de_DE: <url>

Riot supported locales: cs_CZ, de_DE, el_GR, en_AU, en_GB, en_PL, en_US, es_AR, es_ES, es_MX, fr_FR, hu_HU, it_IT, ja_JP, pl_PL, pt_BR, ro_RO, ru_RU, tr_TR.

contacts: [contact email addresses]
Client name: client_name
Logo uri: Require 60px by 60px logo image
Privacy policy uri: link to privacy policy
Terms of service uri:link to terms of service
Grant types: ["authorization_code", "refresh_token"]
Redirect uris: ["]
Post logout redirect uris: [""]
Preferred client id:
Token endpoint auth method: client_secret_jwt or client_secret_basic ?

Understanding the Authorization Code Flow

Send the player to **Riot Sign On**, an authentication service provided by Riot Games, with a specially crafted link. The player logs in, and if they've provided correct credentials to Riot SIgn On, they are redirected to **redirect_uri** that we specified in our initial link with an authorization **code** affixed to the url.

This **code** can be sent to the **token** endpoint, and we will receive Access, Identity, and Refresh tokens. The Access token can be used to get more sensitive information, like Summoner Info and Email from the **userinfo** endpoint.

Throughout the tutorial (and in the final **server.js** file) we will be using a Client ID and Secret that don't exist. For this example to work, you will need to replace these with the ID and Secret you obtained when you registered a client.

To begin demonstrating this, we will need a quick webapp with two routes:

- **/** - the index page, a place where we can click a nice big 'Sign In' link
- **/oauth2-callback** - where we will ask Riot Sign On to redirect the player after a successful Sign In.

```
var express = require('express'),
    app = express();

app.get('/', function(req, res) {
    res.send("index");
});

app.get('/oauth2-callback', function(req, res) {
    res.send("callback");
});

var server = app.listen(3000, function () {
    var host = server.address().address;
    var port = server.address().port;

    console.log('Example app listening at http://%s:%s', host, port);
});
```

Sending Users to Riot Sign On

The purpose of route **/** is to deliver a Sign In link that the player can click to authenticate against Riot Sign On. Riot Sign On expects us to pass several parameters to it, or it will throw an error telling us what we've forgotten. In particular we need:

- **mandatory fields**
  - **redirect_uri** - OAuth 2 Callback route we have set up at our own server. This route needs to be able to process a **code** query parameter that is added to the URI on when Riot Sign On redirects the player back to our URI. We must also be sure we have this URI added as one of the **redirect_uri**s during client registration. **In this example, we will use http://local.exmaple.com:3000/oauth2-callback**
  - **client_id** - id assigned to client during registration. This will be the Client ID you got when you registered a client during the "Before Continuing" steps above.
  - **response_type** - response type expected, should be **code** for authorization code flow
  - **scope** - a predefined data scope, **must** include **openid** to authenticate.
  - Additional scopes that can be requested are
    - **cpid** which returns the game region for league of legends.
    - **offline_access** allows refresh tokens to be used to retrieve new access_tokens that have access to the /userinfo endpoint
- **optional fields**
  - **login_hint** - field used to specify hints to pre-populate data on the Login Page. Supports several formats:
    - **{regioncode}** - lowercase region code, e.g. **dev9**
    - **{regioncode}|{username}** - lowercase region code, pipe, and username, e.g. **dev9|daguava**
    - **{regioncode}#{userid}** - lowercase region code, hash, and user id, e.g. **dev9#2054**
  - **ui_locales** - space-separated list of player's preferred BCP47 language tag values in order of most to least preferred.
  - **state** - an opaque value provided to the authorize endpoint. The same value is returned to you when the endpoint sends its reply. Enables you to compare value sent and received, to prevent CSRF.

Adding all the mandatory fields together, our Sign In link is:

```
https://auth.riotgames.com/authorize?redirect_uri=http://local.leagueoflegends.com:3000/oauth2-callback&client_id=oujzg5jiibvzo&response_type=code&scope=openid
```

I've made code to produce the Sign In link a bit more portable, our **server.js** now looks like:

```
. . .
var appBaseUrl     = "http://local.example.com:3000",
  appCallbackUrl  = appBaseUrl + "/oauth2-callback";

var provider       = "https://auth.riotgames.com",
```

```
  authorizeUrl    = provider + "/authorize";

var clientID       = "oujzg5jiibvzo";

app.get('/', function(req, res) {
  var link = authorizeUrl
        + "?redirect_uri=" + appCallbackUrl
        + "&client_id=" + clientID
        + "&response_type=code"
        + "&scope=openid";
  // create a single link, send as an html document
  res.send('<a href="' + link + '">Sign In</a>');
});
 . . .
```

Here, the player is presented with the Login Page of Riot Sign On, and may sign in.

Response From Riot Sign On

When the player successfully logs in, a 302 Redirect sends their browser to the **redirect_uri** that we included in our Sign In link.

The only problem is our callback route **http://local.example.com:3000/oauth2-callback** doesn't do anything yet.

This route receives a **code** as a url query-string parameter, and the server must then make a server-to-server request to exchange this code for access, identity, and refresh tokens. We'll need to send a few things to Riot Sign On's **token endpoint** to get these tokens back:

- **Authorization: Basic ...** - We must set an Authorization Header. The format of this header is **"Basic " + Base64Encode(client_id + ":" + client_secret)**
- form data
    - **grant_type** - We must tell Riot Sign On we are working with the **authorization_code** grant type.
    - **code** - We pass Riot Sign On the access code, which we received as a querystring parameter to our oauth2-callback route: **...com:3000/oauth2-callback?code=CxhkPgX8GiMKR4-E-YD8Ng**
    - **redirect_uri** - We pass in the same redirect_uri we used before.

To simplify making requests, we will be using the **request** package. Here's what our basic callback route looks like:

```
 . . .

var request = require('request');
```

```
var clientID     = "your-client-id-here",
    clientSecret = "your-client-secret-here";

var appBaseUrl      = "http://local.example.com:3000",
    appCallbackUrl  = appBaseUrl + "/oauth2-callback";

var provider        = "https://auth.riotgames.com",
    authorizeUrl    = provider + "/authorize",
    tokenUrl        = provider + "/token";

. . .

app.get('/oauth2-callback', function(req, res) {
    var accessCode = req.query.code;

    // make server-to-server request to token endpoint
    // exchange authorization code for tokens
    request.post({
        url: tokenUrl,
        auth: { // sets "Authorization: Basic ..." header
            user: clientID,
            pass: clientSecret
        },
        form: { // post information as x-www-form-urlencoded
            grant_type: "authorization_code",
            code: accessCode, // accessCode should be url decoded before
being set here
            redirect_uri: appCallbackUrl
        }
    }, function (error, response, body) {
        // do something with the response?
    });

});
```

A raw token endpoint response looks similar to

```
{
  "scope":"openid",
  "expires_in":600,
  "token_type":"Bearer",
  "refresh_token":"dXJuOnJpb3Q6cOk1qdNal ...
8zN3NzbQ.xw96rZeGEMtrFlDCGLyA",
  "id_token":"eyJhbGciJSUzI1mtpZCInMxIn0 ...
```

```
YiI6InVybjpyaW90OpZDp2MTpNalV",
  "sub_sid":"vldfsXGdDPoafSKfjS932cslKu8JDUKZ-woZvXDoq8",
  "access_token":"eyJhbGciOi1NsImZCI6InM ...
NTkzMTA3LCJjaWQiJnmE-BVnZbYqY"
}
```

Lets change the callback of our response to parse this and grab the tokens.

```
request.post({
   . . .
}, function (error, response, body) {
   if (!error && response.statusCode == 200) {
      // parse the response to JSON
      var payload = JSON.parse(body);

      // separate the tokens from the entire response body
      var tokens = {
         refresh_token:  payload.refresh_token,
         id_token:       payload.id_token,
         access_token:   payload.access_token
      };

      // legibly print out our tokens
      res.send("<pre>" + JSON.stringify(tokens, false, 4) + "</pre>");
   } else {
      res.send("/token request failed");
   }
});
```

An explanation of all fields in the response follows:

- **scope** - Details what level of access the given Access Token provides. See the scopes list for more information.
- **expires_in** - life span (in seconds)
- **token_type** - Method of authorization token provides. *Bearer* means the entire token should be provided.
- Bearer eyJhbGciOi1NsImZCI6InM ... NTkzMTA3LCJjaWQiJnmE-BVnZbYqY

- **sub_sid** - The identifier of an existing session (SID) for the subject (player).
- **(refresh/id/access)_token** - detailed below

Using Tokens and Verification

The Access and ID Tokens are received as the final step of a grant. Access Tokens are used for scoped authentication of a client and player to a resource, while ID Tokens provide information necessary to authenticate a player's identity. ID Token are usually set as a cookie in the user's browser to establish identity between pages and services.

Riot Sign On Access and ID Tokens are encoded as Signed JSON Web Tokens (JWT) to prevent tampering. Additionally Access Tokens are encrypted and cannot be decoded.

Refresh Tokens are used to obtain a new Access Token when a given access token has expired, and thus have a much longer lifespan than an access token. The expiration flow of access tokens ensures that even if one is compromised, it has a very limited life-span.

IMPORTANT
It is IMPORTANT to note that Access Tokens are encrypted and cannot be decoded or verified

Example
Once done, a full example provides the following:

- A **`Sign In`** link to the player
- The link takes the user to Riot Sign On to log in
- The player is redirected back to our **`redirect_uri`** with an access **`code`** appended to the url
- The Oauth2-Callback route exchanges the access **`code`** for tokens
- A server-to-server request is made to exchange the code for Access, Identity, and Refresh Tokens
- Riot Games **verifies the authenticity** of these tokens

Send a player to Riot to authenticate, and receive the auth code returned and exchange it for Access, Identity, and Refresh tokens;

Using Refresh Tokens

The Refresh Token is issued for the purpose of obtaining new Access Tokens when an older one expires. RSO Refresh tokens are self-contained, signed JSON Web Tokens (JWT) that can be inspected and validated locally.

Category	Description
Format	JWT signed

Refreshable?	N/A
Usage	Obtain a new Access Token
Visibility To Javascript	No
Visibility To User	No
Visibility To Server	Yes

Example encrypted token:

dXJuOnJpb3Q6cGlkOnYxOk1qVXdNalE2UkVWV09R.Z2pyamNvaG8zN3NzbQ.xw96rZeGEmeMtrFIDCGLyA

Using a Refresh Token

Refresh Tokens are only used to obtain a new Access Token, usually when the previous one has expired (due to time expiration, or revocation.)

Headers

Authorization: Basic Z2pyamNvaG8NzbTpPLWpTb ... tkcEN6amp13U2ZTOWpjU0w=
POST-Data

grant_type: refresh_token
refresh_token: dXJuOnJpb3Q6cGlknYxO ... G8zN3NzbQ.xw96rZEmeMtrFIDCGLyA
(optional) scope: [same or narrower scope]

URI
https://auth.riotgames.com/token

Server Response:

{
"scope":"openid",
"expires_in":600,

"token_type":"Bearer",
"refresh_token":"dXJuOnJpb3Q6cGlkOn ... amNvaG8zN3NzbQeGEmeMtrFlDCGLyA",
"access_token":"eyJhbGciOiJSUzI1NiI ... sFwkadLmWmwtvJouhX22Tc6vPnfXTk"
}

When using a refresh token, there are two actions RSO may take. If it replies with a new refresh token, it must be used in all future access token refreshes and the previous refresh token is now invalid.
If the same refresh token is received in the response, you may continue to use it in future refresh requests.

Accessing the Userinfo Endpoint

The UserInfo endpoint is a protected RSO endpoint where client applications can retrieve consented claims, or assertions, about the logged in player.

The claims are typically packaged in a JSON object where the sub member denotes the subject (player) identifier.

This is the first endpoint established to support Bearer access tokens from Implementing Authorization Codes on the Web.

Creating a Request

Headers
Authorization: Bearer eyJhbGciOiJSUzI ... axZMUW2WchZU9fGHM_h61A
URI
https://auth.riotgames.com/userinfo

Response

{
"sub":
"2SqiN9ZWecDMZdo-y3Xaaoos32kTazZQDgzOyxzJe66SzGYqIljuuxjmctK0-XbBIhzZn929LZnH5S90L4vNVjx7En27",
"cpid": "NA1"
}