MPASSid Social Login Idp Integration

Phil Smart

Version 0.9, 2018-11-20

Document Brief

The aim of this document is to explore the following:

1. Integration of the MPASSid Social User authentication extensions into the Shibboleth Java Identity Provider.
    a. With configuration and usage examples.
2. Understanding the usage of MPASSid authentication flows that use spring-social connectors for authentication and attribute resolution.
3. Understanding the usage of MPASSid authentication flows that use Nimbus for authentication and attribute resolution
4. A deeper look at the Nimbus SDK when used as a dependency of the IdP.

Integration Setup

The following initial steps were required to integrate the MPASSid social authentication source (master branch https://github.com/Digipalvelutehdas/MPASSid-proxy/tree/master/idp-authn-impl-socialuser) into the Shibboleth Identity Provider source for execution:

1. Added the `idp-authn-impl-socialuser` and `idp-authn-api-socialuser` projects as a dependency of the `idp-parent`. Another option would have been to add the `idp-authn-impl-socialuser`and `idp-authn-api-socialuser` projects as a sub module of `idp-parent` as like other authn implementations.
2. Integrated the `flows`, `views`, and `conf` files from the `idp-authn-impl-socialuser` into the `idp-conf` module (in `src/main/resources`).

MPASSid Spring Social Implementation

The `spring-scocial` project provides a library of connectors for connecting applications to Software as a Service API providers (for example Facebook, LinkedIn, and Twitter) [1: https://projects.spring.io/spring-social/]. Each connector provides domain models, API bindings, and convenience methods for accessing and interacting with those APIs. As those APIs use OAuth 1.0 and OAuth 2.0 for authorisation, spring-social also supports the OAuth 1.0 and 2.0 authorisation protocol, it does do not directly support OpenID Connect (OIDC), although it could help provide it [2: There are also third party extensions that do e.g. https://github.com/molindo/spring-social-openid, and other spring libraries e.g. spring-security-oauth2, could be combined with existing JWT libraries to provide this functionality].

The MPASSid spring-social authentication implementation for the Shibboleth Identity Provider supports Facebook, LinkedIn, Google, and Twitter. Authentication leverages the `spring-social` connectors to first, *authenticate* and *authorise* the user to the providers authorisation server using OAuth1/2 returning an authorisation code, exchanging the authorisation code for an `access_token`, and finally collecting user profile attributes using the providers API (usually the `/me` or `profile` endpoint).

NOTE	Nimbus is not required when using only the `spring-social` connectors. `spring-security-core` is also not a dependency of `spring-social`, only `spring-security-crypto`.

Configuring The IdP with the MPASSid Facebook AuthN

Authentication

Configuration of the Facebook authN flow largely follows that described in the projects README file [3: https://github.com/Digipalvelutehdas/MPASSid-proxy/tree/master/idp-authn-impl-socialuser]. However, because the source has been added as a dependency of the IdP (as described above in [Integration Setup]) some aspects have changed. Consequently, the setup is described again in this section.

NOTE	Setup here is in addition to those initial steps shown in [Integration Setup], where the facebook authn flows have been added to the `idp-conf:src/main/resources/flows/authn` folder.

External Servlet Setup

The `/Authn/SocialUser/*` servlet which is externally redirected to by the authentication flow `socialuserfacebook-authn-flow.xml`, is an annontated `@WebServlet` class called `SocialUserAuthServlet`. For this to work, either the servlet container (e.g. Jetty) must process the annotation, or the servlet must be added manually to the `web.xml` of the `idp-war` module. Here we adopt the later, for example:

```
<servlet>
       <servlet-name>SocialUserAuthServlet</servlet-name>
       <servlet-class>fi.okm.mpass.idp.authn.impl.SocialUserAuthServlet</servlet-
class>
   </servlet>
   <servlet-mapping>
       <servlet-name>SocialUserAuthServlet</servlet-name>
       <url-pattern>/Authn/SocialUser/*</url-pattern>
   </servlet-mapping>
```

Conf Setup

1. Make sure `socialuser-authn-config.xml` is within the `conf/authn` folder.
2. Register the Facebook authenntication flow by adding the Facebook authentication flow bean to `general-authn.xml` e.g:

```
<bean id="authn/SocialUserFacebook" parent="shibboleth.AuthenticationFlow"
          p:nonBrowserSupported="false" p:forcedAuthenticationSupported="true"/>
```

1. Add the following Social User event end-state ids to `/authn/authn-events-flow.xml`:

```
<end-state id="SocialUserException" />
<end-state id="SocialUserCanceled" />
```

1. Enable the authN flow in `idp.properties` e.g. `idp.authn.flows = SocialUserFacebook`

Attribute Resolution

After external authentication has finished, the `Subject` placed in the `ExternalAuthenticationContext` is used to create a `SocialUserContext` that is attached to the `AuthenticationContext`. The `SocialUserContext` includes all the attributes about the user, as obtained from Facebook's Graph API `/me` endpoint. Consequently, attributes are resolved from the `SocialUserContext` using a ScriptedAttributeDefinition e.g.

```
<resolver:AttributeDefinition id="mail" xsi:type="ad:Script">
        <resolver:AttributeEncoder xsi:type="enc:SAML1String"
name="urn:mace:dir:attribute-def:mail" encodeType="false" />
        <resolver:AttributeEncoder xsi:type="enc:SAML2String"
name="urn:oid:0.9.2342.19200300.100.1.3" friendlyName="mail" encodeType="false" />
        <ad:Script><![CDATA[
          authnContext =
resolutionContext.getParent().getSubcontext("net.shibboleth.idp.authn.context.Authenti
cationContext");
          socialContext =
authnContext.getSubcontext("fi.okm.mpass.idp.authn.context.SocialUserContext");
          mail.addValue(socialContext.getEmail());
        ]]></ad:Script>
    </resolver:AttributeDefinition>
```

This does not work when the IdP reuses an *active* IdPSession, as the `SocialUserContext` is not cached/stored within the AuthenticationResult. Hence the ScriptedAttributeDefinition throws an exception during the attribute resolution step.

AuthN Facebook Flow

Figure Facebook OAuth Flow shows the authentication flow for Facebook.

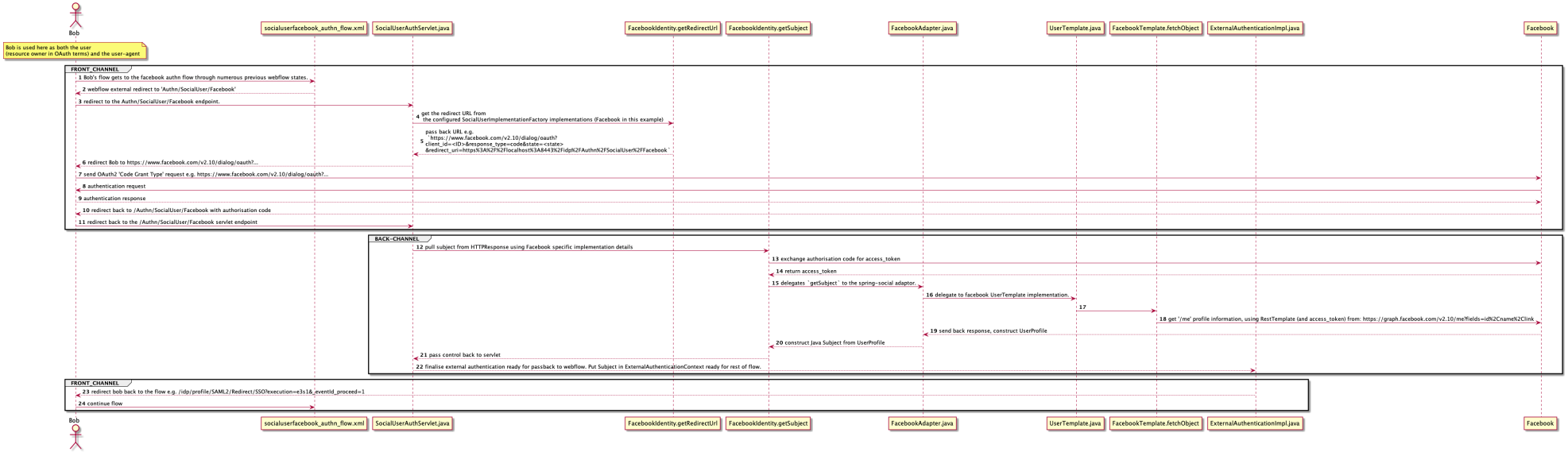

Figure 1. Facebook OAuth Flow

Issues and Lessons

1. As stated in the previous section, attribute resolution will not work during SSO if the IdP session layer is enabled. This could however be disabled, a session **should** exist with the authorization server (Facebook in this case), giving the effect of SSO to the user.
2. The version of `spring-social-facebook` used was requesting attributes that have been removed (see https://stackoverflow.com/questions/39890885/error-message-is-12-bio-field-is-deprecated-for-versions-v2-8-and-higher). Updated to an unreleased SNAPSHOT to resolve.
3. Spring-social provides a complete binding of providers APIs e.g. the Facebook Graph API. A significant amount of these features are not needed, and not used by the authentication implementation. This adds **unnecessary** dependency **bulk** to the IdP code base, although some of this functionality could be obtained using the Nimbus library.
4. As there is no standard specification for an OAuth2 token (unlike OpenID Connect id tokens) each integration is custom and, assuming identity attributes are requested, requires knowledge of their underlaying API to access user *profile* information [4: Although this is similar between implementations, and **could** be to some extend templated - as it has been in the MPASS sample `ExampleOauth2Identity`]. Consequently, it is difficult to see any of these (OAuth providers) provide an appropriate authentication mechanism for the IdP - unless explicitly required for some use case.

MPASSid OpenID Connect Authentication Implementation

The MPASSid authentication implementation also supports a more generic OpenID Connect (OIDC) flow. This bares much similarity to the OAuth flows discussed above, but with the added benefit of obtaining a standardised `id_token` (containing identity attributes etc.) as opposed to further having to access the providers custom APIs.

This implementation does **not** require `spring-social`, only **requiring** the Nimbus SDK. This simplifies transitive dependencies pulled into the IdP.

By default, the MPASSid OIDC flow supports the *authorisation_code* OAuth2 response type, with scope of `openid`. This works by an initial front-channel (from the user-agent) request to the providers authorisation server to obtain an `authorisation code` (as identical to a standard OAuth2 flow, but with an openid scope), this code is then exchanged by back-channel (from the IdP client) communication for an `id_token` (alongside an `access_token`) from the Token Endpoint. Once complete, the system assumes the subject has *authenticated* their identity and those attributes held in the `id_token`. An abstract example of this flow inside a SAML2 request (using HTTP-Redirect binding) is depicted in Figure OIDC Flow within the IdP (abstracted for simplicity, IdP discovery is not shown). below:

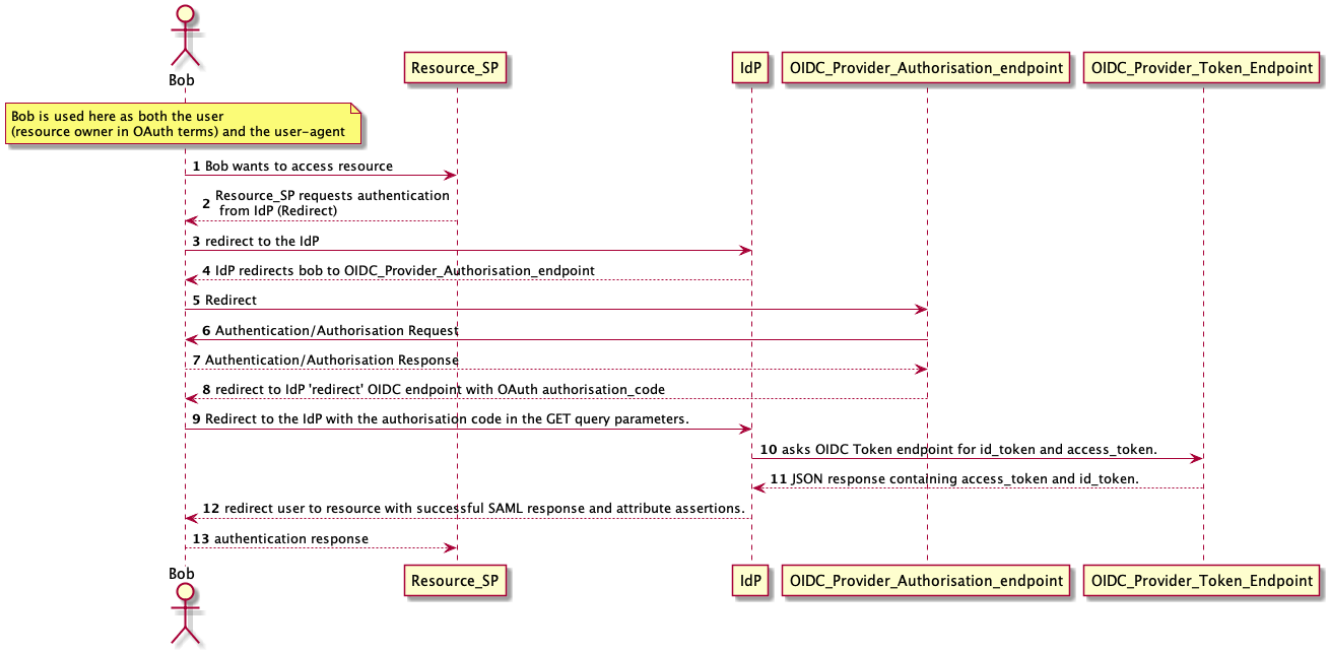

Figure 2. OIDC Flow within the IdP (abstracted for simplicity, IdP discovery is not shown).

NOTE	For all OAuth OpenID connect flows that give an **id_token**, `openid` is a required scope.

For reference, different OpenID Connect flows, based on the response type, are nicely summarised here https://medium.com/@darutk/diagrams-of-all-the-openid-connect-flows-6968e3990660.

Configuring The IdP with MPASSid OIDC AuthN

Authentication

The MPASSid README documentation does not contain specific instructions for enabling the OIDC flow `socialuseropenidconnect-authn-flow.xml` - although some of the general principals described are applicable.

First, two new Servlets need to be added to the idp-war `web.xml`. Both Servlets are used during external authentication. The first, `SocialUserOpenIdConnectStartServlet`, is used to initiate external authentication and redirect the user to the OpenID Connect Identity Provider's authorisation server. The Second, `SocialUserOpenIdConnectEndServlet`, consumes the response from the authorisation server, ending external authentication and redirecting the User (via the user-agent) back to the Spring Webflow.

```
<servlet>
        <servlet-name>SocialUserOpenIdConnectStartServlet</servlet-name>
        <servlet-class>
fi.okm.mpass.idp.authn.impl.SocialUserOpenIdConnectStartServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>SocialUserOpenIdConnectStartServlet</servlet-name>
        <url-pattern>/Authn/SocialUserOpenIdConnectStart</url-pattern>
    </servlet-mapping>

    <servlet>
        <servlet-name>SocialUserOpenIdConnectEndServlet</servlet-name>
        <servlet-class>
fi.okm.mpass.idp.authn.impl.SocialUserOpenIdConnectEndServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>SocialUserOpenIdConnectEndServlet</servlet-name>
        <url-pattern>/Authn/SocialUserOpenIdConnectEnd</url-pattern>
    </servlet-mapping>
```

NOTE	For standard deployers, the `@Servlet` annotation would be sufficient without having to change the `web.xml` - providing the servlet container's annotation scanning is enabled.

The AuthenticationFlow needs to be registered with the IdP in `general-auth.xml`:

```
<bean id="authn/SocialUserOpenIdConnect" parent="shibboleth.AuthenticationFlow"
            p:nonBrowserSupported="false" p:forcedAuthenticationSupported="true"/>
```

The AuthN flow in `idp.properties` needs to be enabled e.g. `idp.authn.flows = SocialUserOpenIdConnect`.

Setting up the OIDC Provider

The `SetOIDCInformation` bean needs to be configured in `socialuseropenidconnect-authn-beans.xml`, this includes:

- The client secret and client ID which you need to obtain from the provider - in this example Google.
- The redirect URL which the provider will (via front-channel) redirect the response to. Which **must** be the `SocialUserOpenIdConnectEnd` endpoint.
- The providers discovery metadata location, which contains the information necessary to interact with the provider e.g. token endpoint location, JSON Web Key location etc.
- The OAuth `Scopes`. These **must** contain `openid` if an `id_token` is to be retrieved from the *Token Endpoint* - the implementation adds this by default if not supplied. It also **should** contain the `email` and `profile` scopes if any meaningful identity information is to be retrieved [5: which

includes; name, email, picture, locale, see OIDCScopeValue.Profile]. Other scopes are available.

```
  <bean id="SetOIDCInformation" class=
"fi.okm.mpass.idp.authn.impl.SetOIDCInformation">
        <property name="clientId" value="ID" />
        <property name="clientSecret" value="SECRET" />
        <property name="providerMetadataLocation"
value="https://accounts.google.com/.well-known/openid-configuration" />
        <property name="redirectURI"
value="https://localhost:8443/idp/Authn/SocialUserOpenIdConnectEnd" />
        <property name="scope">
            <list>
                <value>openid</value> <!-- this is added by default if not set -->
                <value>email</value>
                <value>profile</value>
            </list>
        </property>
    </bean>
```

The default response type is *code*. This requires a second back-channel lookup to the *Token Endpoint* to retrieve an `id_token`. This **could** be changed to another response type e.g. by setting `<property name="responseType" value="id_token"/>`. Be aware, not all response types are valid in this OpenID Connect flow, as some e.g. `id_token`, return the `id_token` in the URL fragment for privacy, and are not forwarded onto the IdP.

It is also not presently possible to change the `response_mode` in the `SetOIDCInformation` class, only a `query` type is supported. However, a `fragment` mode would not work as the parameters are not returned to the server. `form-post` may, providing it is compatible with the `reponse_type` being used. Of Note, Google enforces a particular `response_mode` if you request the wrong type.

NOTE	The post external authentication `GetOIDCTokenResponse` action does check if a token exists in the response from the *Authorisation Endpoint*. Consequently a flow may exist that supported quick *id_token* retrieval, this would need to be checked.

Attribute Resolution

After external authentication has returned to the webflow, the `GetOIDCTokenResponse` is responsible for obtaining the `id_token` from the Token Endpoint using the authorisation code obtained from the external authentication step. The `ValidateOIDCAuthentication` action is then responsible for adding the `UsernamePrincipal` into the `principals` of the Java `Subject`. This is extracted from the `sub` in the `id_token` JWT response.

However, other claims are **not** extracted into the `Subject`, instead they are placed as a JWT *id_token* inside the `SocialUserOpenIdConnectContext`, which is itself attached to the `AuthenticationContext`. As a result, to obtain claims from the `id_token`, a (perhaps brittle) ScriptedAttributeDefinition can be used. For example:

```
<AttributeDefinition xsi:type="ad:Script" id="surname">
       <AttributeEncoder xsi:type="SAML1String" name="urn:mace:dir:attribute-def:sn"
encodeType="false" />
       <AttributeEncoder xsi:type="SAML2String" name="urn:oid:2.5.4.4"
friendlyName="sn" encodeType="false" />
       <ad:Script><![CDATA[
         authnContext =
resolutionContext.getParent().getSubcontext("net.shibboleth.idp.authn.context.Authenti
cationContext");
          socialContext =
authnContext.getSubcontext("fi.okm.mpass.idp.authn.impl.SocialUserOpenIdConnectContext
");

surname.addValue(socialContext.getIDToken().getJWTClaimsSet().getClaims().get("family_
name"));
       ]]></ad:Script>
   </AttributeDefinition>
```

As previously discussed with Facebook authentication, if the IdP session layer is active and the user attempts SSO, the existing session's authentication result is retrieved from the `IdPSession` rather than the OIDC token source. Hence in this scenario the `SocialUserOpenIdConnectContext` is not attached to the `AuthenticationContext` and the *id_token* is not available to extract claims from using the ScriptedAttributeDefinition.

This **maybe** improved by using a `SubjectDerivedAttributeAttributeDefinition`. For this to work a quick (hack) change was made such that all the claims from the OIDCToken were added as `IdPAttributePrincipal's` into the Java `Subject` (in the `ValidateOIDCAuthentication#populateSubject` method). These were then successfully resolvable as `SubjectDerivedAttribute` defintions e.g.

```
<AttributeDefinition xsi:type="SubjectDerivedAttribute" id="surname"
principalAttributeName="family_name">
</AttributeDefinition>
```

This also has the effect of being stored in the `AuthenticationResult` and cached for reuse in an SSO scenario - having been serialised using the `IdPAttributePrincipalSerializer` via the configured `StorageSerializer` in the `UpdateSessionWithAuthenticationResult` Action.

NOTE

A `SocialUserPrincipalSerializer` has been defined in the `idp-authn-impl-socialuser` module to support the serialisation of a `SocialUserPrincipal`. However the `SocialUserPrincipal` is not used in the OIDC flow implementation. The `SocialUserPrincipalSerializer` also does not appear to be used in the spring-social authn flows either e.g. Facebook, even though the `SocialUserPrincipal` is - the `SocialUserPrincipal` is attached to the `SocialUserContext` inside the `AuthenticationContext`, and is not added to the `AuthenticationResult` for storage.

Google OIDC Flow

Figure Google OIDC Flow shows the working authentication flow for Google using OpenID Connect.

NOTE	This sequence does not show the executed webflow before the **new** socialuseropenidconnect-authn-flow.xml is called, or what webflow is executed after that flow is finished. It also only documents the main OAuth2 interactions with Google, and not every **Action** of the socialuseropenidconnect-authn-flow.xml flow.

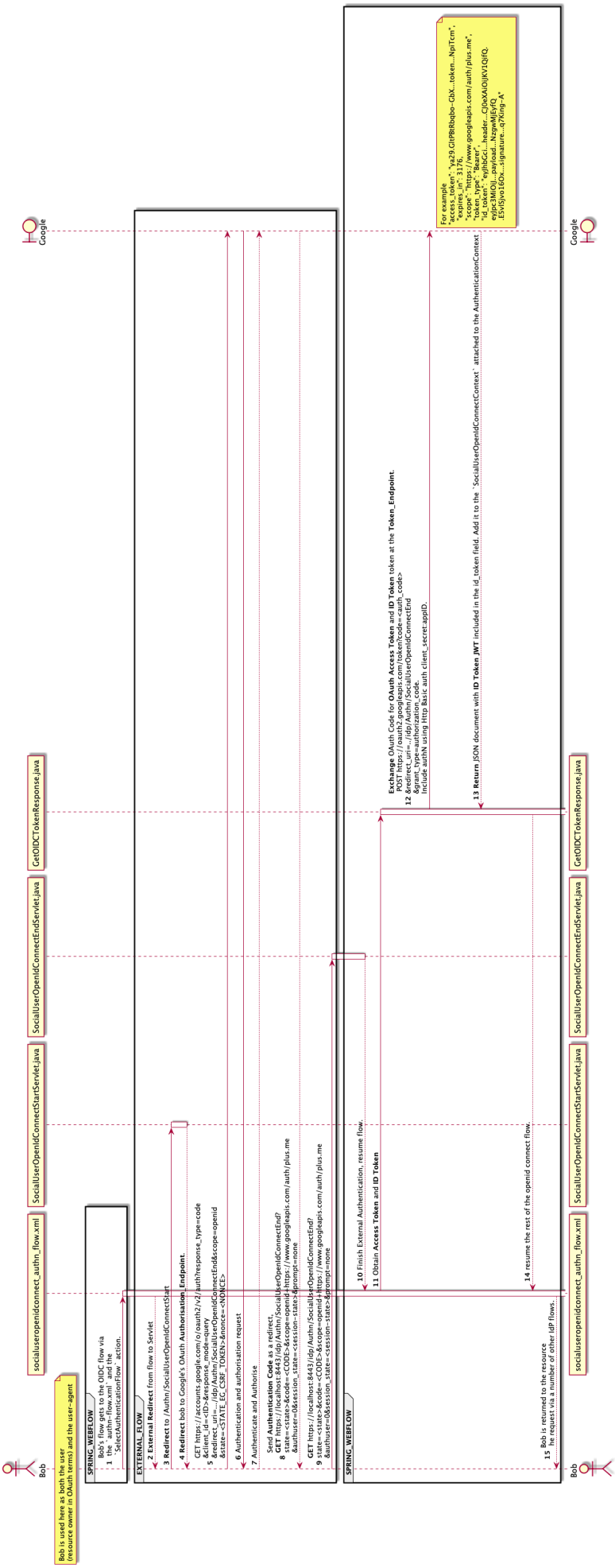

Figure 3. Google OIDC Flow

MPASSid OIDC Nimbus (v4.33) Information

The MPASSid OIDC authentication implementation delegates a number of functions e.g. `id_token` handling, signature verification, and OAuth2 HTTP flows, to the Nimbus SDK. Consequently, if the implementation is to be supported, Nimbus would need to be added to the IdP as a dependency. In this section, this following questions are answered:

1. How does Nimbus perform JSON handling.
2. What cryptographic extensions does Nimbus use to handle JWT encryption and signature validation.

JSON Handling

json-smart v1 provides JSON processing. Json-smart is a performance focused, JSON processor library.

json-smart v1 has not been updated in 3 years. A newer json-smart v2 is available, which has not been updated in 2 years. Lastest versions of both GitHub and Maven central is from March 2017. Despite seemingly being stale, there were no obvious issues with its usage.

json-smart is used by 97 other libraries - as recorded by maven central across all versions. For comparison there are 5,224 usages of V2 of jackson-core.

Cryptography Extension/Library Usage

Nimbus includes Bouncycastle 1.60 (And 1.55 for PKIX) as a dependency. This can either be used as an API to low level cryptographic functions, or as JCE provider.

Within the Nimbus SDK, there does **not** appear to be a requirement to use BouncyCastle as a JCE provider.

Google return a signed JSON Web Token (JWS). The signature is derived using the RSA-SHA256 algorithm. The SunRsaSign provider (from 1.4 onwards, tested 1.8) is sufficient to check this signature.

Of note, symmetric shared key HmacSHA256 MACs is the only required supported algorithm for JWS tokens (rfc7518). HMACs do not provide non-repudiation, making it more difficult (or impossible) to prove the message source. In this case, it seems better to support *signatures* using asymmetric keys - as Google do.

Google do not support Encrypted JSON Web Tokens (JWE). It appears common for OIDC providers to **not** provide support for encrypted `id_tokens` - although in this context this is perhaps less of an issue, as the `id_token` it is only acquired via back-channel communication.

The A128CBC-HS256 and A256CBC-HS512 encryption algorithms are the only required encryption algorithms for encrypting the claims inside a JWE (rfc7518). The Encrypted Key (encryption of the claims Content Encryption Key) does not seem to specify a required algorithm, an example would be RSA-OAEP. There are of course numerous optional and recommended algorithms in the JSON Web Algorithms spec [6: https://tools.ietf.org/html/rfc7518]. It becomes a topic in itself to determine

which is supported by which JCE/JCA provider. Such work could be undertaken if required.

BouncyCastle itself is not initialised as a security provider at runtime e.g. via `Security.addProvider(new BouncyCastleProvider());`, and there is no mention of adding it to the `security.providers` file.

Nimbus does include a singleton accessor pattern for retrieving a BouncyCastleProvider (`BouncyCastleProviderSingleton`) at runtime. However, that does not appear to be used within the Nimbus source.

Bouncy Castle **is** used directly (as a light-weight API) in the `ECKey` class, to parse an Elliptic Curve JWK X509 Certificate via their `JcaX509CertificateHolder` class. This (as is PKIX) requires the `bcpkix-jdk15on` dependency.

Other

`jose.4.j` (JavaScript Object Signing and Encryption) JWS and JWE token handling is an included *test* library, used by unit test classes.

Issues and Lessons

1. Spring-social and spring-security are not required for the generic MPASSid OIDC provider. Nimbus is a requirement, a rough measure of its usage can be seen in appendix [Nimbus JOSE+JWT Usage] - which shows source code execution during a SAML2 SSO Redirect flow.
    a. JSON parsing within Nimbus is handled mostly by a third party library (json-smart). Although from inspection looks like it could be replaced by Jackson as supported by the IdP.
2. It appears Bouncycastle is only used as a lightweight API to support PKIX certificates. SunJCE and SunRsaSign are used and seem sufficient as the JCE/JCA providers to support JSON Signed and JSON Encrypted tokens using both required and optional algorithms - athough that would need further investigation to be sure.
3. The OIDC MPASSid flows are early stage, do not seem complete w.r.t to the spec, and are not finalised w.r.t to the code. They would probably need some cleanup before they could be integrated.
4. Despite the *authorisation code* OAuth2 flow not providing the openid connect *id_token* in the response from the initial request to the authorisation server, the `GetOIDCTokenResponse` class does check for its existence. Hence, a flow where this is possible could be explored - although probably unnecessary.
    a. Some of the MPASSid Spring Webflow Actions could be split using *Decision States* e.g. in the example above, check for the `id_token` first, then proceed to different action states based on the result. This would make the flow more readable.

Identity Providers

OIDC Providers

1. **Google** have an OpenID Connect implementation which conforms to the specifications and is

OpenID Connect Certified. It also supports the OpenID Connect Discovery Specification (Service Metadata). This works with the generic OIDC authn flows provided by MPASSid.

2. **Microsoft**, ADFS Supports OpenID Connect, so does AzureAD. Not tested.

OAuth Providers

- **Facebook** is **not** an openID Connect provider. Facebook use custom, closed, extensions to OAuth2 - termed `Facebook Connect`. The MPASSid social IdP extension for Facebook uses OAuth2 to provide authentication, alongside a second call to the Facebook Graph API `/me` endpoint to resolve attributes (during the authentication stage).
- **Twitter** supports an OAuth 1.0a flow for API access. Although an OAuth 2.0 flow seems possible from **some** of their docs.
- **LinkedIn** does not support OpenID Connect. Only OAuth2. Interestingly they also support a, redirected, passive, SAML authentication inside an OAuth authorisation code flow.

Other OIDC Complaint Providers

- See https://openid.net/developers/certified/ for a list of other OpenID Connect Identity Providers.

Appendix A: SAML2 SSO Redirect Flow with MPASSid OIDC AuthN

Each Spring Webflow *STATE* for a SAML2 SSO Redirect flow, including the 'STATE's within the SocialUserOpenIDConnect flow, are depicted in the state diagram below. The MPASSid Action classes are described in the table below that.

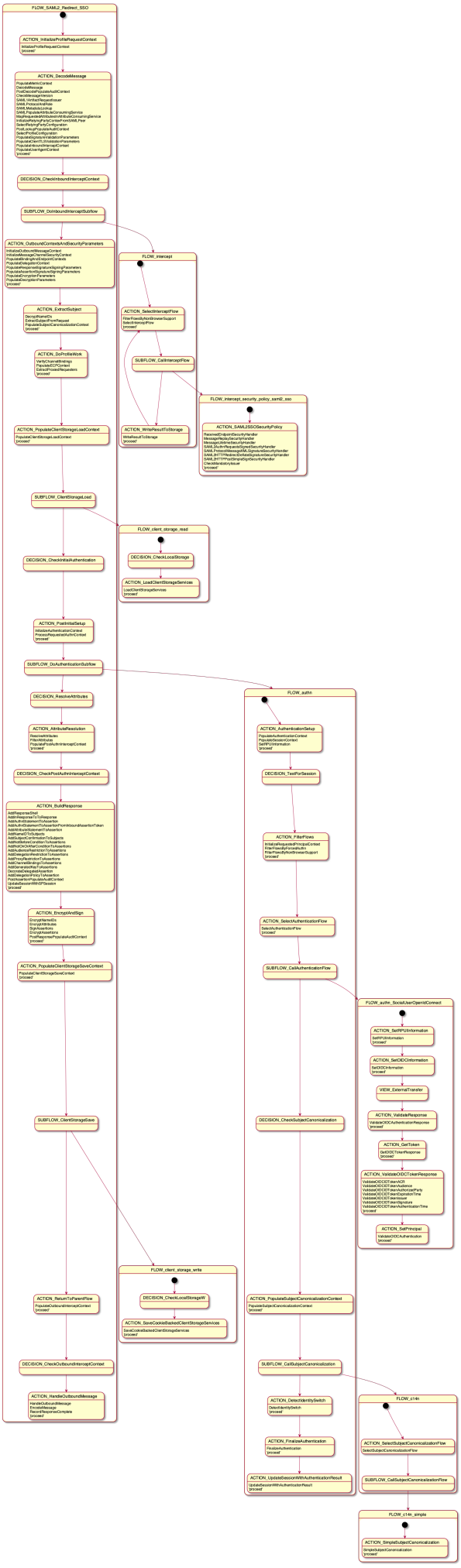
FLOW_SAML2_Redirect_SSO
ACTION_InitialisesProfileRequestContext
ACTION_DecodeMessage
DECISION_CheckInboundInterceptContext
SUBFLOW_DoInboundInterceptSubflow
ACTION_OutboundContextsAndSecurityParameters
ACTION_ExtractSubject
ACTION_DoProfileWork
ACTION_PopulateClientStorageLoadContext
SUBFLOW_ClientStorageLoad
DECISION_CheckInitialAuthentication
ACTION_PostInitialSetup
SUBFLOW_DoAuthenticationSubflow
DECISION_ResolveAttributes
ACTION_AttributeResolution
DECISION_CheckPostAuthnInterceptContext
ACTION_BuildResponse
ACTION_EncryptAndSign
ACTION_PopulateClientStorageSaveContext
SUBFLOW_ClientStorageSave
ACTION_ReturnToParentFlow
DECISION_CheckOutboundInterceptContext
ACTION_HandleOutboundMessage
FLOW_intercept
ACTION_SelectInterceptFlow
SUBFLOW_CallInterceptFlow
ACTION_WriteResultToStorage
FLOW_intercept_security_policy_saml2_sso
ACTION_SAML2SSOSecurityPolicy
FLOW_client_storage_read
DECISION_CheckLocalStorage
ACTION_LoadClientStorageServices
FLOW_client_storage_write
DECISION_CheckLocalStorageW
ACTION_SaveCookieBackedClientStorageServices
FLOW_authn
ACTION_AuthenticationSetup
DECISION_TestForSession
ACTION_FilterFlows
ACTION_SelectAuthenticationFlow
SUBFLOW_CallAuthenticationFlow
DECISION_CheckSubjectCanonicalization
ACTION_PopulateSubjectCanonicalizationContext
SUBFLOW_CallSubjectCanonicalization
ACTION_DetectIdentitySwitch
ACTION_FinalizeAuthentication
ACTION_UpdateSessionWithAuthenticationResult
FLOW_authn_SocialUserOpenIdConnect
ACTION_SetIPIUtilInformation
ACTION_SetOIDCInformation
VIEW_ExternalTransfer
ACTION_ValidateResponse
ACTION_GetToken
ACTION_ValidateOIDCTokenResponse
ACTION_SetPrincipal
FLOW_c14n
ACTION_SelectSubjectCanonicalizationFlow
SUBFLOW_CallSubjectCanonicalizationFlow
FLOW_c14n_simple
ACTION_SimpleSubjectCanonicalization

Table 1. MPASSid Classes and their function

Class	Function
SetOIDCInformation	Creates a SocialUserOpenIdConnectContext with OIDC provider details and adds it to the Authentication Context.
SocialUserOpenIdConnectStartServlet	Called from the webflow to begin ExternalAuthentication. Populates the HTTPSession with a SocialUserOpenIdConnectContext. Redirects the user to the OAuth AUTHORISATION server endpoint for the OIDC Provider.
SocialUserOpenIdConnectEndServlet	Handles the response from the OAuth AUTHORISATION server endpoint. Ends the ExternalAuthentication. Returns to the webflow.
GetOIDCTokenResponse	Checks if the AUTHORISATION server response already contains an id_token, if not, requests the id_token from the Token Endpoint using the AUTH_TOKEN in the response.
ValidateOIDCAuthenticationResponse	Checks for successful HTTP response from OIDC provider was returned, adds id_token and AuthenticationSuccessResponse to SocialUserOpenIdConnectContext
ValidateOIDCIDTokenACR	Verifiess Authentication Context Class Reference claims from id_token if present - although incomplete, does not actually check ACR.
ValidateOIDCIDTokenAudience	Checks the audience of the id_token is the same as the client that requested it (so checks the client_id configured against that presented in the id_token)
ValidateOIDCIDTokenAuthenticationTime	verifies Authentication Time of ID Token is within an acceptable *skew* and configured authentication lifetime.
ValidateOIDCIDTokenAuthorizedParty	Verifies the Authorised Party of ID Token. Checks the *azp* value in the id_token is the same as the configured client_id. Not sure this is complete w.r.t to the spec.
ValidateOIDCIDTokenExpirationTime	Checks the current processing time is before the token expiration time.
ValidateOIDCIDTokenIssuer	Verifies that the issuer of the id_token is the same as the one presented by the Providers metadata.
ValidateOIDCAuthentication	Sets the UsernamePrincipal in the Java Subject.

SocialUserOpenIdConnectContext	Context that holds OIDC information required during webflow processing.

Appendix B: Nimbus JOSE+JWT Usage

11.1% of the Nimbus JOSE+JWT library is used for a first login (new IdP session) to the Google OIDC Identity Provider. In more detail, the degree of source code execution for the Nimbus library - for the most significant classes - is shown in the table below.

Of course code coverage is not a particularly good metric for judging the value of the library. It can be said that many of these classes are entity/model (beans) style classes which hold various JWT components. e.g. to hold the parsed AccessToken and the JWT ID Token. The others are utility functions which could potentially be replaced - but that would need further investigation. Of note, this could be repeated for the OAuth part of the Nimbus SDK as well for a complete picture.

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
nimbus-jose-jwt src/main/java **(ALL Lib)**	**11.1 %**	2,503	20,123	22,626
SignedJWT.java	62.5 %	35	21	56
JWTParser.java	45.2 %	33	40	73
JWTClaimsSet.java	39.7 %	325	493	818
Base64Codec.java	92.2 %	567	48	615
Base64.java	47.1 %	40	45	85
JSONObjectUtils.java	23.0 %	46	154	200
Algorithm.java	70.6 %	48	20	68
JWSHeader.java	46.4 %	235	272	507
JWSObject.java	44.8 %	145	179	324
JOSEObject.java	41.9 %	126	175	301
JWSAlgorithm.java	37.9 %	100	164	264
JOSEObjectType.java	36.4 %	24	42	66
Payload.java	29.3 %	120	289	409
Header.java	20.3 %	55	216	271
CommonSEHeader.java	17.9 %	35	161	196
JWKMetadata.java	59.8 %	52	35	87
KeyUse.java	39.2 %	60	93	153
KeyType.java	27.8 %	45	117	162
JWK.java	11.0 %	33	267	300
RSAKey.java	10.5 %	174	1,487	1,661

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
RSASSAProvider.java	100.0 %	36	0	36
BaseJWSProvider.java	68.0 %	17	8	25
RSASSAVerifier.java	53.4 %	47	41	88
CriticalHeaderParamsDeferral.java	37.0 %	20	34	54
RSASSA.java	15.2 %	19	106	125