

# OAuth 2 and JWT with Cookies

The challenge we are solving with this article is that OAuth 2.0 requires "smart" clients to perform the login, the refresh process, as well as to put the access token in an `Authorization: Bearer` header on all outgoing requests. This is the standard approach for many native, mobile or JavaScript client applications. For those who need to also support plain HTML browser apps, which do not use JavaScript, migration to OAuth 2.0 and JSON Web Tokens (JWT) are out of reach making it impossible to support old and new clients with the same security architecture.

This tutorial shows how to use the experimental feature of the Tribestream API Gateway (TAG) to send the JWT back as an HTTP Cookie in order to support legacy/plain HTML browser apps.

## Some regular OAuth 2.0 for context

On a regular OAuth 2.0 authentication flow using the Tribestream API Gateway (TAG) the user posts the username and password and receives a JSON object containing two different JWTs. One representing the access token (AT), and another representing a refresh token (RT). The "smart" client side usually stores them on the available local storage.

The AT is then sent in all requests sent from the client side to the backend side to authenticate the calls. This token has a validity period and can be refreshed if an RT is available. When the AT expires the client application must detect the expiration and automatically re-authenticate by posting the RT to the authentication endpoint. If valid, the server then replies with a new AT+RT pair. This can be repeated many times until the RT itself expires. At this point, the user must re-authenticate with username and password.

What we achieve with this flow is a frequent refresh of the user credentials. Among other advantages, in case of AT interception, a rogue system has a very limited amount of time to impersonate the user.

JWTs frequently include relevant, signed information that can be used both by the client and the server side to effectively achieve stateless authentication.

Handling the AT, RT requires some cleverness from the client side, usually performed with JavaScript.

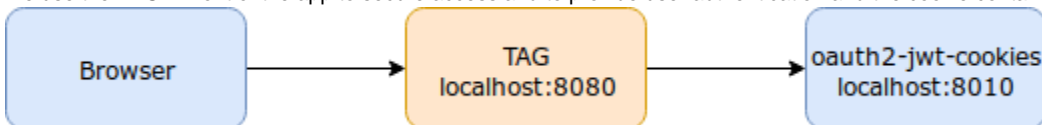
For legacy applications, this might be a problem. Hence we also provide the AT in the form of a cookie. This doesn't require client application intervention, but, if needed a JavaScript JWT decoding library can be used to access the JWT data.

## Use the OAuth 2.0 cookie-based authentication.

To demonstrate this type of authentication we created a simple form based application. The source code can be obtained here: <https://github.com/tomitribe/oauth2-jwt-cookies>.

The app has just 2 pages, a **login form** and a **main page** where we demo JWT decoding with JavaScript.

We use the TAG in front of the app to secure access and to provide user authentication and the cookie containing the AT JWT.



## The app deployment

The `oauth2-jwt-cookies` project creates a WAR file that can be deployed on a standard `TomEE 7.0.4` application server. For simplicity, we assume that TomEE will run on `localhost:8010`. You can use the `server.xml` sample file that we provide in the project for that configuration.

In alternative, you can just checkout the code, build it with Maven and run Tomee, in one go, with the following commands:

```
git clone https://github.com/tomitribe/oauth2-jwt-cookies.git
cd oauth2-jwt-cookies
mvn clean install tomee:run
```

## TAG deployment

We have an experimental TAG Docker image that you can use. On Linux, if you have Docker already installed, just execute:

```
docker run -it --net="host" --name tag  
tomitribedev/tribestream-api-gateway
```

If you agree with the TAG license, you must add the following attribute to the previous command:

```
-e LICENSE=accept
```

If you already have a contained named tag, you need to remove it by executing:

```
docker rm tag
```

On the TAG side we need to set up the configurations described in the next sections. You can do this by logging into the TAG at <http://localhost:8080/tag/> using username and password as *admin*.



## OAUTH2 PROFILE.

We need to perform two changes in the OAuth2 Profile page.

Go to Security Profiles (<http://localhost:8080/tag/profiles>) and select the "OAUTH2 AUTH PROFILE". The client id is not required:

## CLIENT ID REQUIRED

no ☐ yes

To better demonstrate the AT expiration, reduce it to just 1 min:

## ACCESS TOKEN POLICY

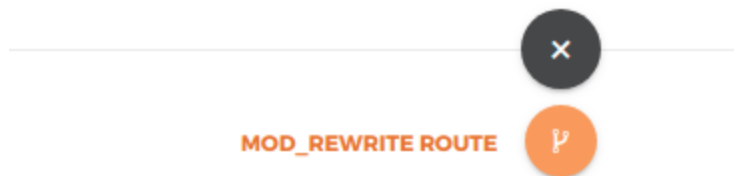
Expire 1 minute after issue

Save your change by clicking on the top right corner save icon.

## Routes

From the Oauth2 Profile page navigate to <http://localhost:8080/tag/routes>

We need to create 2 new routes to control the access to our app. For that, let's add the routes by clicking MOD\_REWRITE ROUTE on the menu to the right of the screen:



## The Main route

This 1st route will protect the main page, the one we are going to see after a successful authentication. This authentication is enforced by adding the *auth* flag to the rewrite rule and selecting the "OAUTH2 AUTH PROFILE" we configured before.

## CREATE MOD\_REWRITE ROUTE

NAME	form
------	------

form

## MOD\_REWRITE

```
RewriteRule ^/?oauth2-jwt-cookies/main/(.*)$ "http://localhost:8010/oauth2-jwt-cookies/main/$1"
[QSA,P,NE,auth]
```

## SECURITY PROFILES

 OAuth2 Auth Profile

## ROLES

Select role...

## DESCRIPTION

B I H         

Add here your HTTP description

## TAGS

Select tag...

 [Create another](#)

Cancel

SAVE

Replace all the pre filled `mod_rewrite` examples with the following `mod_rewrite` rule:

```
RewriteRule "^/?oauth2-jwt-cookies/main/(.*)$"
"http://localhost:8010/oauth2-jwt-cookies/main/$1" [QSA,P,NE,auth]
```

On the bottom, I checked the “Create another” checkbox and hit SAVE.

## The Login route

This route allows access to the login page and other assets like all the images and css files. On this one, we must not configure authentication:

CREATE MOD\_REWRITE ROUTE

×

NAME

form-login

MOD\_REWRITE

RewriteRule `"/oauth2-jwt-cookies/(.*)$" "http://localhost:8010/oauth2-jwt-cookies/$1" [QSA,P,NE]`

DESCRIPTION

B

I

H

</>

“”

≡

≡

✍

□

👁

✕

?

Add here your HTTP description

TAGS

Select tag...

☐ Create another
Cancel





SAVE

You just need to set this rewrite rule on the mod\_rewrite body:

```
RewriteRule "/oauth2-jwt-cookies/(.*)$"
"http://localhost:8010/oauth2-jwt-cookies/$1" [QSA,P,NE]
```

Hit SAVE.

On the routes list page, we must ensure that the *Main* route is above the *Login* route, ensuring a higher priority. This is because the *Login* route has a broader scope, `"/oauth2-jwt-cookies/(.*)$" would grab all traffic, including the one for "/oauth2-jwt-cookies/main/(.*)$":`

 FORM	<code>"/oauth2-jwt-cookies/main/(.*)\$"</code>	 1
 FORM-LOGIN	<code>"/oauth2-jwt-cookies/(.*)\$"</code>	 0

## Hands on

Going back to the application that we already lunched... In order to perform the login we need to make the form send the authentication request:

```

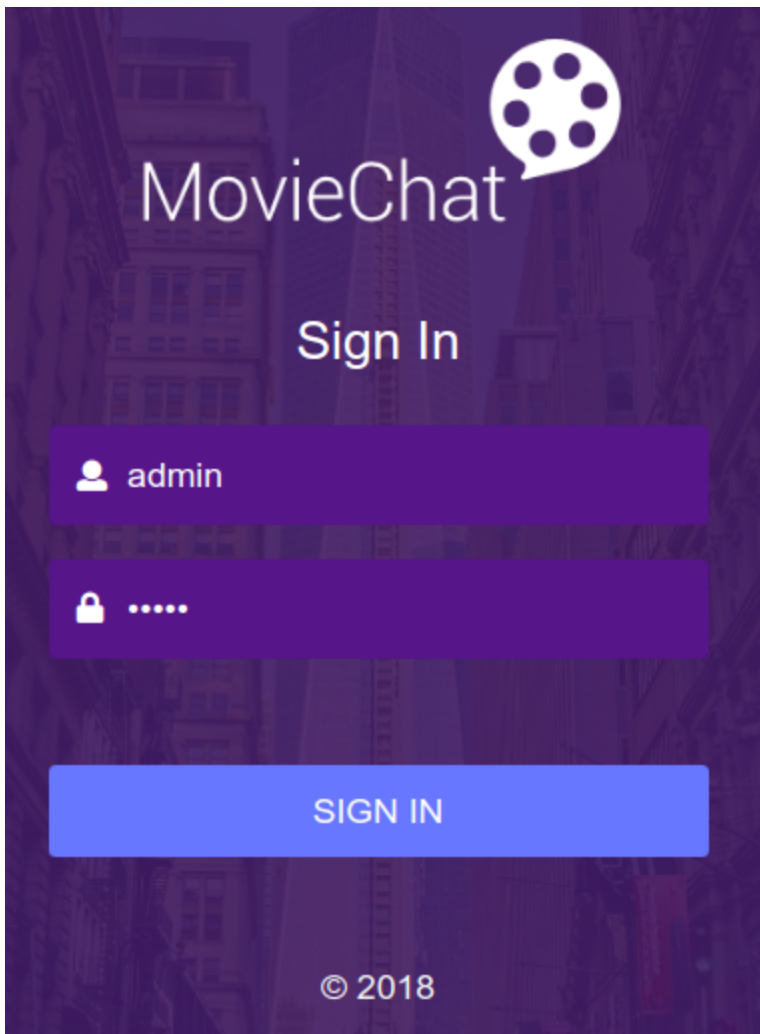
<form class="form-login" action="/oauth2/cookie" method="post">
  <input type="hidden" name="ref" value="/oauth2-jwt-cookies/main/">
  
  <h1 class="h4 mb-4 font-weight-light pt-4">Sign In</h1>
  <div class="inner-addon left-addon">
    <label for="inputUsername" class="sr-only">Username</label>
    <i class="fa fa-user"></i>
    <input type="text" name="username" id="inputUsername"
class="form-control" placeholder="Username" required autofocus>
  </div>
  <div class="inner-addon left-addon">
    <label for="inputPassword" class="sr-only">Password</label>
    <i class="fa fa-lock"></i>
    <input type="password" name="password" id="inputPassword"
class="form-control" placeholder="Password" required>
  </div>
  <input type="password" type="text" name="grant_type"
value="password" hidden required>
  <button class="btn btn-lg btn-primary btn-block text-uppercase mt-5"
type="submit">Sign in</button>
  <p class="mt-5 mb-5 text-muted">&copy; 2018</p>
</form>

```

You can check the [code here](#).

To test the application open the following url on a browser:

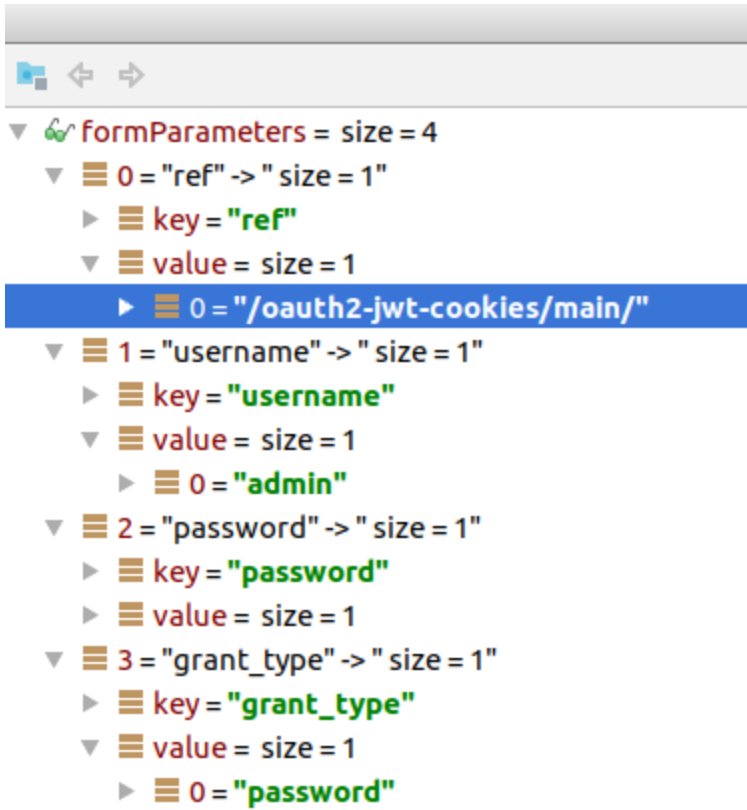
<http://localhost:8080/oauth2-jwt-cookies/login>. Please notice the 8080 port. You will be accessing the APP through the TAG.



Use admin, admin as username and password. Once you hit "sign in", the form will post the user credentials to the <http://localhost:8080/oauth2/cookie> endpoint of the TAG, then you will be redirected to the main page. This redirect happens because the form also submits the end URL for the redirect:

```
<input type="hidden" name="ref" value="/oauth2-jwt-cookies/main/">
```

These are the field posted to the TAG:




This is the main page you get:




← → ↻

localhost:8080/oauth2-jwt-cookies/main/

 **MovieChat**

[Login page](#) [Clear cookie](#)

 **WELCOME**  
admin

Access Token

```
{
  "header": {
    "zip": "GZIP",
    "cty": "json",
    "typ": "JWT",
    "alg": "HS256",
    "kid": "oauth2-secretkey"
  },
  "payload": {
    "token-type": "access-token",
    "nbf": 1527849355,
    "roles": [
      "admin"
    ],
    "name": "admin",
    "iss": "/oauth2/token",
    "groups": [],
    "tag-internal": {
      "grant-type": "password",
      "refresh-exp": 1530441355730,
      "profile": "OAuth2 Auth Profile",
      "refresh_token_id": "9a8ef4c2da2ffc35",
      "version": "1.0",
      "refresh-nbf": 1527849355730,
      "username": "admin"
    },
    "exp": 1527849415,
    "iat": 1527849355,
    "jti": "49138bf7c75263ef"
  },
  "signature": "Htq_o2hLoNCqv1JnMzZpIBpuoW5BLGxlq1KVHTc-uv8"
}
```

Refresh Token

undefined

Cookie

```
{
  "access_token": "eyJ6aXAiOiJHwklQIiwia3R5IjoianNvb2IiInR5cCI6IkpXVCIsImFsZyI6IkhTMjU2Iiwia2lkIjoib2F1dGgyLXNlY3JldGtleSJ9.H4sIAAAAAAAAAAF2Q3w7CMAYFXwX5uh1tk64_d3uC7X5MyLQ0hJwStL9CPHucyKY2G4i2T5fzrHPE0w7mTx8zwQ94DCQ93nqQQZmp6Av66ppZSfq0gNnJ_LQvwK0J23gjSV4SmCqM9Cex7BZW1zCodqsbz_tnV3mSDIScJ9rE8gZnKA_8wxNuCWy0ftP60ZmHClH_pDT1xxTiELKk1M0oshgd1bpKQLPT9FpFd_Vy7X7y26T_1aPL0ywJSWHasRKqUHUrpog57U1PCwfijvHf3snx8XHWfLXjK4Jks6WfJ9NIa_BzsGzYDsStHuVDM0dfUoSMH1By20cs54AQAA.Htq_o2hLoNCqv1JnMzZpIBpuoW5BLGxlq1KVHTc-uv8",
  "scope": "admin",
  "token_type": "bearer",
  "expires_in": 59
}
```

In the main page, we use some JavaScript to demonstrate how to access the contents of the cookie.

The Cookie value is [base64URL](#) encoded and we need to decode it. That will give you the right column.

Next, we uncompress the payload part of the AT and decode its contents into the left column.

To reduce the cookie size we are not sending the RT, to make possible the transparent refresh of the AT, we added the following claims:

- **refresh-exp**, the expiration date of the refresh token.
- **refresh\_token\_id**, the ID of the refresh token, cached in the TAG.
- **refresh-nbf**, We will not allow this refresh token to be used before this date

These claims will make the refresh possible and more efficient.

## Security considerations

Before going too deep, please consider that the cookie approach has security disadvantages. This is why we only recommend it for compatibility reasons, on legacy applications.

Cookies work well with singular domains and sub-domains, but cause problems when it comes to managing cookies across different domains.

Cookies are more vulnerable to these kinds of attacks than the header approach:

- Man in the middle
- Cross-site scripting (XSS)
- Cross-site request forgery (XSRF)

We must also state that **the JWT in a cookie approach is also a security improvement in relation to an old session ID identifying the user**. This is because that the AT inside the cookie can be frequently rotated, hence reducing the amount of time where it can be used.

The AT inside the cookie cannot be forged because it was cryptographically signed by the server.