# Introduction

The blueprint for implementing the CSRF protection can be found in the OWASP site ( <https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet>)

The implementation involves

1. a token will be generated with a random value and persisted in the browser via cookie (a non http enabled cookie)
2. forward the cookie in at least all POST/PUT/DELETE requests
3. add a header to POST/PUT/DELETE requests that contains the value of the token (the same value set in the cookie)
4. verify, in the server, and prior to the request´s processing, that the token included in the cookie and the one in the header, are identical

# Cookie Generation

## Assumption

The Angular applications and the API that these applications consume are hosted in the same domain (non CORS). The cookie issued to the browser will be accessible both by the Angular SPA and by the XHR API requests.

## Generation Alternatives

The de-facto standard is to generate a cookie with the name XSRF-TOKEN, defined as non http (so that it can be included in XHR requests).

A particularly of the OCE architecture is that the container serving the SPA and the container hosting the APIs are different. However, the two containers are accessible through a reverse proxy (APIGee & Nevis), so the DNS host domain used for downloading the Angular application and for making API calls, is the same. This is a key factor for making the cookie accessible by both SPA and API regardless of where the cookie is generated (by downloading the SPA, or by making an API call).

There are several alternatives to generate the cookie:

|  | **Pros** | **Cons** |
| --- | --- | --- |
| APIGee | * Will be valid regardless of the technology used to implement the MiSe (node or spring) * Will target the APIs that are exposed to the SPA * APIGee could also be used as enforcement point |  |
| Nevis | * Will be valid regardless of the technology used to implement the MiSe (node or spring) * Will target the APIs that are exposed to the SPA | * Impact a 3rd party |
| Microservice | * Spring provides libraries to support CSRF out of the box | * Will need to be implemented in two different technologies (Node & Spring) * Internal API calls (not exposed to the SPA) will be affected by the CSRF validation; We would require additional logic to exempt these cases |

### Spring

The implementation in Spring of the CSRF leverages Spring Security:

1. Modify the oce-mise-lib to include Spring Security dependencies
2. Create a library to include the Spring security configuration. In the following example we apply CSRF validation to all end-points except those under /public/\*\* resource:

@Override

**public** **void** configure(HttpSecurity http) **throws** Exception {

http

.anonymous()

.and()

.csrf()

.csrfTokenRepository(CookieCsrfTokenRepository.*withHttpOnlyFalse*())

.ignoringAntMatchers("/public/\*\*")

# Forward the Cookie

Once the cookie is generated, we need to forward it in all the API calls made from the SPA, through Nevis/APIGee to the iAPC.

## Angular

When the url requests are relative, the forwarding of the cookies is done automatically (in Angular). When the url request made to the API are set as absolute (OCE´s case), we will require to implement an interceptor in the Angular applications.

### Add a CSRF interceptor to the App Module

import{HttpClientXsrfModule} from '@angular/common/http';

import { HttpXsrfInterceptor } from './hero.interceptor';

import { HTTP\_INTERCEPTORS } from '@angular/common/http';

…

@NgModule({

imports: [

…

//CSRF

HttpClientXsrfModule,

…

],

…

//CSRF

providers:[

…

HttpXsrfInterceptor,

{ provide: HTTP\_INTERCEPTORS, useClass: HttpXsrfInterceptor, multi: true }

…

],

### Create the interceptor

import { Injectable } from '@angular/core';

import { HttpInterceptor, HttpRequest,HttpHandler,HttpEvent,HttpXsrfTokenExtractor } from '@angular/common/http';

import { Observable} from 'rxjs';

@Injectable()

export class HttpXsrfInterceptor implements HttpInterceptor {

constructor(private tokenExtractor: HttpXsrfTokenExtractor) {

}

intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

const headerName ='X-XSRF-TOKEN';

let token = this.tokenExtractor.getToken() as string;

if (token !== null && !req.headers.has(headerName)) {

req = req.clone({ headers: req.headers.set(headerName, token) });

}

return next.handle(req);

}

}

## APIGee & Nevis

The Cookie and Set-Cookie headers have to be accepted and forwarded by APIGee & Nevis.

# Add the CSRF header in POST/PUT/DELETE requests

The interceptor described in the previous section will also automatically add in the api request a header with the value of the CSRF token.

# Verify that the header has the right token

The API will have to check that the value specified in the header X-XSRF-TOKEN has the same value as the token included in the cookie XSRF-TOKEN.

This Enforcement can be performed at two different levels:

|  | **Pros** | **Cons** |
| --- | --- | --- |
| APIGee | * Will be valid regardless of the technology used to implement the MiSe (node or spring) * Will target the APIs that are exposed to the SPA | * A new custom policy will be required. The policy will validate that header and cookie are present in the request, and have the same value |
| Microservice | * Spring provides libraries to support CSRF out of the box | * Calls originated in the backend should be exempt of the CSRF validation. These calls should have an specific resource * MiSe to MiSe calls will have to forward the Cookie and Header in order to have the CSRF validation passed in the target |

## Spring

The same configuration stablished for enabling the generation of the CSRF cookie will also stablish the Enforcement point. The validation will be performed “before” the check-permissions is reached.