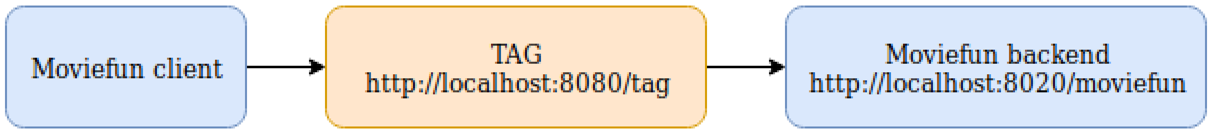
OAuth 2.0 authentication with vue.js

Here we explain how a client application implemented with the vue.js framework can perform OAuth 2.0 authentication against the Tribestream API Gateway (TAG) and also access the contents of the provided JWT.

To do this we created a setup with the TAG proxying the requests and handling authentication for the Moviefun app, running on an independent server.



The app deployment

The OAuth2-jwt-vuejs 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:8020.

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-vuejs

cd oauth2-jwt-vuejs

mvn clean package 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 two 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:

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

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:

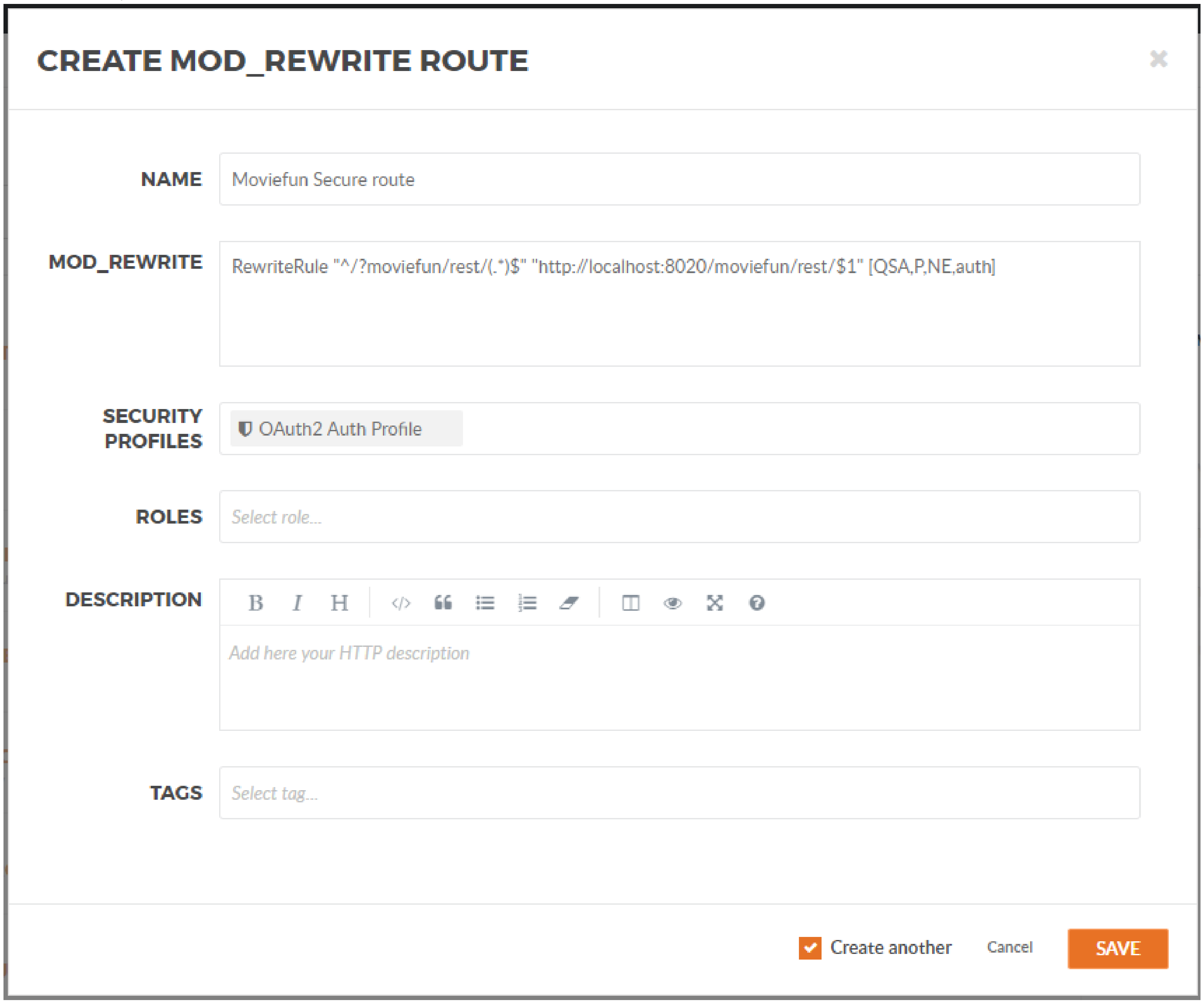
The Secure route

This 1st route will protect the Rest API endpoint, the one we are going to access 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.

Replace all the pre filled mod_rewrite examples with the following mod rewrite rule:

```
RewriteRule "^/?moviefun/rest/(.*)$" "http://localhost:8020/moviefun/rest/$1" [QSA,P,NE,auth]
```

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



CREATE MOD_REWRITE ROUTE

NAME	Moviefun Secure route
MOD_REWRITE	RewriteRule "^/?moviefun/rest/(.*)$" "http://localhost:8020/moviefun/rest/$1" [QSA,P,NE,auth]
SECURITY PROFILES	OAuth2 Auth Profile
ROLES	Select role...
DESCRIPTION	Add here your HTTP description
TAGS	Select tag...

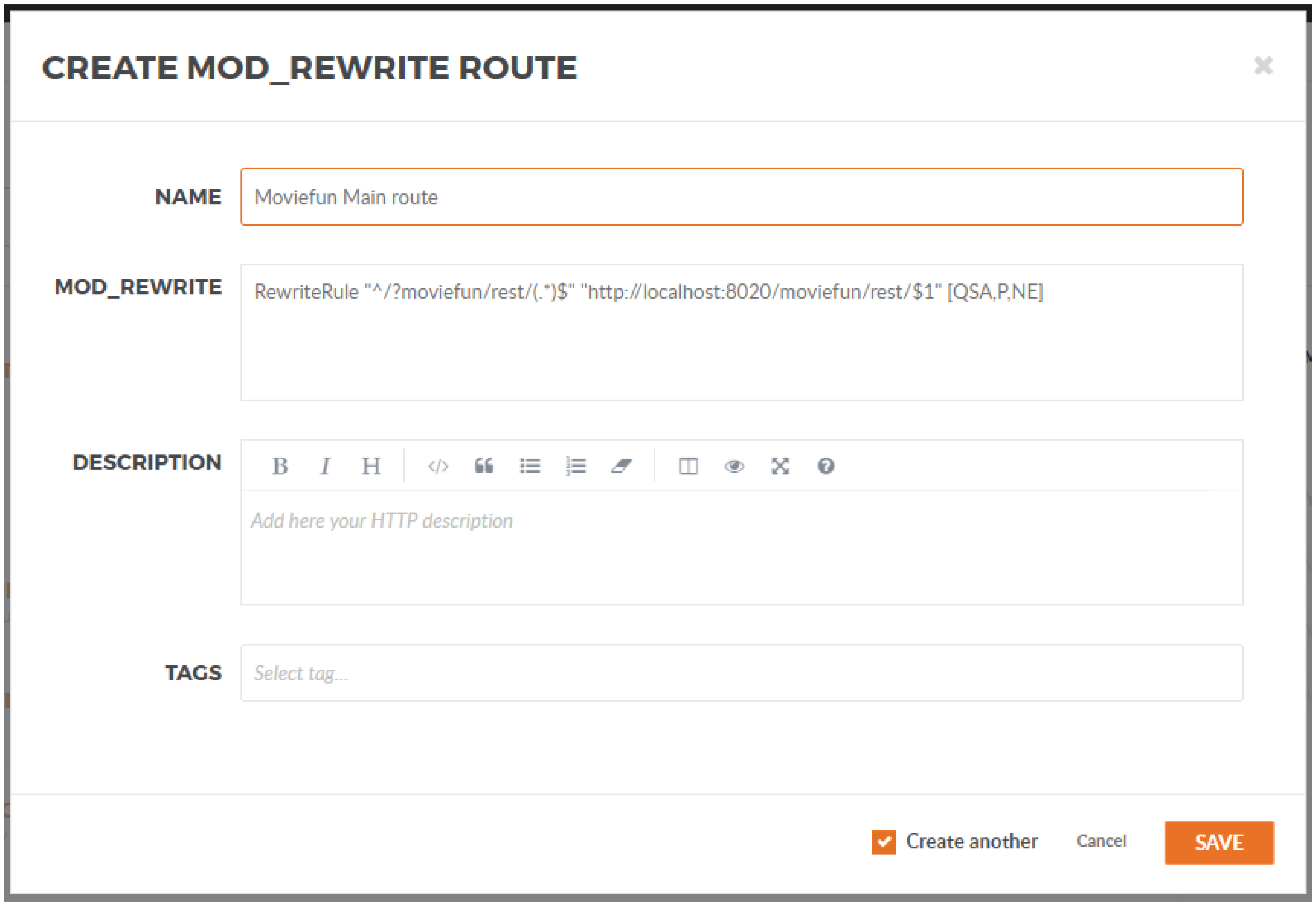
The Main route

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

You just need to set this rewrite rule on the mod_rewrite body:

```
RewriteRule "^/?moviefun/(.*)$" "http://localhost:8020/moviefun/$1" [QSA,P,NE]
```

Hit SAVE…


CREATE MOD_REWRITE ROUTE

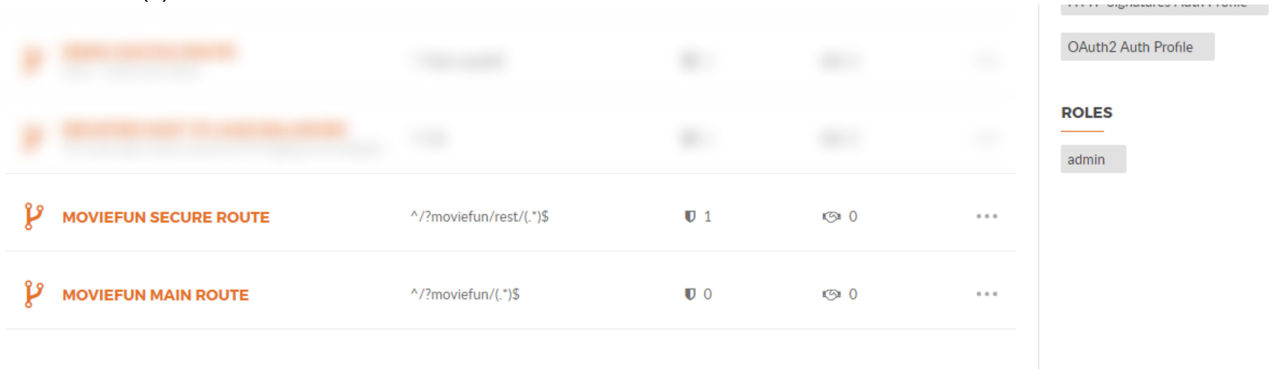
NAME: Moviefun Main route

MOD_REWRITE: RewriteRule "^/?moviefun/rest/(.*)$" "http://localhost:8020/moviefun/rest/$1" [QSA,P,NE]

DESCRIPTION: Add here your HTTP description

TAGS: Select tag...

After successful creation of both routes go to the routes list page once again, you must ensure that the *Secure* route is above the *Main* route, ensuring a higher priority. This is because the *Main* route has a broader scope, "^/?moviefun/(.*)$" would grab all traffic, including the one for "^/?moviefun/rest/(.*)$":

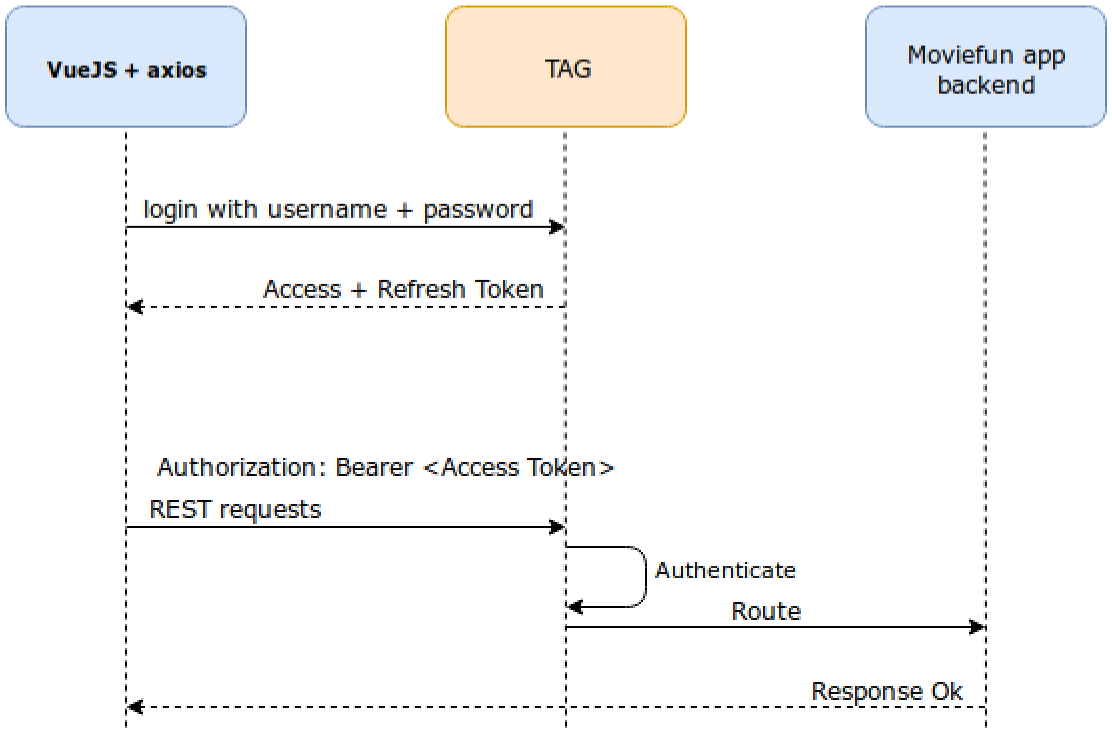


MOVIEFUN SECURE ROUTE	^/?moviefun/rest/(.*)$	1	0
MOVIEFUN MAIN ROUTE	^/?moviefun/(.*)$	0	0

Vue.js

For Vue.js we are using Axios as the HTTP client.

Login



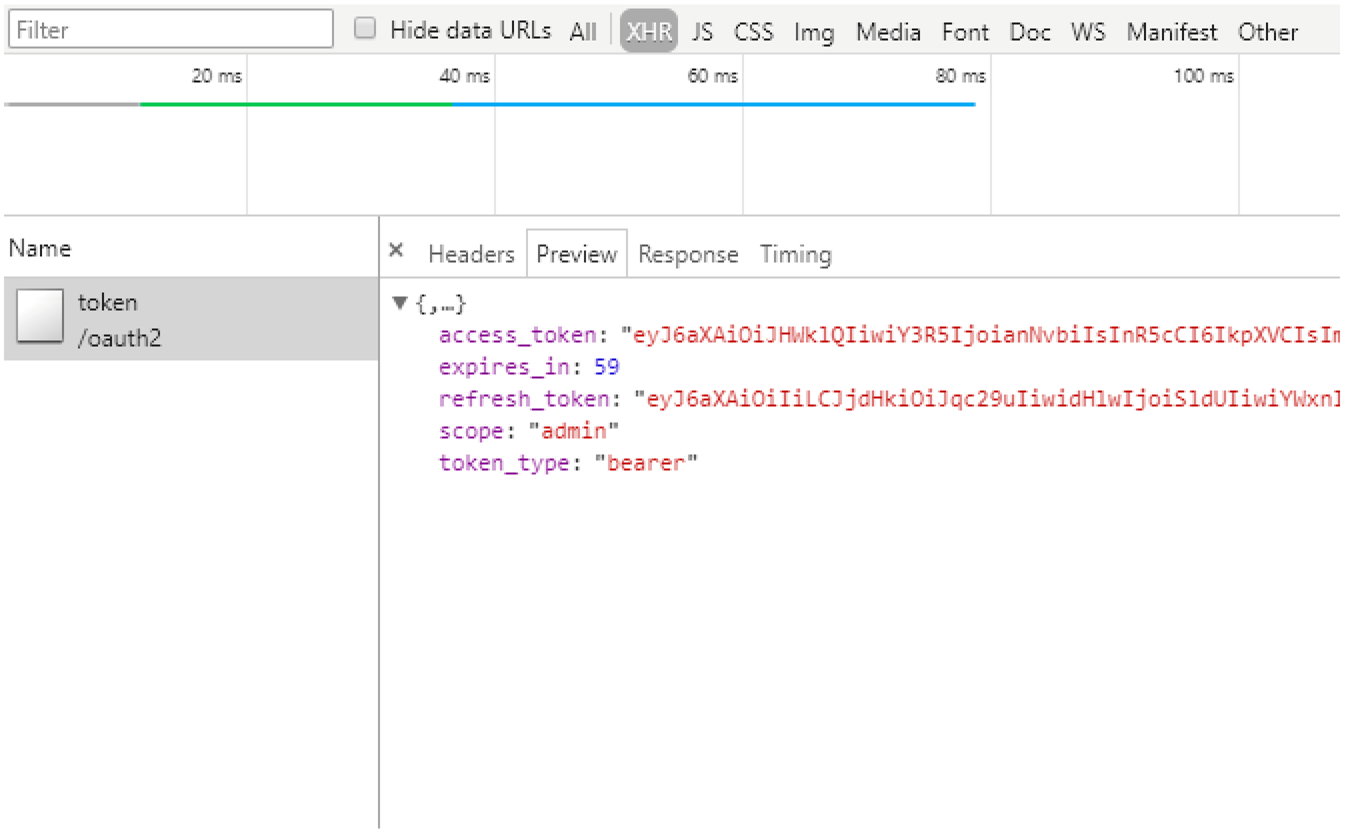
To post a **Login Request** we need to know TAG Oauth2 profile endpoint and send our credentials object there: http://localhost:8080/oauth2/token

```
{
     username: "<username_string>",
     password: "<user_password_string>",
     grant_type: "password"
 }
```

In `x-www-form-urlencoded` form (You can check js/store/auth.js file from the project)

```
axios({
    method: 'post',
    url: location.origin + '/oauth2/token'),
    headers: {
        'Content-type': 'application/x-www-form-urlencoded'
    },
    data: $.param({
        username: credentials.username,
        password: credentials.password,
        grant_type: credentials.grant_type
    })
})
```

on this request you will receive a response object:



```
▼ {,…}
    access_token: "eyJ6aXAiOiJHWk1QIiwiY3R5IjoianNvbiIsInR5cCI6IkpXVCIsIm
    expires_in: 59
    refresh_token: "eyJ6aXAiOiIiLCJjdHkiOiJqc29uIiwidHlwIjoiS1dUIiwiYWxnI
    scope: "admin"
    token_type: "bearer"
```

```
{
    "access_token": "<access_token_string>",
    "refresh_token": "<refresh_token_string>",
    "scope": "admin",
    "token_type": "bearer",
    "expires_in": 59
}
```

`token_type` property specifies what prefix tag is expecting for `Authorization` header (case insensitive) e.g:

```
"headers": {
    "Authorization": "bearer <access_token_string>"
}
```

`expires_in` property specifies number of seconds our Access Token will expire in.

Token Storage

For this specific case we are using `localStorage` storage for both Access and Refresh tokens, and vuex as a state management pattern and a store. You can check js/store/auth.js file from the project.

```
mutations: {
    /*...*/
    STORE_ACCESS_TOKEN(state, accessToken) {
        state.accessToken = accessToken;
        setAuthorizationHeader(state.accessToken);
        localStorage.setItem('accessToken', accessToken)
    },
    STORE_REFRESH_TOKEN(state, refreshToken) {
        state.refreshToken = refreshToken;
        localStorage.setItem('refreshToken', refreshToken)
    },
    LOGOUT_USER(state) {
        state.accessToken = null;
        state.refreshToken = null;
        setAuthorizationHeader(state.accessToken);
        localStorage.removeItem('accessToken');
        localStorage.removeItem('refreshToken')
    }
},
```

```
getters: {
    /*...*/
    accessToken(state) {
        return state.accessToken
    },
    refreshToken(state) {
        return state.refreshToken
    }
},
```

to get or re-initialize our store from `localStorage` on reload we are using `router.beforeEach`.

You can check js/route.js file from the project.

```
router.beforeEach((to, from, next) => {
    let accessToken = localStorage.getItem('accessToken') ?
localStorage.getItem('accessToken') : null
    let refreshToken = localStorage.getItem('refreshToken') ?
localStorage.getItem('refreshToken') : null

    if (accessToken) {
        router.app.$options.store.dispatch('auth/setUserAndTokens', {
            accessToken: accessToken,
            refreshToken: refreshToken
        })
    }

    /*...*/
    next()
});
```

Token usage and integrations

After you received access_token and refresh_token you might want to unwrap and decode them. Remember that the JWT payload is GZIP compressed so it might need additional GZIP unzip on user part. We are creating a separate lib for a purpose of improving and simplifying it. But for the simplest solution is here in separated structure.

Now to access any secured endpoint you just need to add additional `Authorization` header with content: `"Bearer <access_token_string>"` . Here are a few alternatives to do that:

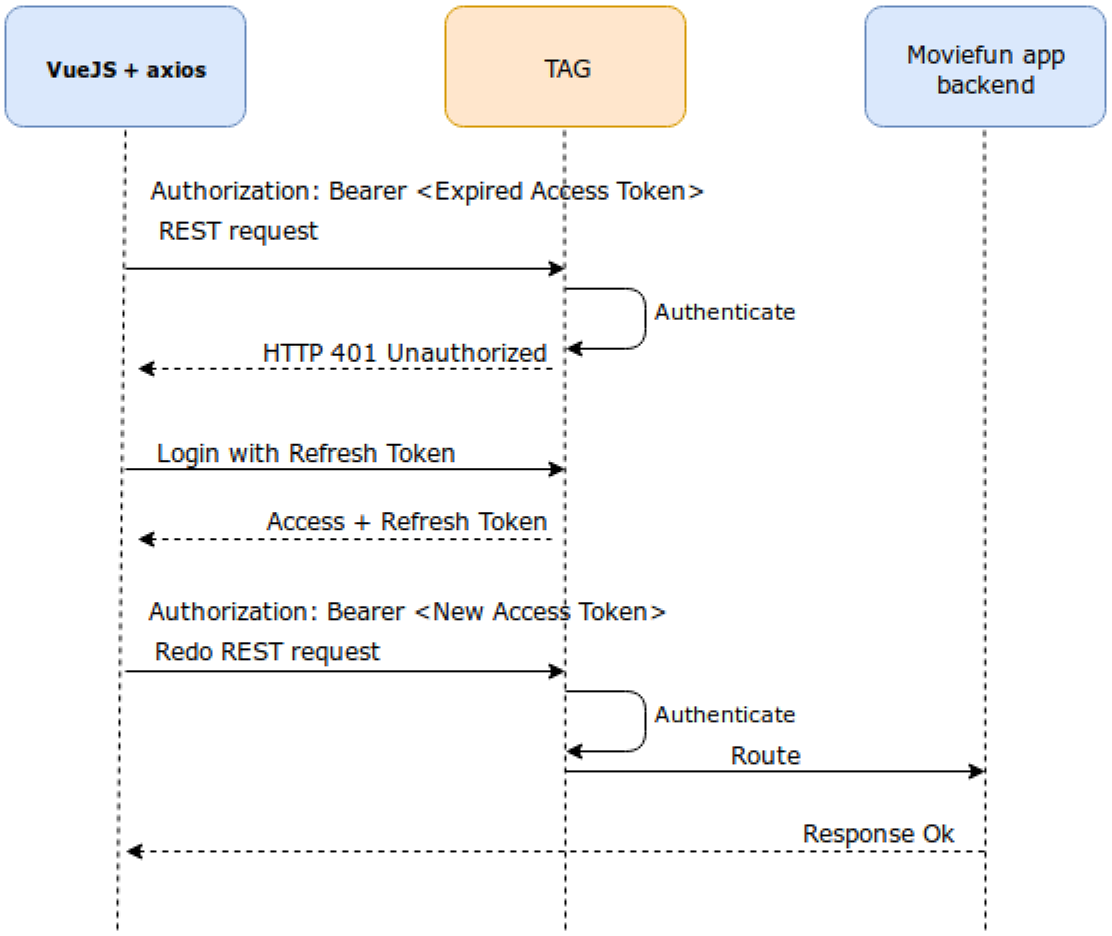
1) to add it globally to defaults. You can check js/common/header.js file from the project.

```
function setAuthorizationHeader(token) {
    if (token) {
        axios.defaults.headers.common['Authorization'] = `Bearer
${token}`;
    } else {
        delete axios.defaults.headers.common['Authorization'];
    }
}
```

2) or add an interceptor which will do that for you.

```
axios.interceptors.request.use((config) => {
  if (token) {
    config.headers.common['Authorization'] = `Bearer ${token}`;
  } else {
    delete config.headers.common['Authorization'];
  }
  return config;
});
```

Token refresh sequence



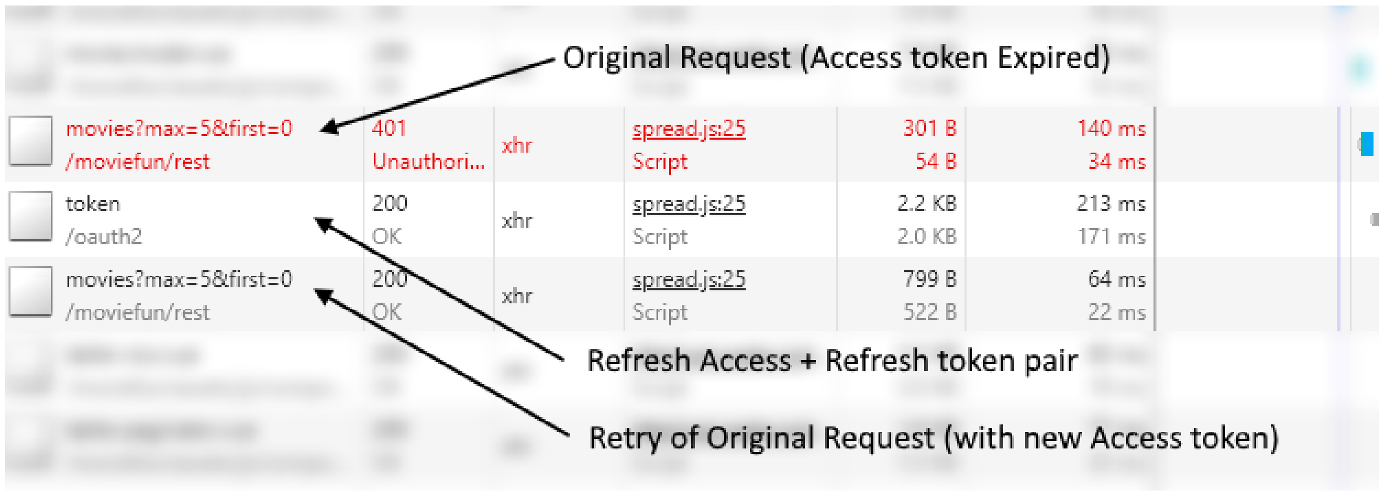
Last sequence, but not least, to use a Refresh Token to get a new Access + Refresh Token pair after your Access Token has expired.

For this sequence you need two parts:

Unauthorised Response interceptor

To catch a *Failed Request* with status 401 Unauthorized, we use `axios.interceptors.response`.

We'll have to get a new Access + Refresh Token pair. For that, we will use our current Refresh Token to login at the TAG's token endpoint: http://localhost:8080/oauth2/token (read **Refresh Token request**)


Original Request (Access token Expired)
Refresh Access + Refresh token pair
Retry of Original Request (with new Access token)

movies?max=5&first=0 /moviefun/rest	401 Unauthori...	xhr	spread.js:25 Script	301 B 54 B	140 ms 34 ms
token /oauth2	200 OK	xhr	spread.js:25 Script	2.2 KB 2.0 KB	213 ms 171 ms
movies?max=5&first=0 /moviefun/rest	200 OK	xhr	spread.js:25 Script	799 B 522 B	64 ms 22 ms

And after a successful **Refresh** we resend the *Original Failed Request* with new Authorization header `axios(originalRequest)`.

Example (You can check js/common/header.js file from the project):

```
axios.interceptors.response.use((response) => {
    return response
}, async (error) => {
    let originalRequest = error.config;
    if (error.response.status === 401 && !originalRequest._retry) {
        originalRequest._retry = true;
        try {
            const rt = store.getters['auth/refreshToken'];
            if (!rt) throw new Error('No valid refreshToken');
            const response = await
store.dispatch('auth/refreshUserTokens');
            await store.dispatch('auth/setUserAndTokens', {
                accessToken: response.data.access_token,
                refreshToken: response.data.refresh_token
            });
            originalRequest.headers['Authorization'] = 'Bearer ' +
response.data.access_token;
            originalRequest.baseURL = '';
            setAuthorizationHeader(response.data.access_token);
            return axios(originalRequest)
        } catch (error) {
            // All Vuex modules must logout here
            await store.dispatch('auth/userLogout');
            router.replace({
                name: 'login'
            });
            Vue.toasted.error('To verify your session, please login.');
            return Promise.reject(error)
        }
    }
    return Promise.reject(error)
});
```

Refresh Token request

It works the same way as **Login Request** but it has different post data parameters:

```
{
    refresh_token: "<refresh_token_string>",
    grant_type: "refresh_token"
}
```

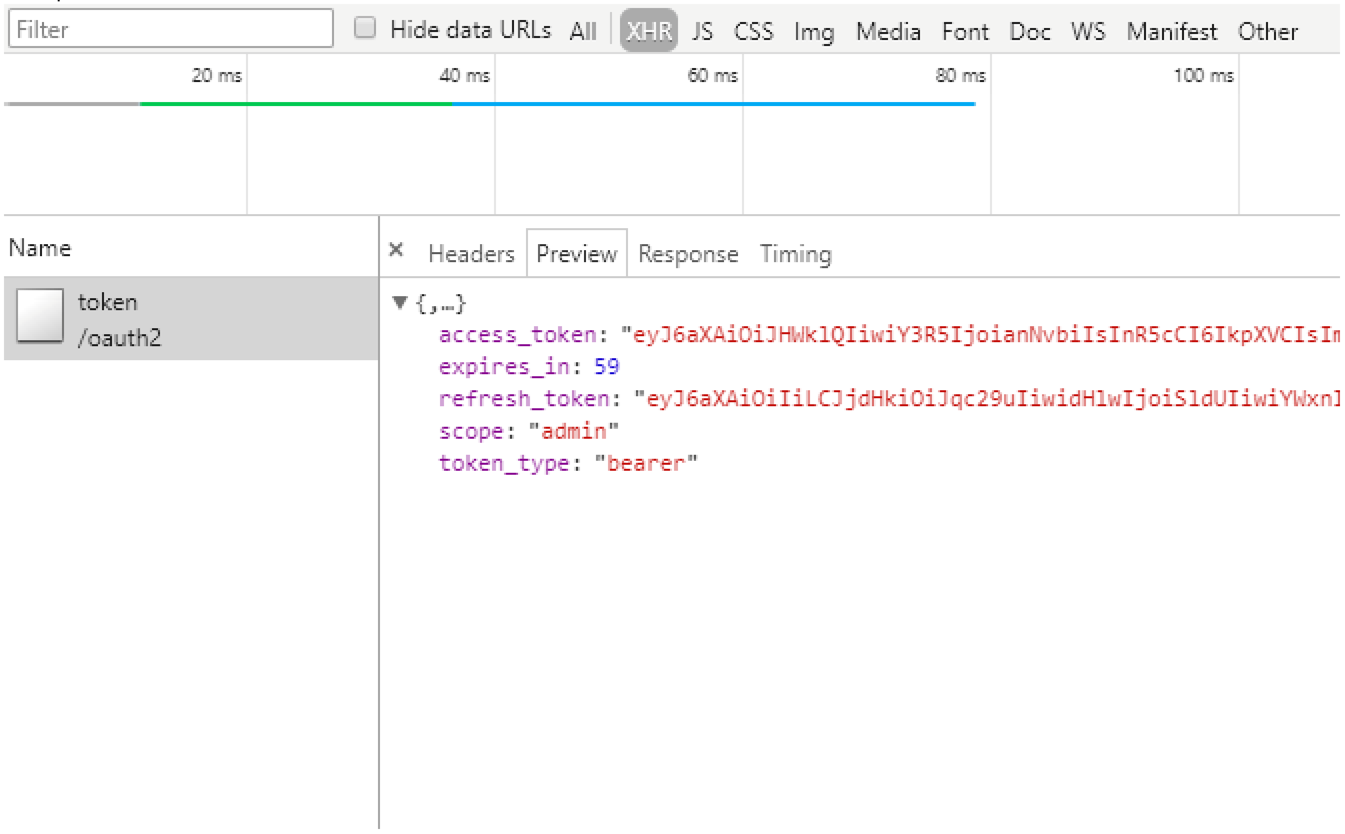
Example (You can check js/store/auth.js file from the project.):

```
axios({
    method: 'POST',
    url: location.origin + '/oauth2/token',
    headers: {
        'Content-type': 'application/x-www-form-urlencoded'
    },
    data: $.param({
        refresh_token: getters.refreshToken,
        grant_type: 'refresh_token'
    })
})
```

on this request you will receive new response object (properties list is same as in login request)

```
{
    "access_token": "<access_token_string>",
    "refresh_token": "<refresh_token_string>",
    "scope": "admin",
    "token_type": "bearer",
    "expires_in": 59
}
```

Example:



token
/oauth2

```
▼ {,…}
    access_token: "eyJ6aXAiOiJHWk1QIiwiY3R5IjoianNvbiIsInR5cCI6IkpXVCIsIm
    expires_in: 59
    refresh_token: "eyJ6aXAiOiIiLCJjdHkiOiJqc29uIiwidHlwIjoiS1dUIiwiYWxn
    scope: "admin"
    token_type: "bearer"
```