aw (/)

CORS, Dart, Dropwizard and Shiro

October 08, 2014

I'm in the process of re-architecting an application, following the the four principles of modern web development (http://blog.awolski.com/is-it-time-to-get-off-the-jsf-bandwagon/). The existing application is a JSF (Primefaces) front end, backed by Hibernate and MySQL, with the front end and database layers glued together using CDI/Weld, all running on Tomcat, and deployed to Jelastic. I've recently replaced the PicketLink (http://picketlink.org/) security component with Shiro + Stormpath (https://stormpath.com/blog/stormpath-apache-shiro-love/).

Although it's not huge, I still classify it as a monolithic web application (http://blog.awolski.com/javaone-2013-decompose-that-war-architecting-for-adaptability-scalability-and-deployability/); it's got multiple components all packaged up into a single war file, which makes changing a small, isolated part of the app more cumbersome than it should be. So I thought while I'm in the process of learning about microservice architecture (http://www.activestate.com/blog/2014/08/microservices-and-paas-part-i), why not pull this one apart bit by bit to accelerate my education.

Anyway, I decided to extract a very small client component out into it's own Dropwizard (https://dropwizard.github.io/dropwizard/) rest api serving JSON. I've been playing around with Dropwizard for the last month or so, and I really, really like it. It gives you so much for so little effort. And starts up in no time at all. I've only just scratching the surface too. You can check out my progress at my GitHub repository (https://github.com/awolski/client-api).

I also decided to experiment with Dart (https://www.dartlang.org/) as web UI to rest api. I've heard a lot of good things about Dart, particularly about it's speed and productivity benefits. After booting my Dropwizard service, I used Dart's `HttpRequest` to make a call to the api:

```
HttpRequest.request("http://localhost:8888/clients", method: 'GET',
  requestHeaders: {
    "Accept": "application/json",
    'Authorization' : 'Basic <base_64_enconded_credentials>'
  }
).then((HttpRequest resp) {
  // do something with the response
});
```

Initially I was getting CORS errors:

```
XMLHttpRequest cannot load http://localhost:8888/clients.  \ No 'Access-Control-Allow-Origin'
 header is present on the requested resource. Origin 'http://localhost:8080' is therefore not
 allowed access.
Exception: Uncaught Error: Instance of '_XMLHttpRequestProgressEvent'
```

This was because I hadn't added cross origin headers to the client service, but this is easily solved by following the instructions in this blog post (http://jitterted.com/tidbits/2014/09/12/cors-for-dropwizard-0-7-x/):

```
import org.eclipse.jetty.servlets.CrossOriginFilter;
...

@Override
public void run(ClientConfiguration configuration, Environment environment) throws Exception
 {
    FilterRegistration.Dynamic filter = environment.servlets().addFilter("CORS", CrossOriginF
ilter.class);
    filter.setInitParameter(CrossOriginFilter.ALLOWED_METHODS_PARAM, "GET,PUT,POST,DELETE,OPT
IONS");
    filter.setInitParameter(CrossOriginFilter.ALLOWED_ORIGINS_PARAM, "*");
    filter.setInitParameter(CrossOriginFilter.ACCESS_CONTROL_ALLOW_ORIGIN_HEADER, "*");
    filter.setInitParameter(CrossOriginFilter.ALLOWED_HEADERS_PARAM, "Content-Type,Authorizat
ion,X-Requested-With,Content-Length,Accept,Origin");
    filter.addMappingForUrlPatterns(EnumSet.allOf(DispatcherType.class), true, "/*");
```

But even after adding the above `CrossOriginFilter` I was still getting what looked to be the same error — at first glance. Upon closer inspection I could see that the actual error was a 401 Unauthorized:

```
Failed to load resource: the server responded with a status of 401 (Unauthorized)
    http://localhost:8888/clients
XMLHttpRequest cannot load http://localhost:8888/clients. No 'Access-Control-Allow-Origin' he
ader is present on the requested resource. Origin 'http://localhost:8080' is therefore not al
lowed access.
Exception: Uncaught Error: Instance of '_XMLHttpRequestProgressEvent'
```

Turns out the problem was that Dart/Chromium was sending a preflight request (https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS#Preflighted_requests), which is an OPTIONS request to the other domain (client-api) in order to determine whether the actual request is safe to send. The `OPTIONS` request was being intercepted by Shiro's `org.apache.shiro.web.filter.authc.BasicAuthenticationFilter.isAccessAllowed` method, which was failing because the `Authorization` header wasn't added to the preflight request.

The solution was to override the `isAccessAllowed` method for Shiro's `BasicAuthenticationFilter`. First, create a subclass the returns true for `OPTIONS` requests:

```
@Override
protected boolean isAccessAllowed(ServletRequest request, ServletResponse response, Object ma
ppedValue) {
    HttpServletRequest httpRequest = WebUtils.toHttp(request);
    if ("OPTIONS".equals(httpRequest.getMethod())) {
        return true;
    }
    return super.isAccessAllowed(request, response, mappedValue);
}
```

Next, configure Shiro to use the new overridden method/class. In shiro.ini:

```
authcBasic = com.awolski.shiro.web.filter.CrossOriginBasicHttpAuthenticationFilter
```

Now the preflight `OPTIONS` request makes it through Shiro filter's authorization to the CrossOriginFilter, which adds the appropriate CORS headers to the response, and thus letting the subsequent GET (or other method) know that the cross origin request is allowed. This subsequent GET/POST (or other method) is authorized as normal.