AuthenticationConfiguration

Current File(s): *conf/authn/*, views/login.vm, conf/c14n/**

Format: Native Spring

Legacy V2 File(s): *conf/handler.xml, login.jsp, others*

- Overview
- General Configuration
- Login Flows
- Post-Login Canonicalization
- Advanced Features
    - Custom Events
    - "Initial Authentication"
    - Attribute Lookup
    - Extending the Serialization of Results 3.3
- References
    - Beans
    - Properties
- V2 Compatibility
- Notes

Overview

The files in **conf/authn/** collectively configure authentication both generally and the specific mechanisms that are built-in to the software. You will likely only ever look at a few of them, depending on which login methods you want to use. Authentication is heavily based on Spring Web Flow; selection of the mechanisms to invoke and each individual mechanism are implemented as "subflows" that run inside of the web flows that service requests.

Authentication subflows are the equivalent of V2 LoginHandler plugins and make use of SWF's support for presentation views and state transitions instead of the servlet-based design in V2. However, because of SWF's handling of URLs, there are cases in which use of a subflow exclusively is problematic, and so a defined External method for branching out to a servlet design is also provided.

Comprehensive support for complex mechanism selection criteria has been implemented; assuming prior configuration, all of the SAML 2 operators (exact, minimum, maximum, better) for requesting authentication context types are now supported.

While fully decoupled from the authentication layer, session management is obviously related. Most session management configuration is automatic, and the properties that can override the defaults are documented below. It's possible to fully disable the session mechanism and turn off SSO globally with a single property. Note that unlike V2, the default session mechanism in V3 is a client-side storage option using an encrypted cookie. This mechanism depends on the configuration of a secret key, which is typically handled automatically during installation or upgrade.

Authentication configuration is divided into general and mechanism-specific parts. Separate topics exist for each mechanism included with the software and certain other subtopics.

See the Authentication topic for extensive discussion of the design of this layer, and how to use and extend it (e.g. how to build a custom login flow of your own design).

General Configuration

The *authn/general-authn.xml* file is where all supported login flows are described to the system. They can be enabled and disabled with a property, so even when defined they can be globally activated or deactivated without editing this file.

The file contains a Spring list bean named **shibboleth.AvailableAuthenticationFlows.** Each mechanism for authentication is called an authentication flow, and each flow has a descriptor inside this list. Descriptors are of a specific class, and include a number of properties that influence whether a mechanism can be used for specific requests.

The `id` property of each descriptor is **not** arbitrary. It MUST be prefixed by "authn/" and it corresponds to a web flow definition. The predefined beans correspond to built-in flows. Creating a new flow involves not only describing the flow in this list, but ensuring the `id` matches a flow definition created inside *flows/authn/*. Specifically, creating the custom login flow "authn/foo" requires that the flow definition file be named *flows/authn/foo/foo-flow.xml*.

Even if a flow is defined to the system, it is not necessarily available for use at runtime. The overall list of enabled flows is controlled using the **idp.authn.flows** property that expresses the flows to enable. In addition, a list of flow IDs (minus the "authn/" prefix) can be configured per-profile, per-relying-party using the `authenticationFlows` property on a profile configuration bean. Any flow not enabled will be ignored.

relying-party.xml example enabling a login flow for a specific profile

```
...
<bean parent="SAML2.SSO" p:authenticationFlows="#{{'RemoteUser'}}" />
...
```

The parent bean for all flows is defined in a system file and looks like this:

Parent bean for Authentication Flow Descriptors

```
    <bean id="shibboleth.AuthenticationFlow" abstract="true"
            class="net.shibboleth.idp.authn.AuthenticationFlowDescriptor"
            p:resultSerializer-ref="shibboleth.
DefaultAuthenticationResultSerializer"
            p:passiveAuthenticationSupported="false"
            p:forcedAuthenticationSupported="false"
            p:nonBrowserSupported="true"
            p:lifetime="%{idp.authn.defaultLifetime:PT60M}"
            p:inactivityTimeout="%{idp.authn.defaultTimeout:PT30M}">
        <property name="supportedPrincipals">
            <list>
                <bean parent="shibboleth.SAML2AuthnContextClassRef"
                    c:classRef="urn:oasis:names:tc:SAML:2.0:ac:classes:
PasswordProtectedTransport" />
                <bean parent="shibboleth.SAML2AuthnContextClassRef"
                    c:classRef="urn:oasis:names:tc:SAML:2.0:ac:classes:
Password" />
                <bean parent="shibboleth.SAML1AuthenticationMethod"
                    c:method="urn:oasis:names:tc:SAML:1.0:am:password" />
            </list>
        </property>
    </bean>
```

The comments in *general-authn.xml* and the system bean above describe some of the defaults, and illustrate how to define a non-password-based mechanism (IPAddress or X509), which requires overriding the `supportedPrincipals` property to prevent misuse. This property is used to associate a flow with any number of custom objects that represent, in the specific case of SAML, SAML 1 AuthenticationMethod or SAML 2 AuthnContextClassRef or AuthnContextDeclRef constants. These are URI constants that represent different kinds of authentication. They can also be used to represent more advanced concepts such as identity proofing and auditing requirements which go beyond just technology choices.

To associate these constants with a flow, beans are used that inherit from one of the following built-in beans:

- **shibboleth.SAML2AuthnContextClassRef**
- **shibboleth.SAML2AuthnContextDeclRef**
- **shibboleth.SAML1AuthenticationMethod**

Note that if you associate more than one such constant/bean with a flow, the IdP will ordinarily pick one at random to use as a SAML "result" when it builds an assertion, assuming the SP didn't request a specific one. If you need to guarantee that a particular one will be used as a default, you can assign "weights" to them in a map bean that's defined in the *general-authn.xml* file named **shibboleth.AuthenticationPrincipalWeightMap**.

As a caution, if you add a custom flow or make use of one of the technology-independent flows in a way that is not password-based, don't forget to override the `supportedPrincipals` property, or you will create problems for any SPs that are attempting to request stronger authentication methods. It's very easy to accidentally "lie" to the world by misconfiguring the IdP, because the software can't read your mind to understand your intentions, and it only knows what you tell it. A complete explanation of how the selection process works is here.

Login Flows

Login flows provided with the software are listed below:

- Password
- RemoteUser
- RemoteUserInternal
- X509
- X509Internal
- SPNEGO / Kerberos [3.2]
- IPAddress
- External
- Duo [3.3]
- Multi-Factor [3.3]
- Function [3.4]

Post-Login Canonicalization

After a successful login takes place, the IdP needs to be able to establish an "official" username to represent the subject throughout the system in order to disambiguate subjects from each other. Chiefly this is needed so that the IdP can detect when the identity of the subject has changed /switched, usually because of the use of shared computers. It also allows you to build an AttributeResolverConfiguration that can count on the form of the subject's name when looking up entries in directories or databases, essentially offloading some of the complexity that would otherwise end up in the attribute lookup logic.

This particular use of subject canonicalization is referred to as post-login c14n and the input is an instance of a Java Subject, which is a fairly open-ended object that can contain any number of custom Principal objects and collections of public and/or private credentials.

With the built-in login flows, the Subject is typically fairly simple and often contains a UsernamePrincipal that carries a username set by the login flow, often something entered by a user. In the simple case, the "right" canonicalized value may just be that name, and nothing needs to be configured, it will simply happen automatically. In less simple cases, you may need to configure a more advanced c14n subflow to do your bidding, or create one of your own design.

The post-login c14n flows provided with the software are listed below:

- simple
- x500
- attribute

The first two are enabled by default, while the third requires additional configuration to operate sensibly.

Advanced Features

Custom Events

More typically in the Password or MFA flows, but generally applicable, it's possible for login flows to return custom Spring Web Flow events in addition to the events that are hardwired into the system. This typically arises as a result of mapping a specific error message to an event by means of a "classified message" map, or in advanced cases might be triggered by a button or other user input on a login form to cause some other login method to execute.

To support this, it's not sufficient to just signal or map a message to the event; you must also define the event to the system by editing *conf/authn/authn-events-flow.xml*. There are two steps needed: adding an `<end-state>` for the event and adding a `<transition>` rule to authorize the event as a signal to cause the flow to terminate with that state. The file as of V3.3.1 includes commented examples, and a sample is shown below that authorizes "MyCustomEvent" to be surfaced:

Authorizing custom login flow events in conf/authn/authn-events-flow.xml

```
    <end-state id="MyCustomEvent" />

    <global-transitions>
        <transition on="MyCustomEvent" to="MyCustomEvent" />
        <transition on="#{!'proceed'.equals(currentEvent.id)}" to="
InvalidEvent" />
    </global-transitions>
```

In many cases, you also want this custom event to result in a custom error message/page at the IdP rather than a response to the SP. This isn't the default for most events, but you can modify the **shibboleth.LocalEventMap** bean in *conf/errors.xml* to change the behavior, and add custom error messages for the standard error view to *messages/messages.properties*. See ErrorHandlingConfiguration for details.

"Initial Authentication"

> This feature pre-dates the development of the Multi-Factor login flow, and in virtually all cases it should be used in place of this feature. **It is deprecated and will be removed in V4.0.**
>
> In addition, you should consider the value of the **idp.authn.identitySwitchIsError** property. The default of "false" can result in anomalous behavior of the authenticated identity produced during the "regular" authentication sequence doesn't match the value produced initially.

Initial authentication refers to login flows to run when no pre-existing session exists, and that run regardless of standard preferences, configuration, or what the SP requests. This typically will be a password-based flow that might be upgraded later to a multi-factor result depending on who the subject is and what the request looks like.

In all respects, the result of an initial flow is the same as any other result and is preserved in a new session created for the subject, and if the request later determines that the initial flow is good enough to satisfy the request, then the user won't see any further challenges.

To enable initial authentication, set the **idp.authn.flows.initial** property in a similar fashion to the **idp.authn.flows** property to select the flow(s) to run. If a session already exists, this setting will have no impact on the request. For obvious reasons, don't combine this feature with disabling the session mechanism.

Attribute Lookup

> This feature pre-dates the development of the Multi-Factor login flow, which should in virtually all cases should be used in place of this feature. **It is deprecated and will be removed in V4.0**.
>
> In addition, you should consider the value of the **idp.authn.identitySwitchIsError** property. The default of "false" can result in anomalous behavior of the authenticated identity produced during the "regular" authentication sequence doesn't match the value produced initially, because the attributes retrieved will be based on the original identity.

You can set the **idp.authn.resolveAttribute** property to the name of an attribute to lookup using the AttributeResolverConfiguration. This only occurs if a pre-existing session identifies the subject to look up attributes for. The values of the attribute to resolve will be used to filter the login flows to allow by comparing the values against the names of the `supportedPrincipals` attached to the flow definition.

Beginning with V3.2, by default and if the **idp.authn.filterActiveResultsByAttribute** property is true, the values are also applied to filter the active authentication results to potentially reuse for SSO.

You can, if necessary, provide a comma-delimited list of attribute names to resolve in the property, but the first attribute in the list is the one used to filter flows and results.

Extending the Serialization of Results [3.3]

When deploying custom authentication flows, you may need to teach the IdP how to store the results your flows create. The system comes with support built-in for storing results produced by LDAP and some of the other custom Principal types used by the IdP for its own use, and as of V3.3 it's possible to extend this set. If you need to preserve custom Principal types the IdP doesn't know about, you will need to create a plugin that implements the PrincipalSerializer interface, and register it with the system by defining a new bean in *conf/authn/general-authn.xml* that merges your custom plugin definition into the list of default serializers:

Registering a Custom PrincipalSerializer

```
<!-- Define anywhere in general-authn.xml -->
<bean id="shibboleth.PrincipalSerializers" parent="shibboleth.
DefaultPrincipalSerializers">
        <property name="sourceList">
                <list merge="true">
                        <bean class="org.example.MyPrincipalSerializer" />
                </list>
        </property>
</bean>
```

References

Beans

Beans defined for general authentication configuration follow:

Bean ID	Type	Function
shibboleth.AvailableAuthenticationFlows	List<AuthenticationFlowDescriptor>	List of descriptors enumerating the supported authentication flows that can be used
shibboleth.AuthenticationPrincipalWeightMap 3.1	Map<Principal,Integer>	Map of weights to assign to particular custom Principal objects so that flows can pick an appropriate default Principal to associate with their result (see comment in file for more detail)
shibboleth.AuthenticationFlow	AuthenticationFlowDescriptor	Parent bean for defining new flow descriptors
shibboleth.SAML2AuthnContextClassRef	AuthnContextClassRefPrincipal	Parent bean for attaching SAML 2.0 AuthnContextClassRef constants to flows
shibboleth.SAML2AuthnContextDeclRef	AuthnContextDeclRefPrincipal	Parent bean for attaching SAML 2.0 AuthnContextDeclRef constants to flows
shibboleth.SAML1AuthenticationMethod	AuthenticationMethodPrincipal	Parent bean for attaching SAML 1.1 AuthenticationMethod constants to flows
shibboleth.DefaultPrincipalSerializers 3.3	List<PrincipalSerializer>	Default list of principal serializer plugins needed to support storage of authentication results
shibboleth.DefaultPrincipalSymbolics 3.3	Map<String,Integer>	Default mappings that shrink authentication result data by storing commonly seen strings as numbers
shibboleth.PrincipalSerializers 3.3	List<PrincipalSerializer>	User-supplied list of values to merge into shibboleth.DefaultPrincipalSerializers bean
shibboleth.PrincipalSymbolics 3.3	Map<String,Integer>	User-supplied list of values to merge into shibboleth.DefaultPrincipalSymbolics bean
shibboleth.FixedAuthenticationEventStrategy 3.4	Function<ProfileRequestContext,String>	Function for producing a Spring WebFlow Event to signal from login flow validation actions to artifically test error or warning customizations.

The following beans are used to configure comparison rules for custom Principals to support rules for login flow selection when requests specify particular methods, as described in AuthenticationFlowSelection.

Bean ID	Type	Function
shibboleth.AuthnComparisonRules	Map used as constructor argument to PrincipalEvalPredicateFactoryRegistry	Map of comparison rules
shibboleth.SAMLAuthnMethodExact shibboleth.SAMLACClassRefExact shibboleth.SAMLACDeclRefExact shibboleth.SAMLACClassRefMinimum shibboleth.SAMLACDeclRefMinimum shibboleth.SAMLACClassRefMaximum shibboleth.SAMLACDeclRefMaximum shibboleth.SAMLACClassRefBetter shibboleth.SAMLACDeclRefBetter	Pair<Class<? extends Principal>, String>	Pairs of custom Principal types and matching operators for all the SAML 1.1 and 2.0 principal and comparison types supported, used as keys for the **shibboleth.AuthnComparisonRules** map
shibboleth.ExactMatchFactory shibboleth.InexactMatchFactory	PrincipalEvalPredicateFactory	Template beans for values of the **shibboleth.AuthnComparisonRules** map
shibboleth.BetterClassRefMatchFactory shibboleth.MinimumClassRefMatchFactory shibboleth.MaximumClassRefMatchFactory shibboleth.BetterDeclRefMatchFactory shibboleth.MinimumDeclRefMatchFactory shibboleth. MaximumDeclRefMatchFactory	PrincipalEvalPredicateFactory	Beans supplying matching rules for implementing SAML 2.0 "inexact" comparisons of AuthnContextClassRef or AuthnContextDeclRef constants
shibboleth.IgnoredContexts [3.2]	Collection<String>	A collection of SAML 2.0 AuthnContextClassRef or AuthnContextDeclRef values to ignore if found in an <AuthnRequest> message

Properties

Because there are a variety of different login methods, most of the actual configuration is in per-method Spring configuration files, but there are a few general properties used, and a larger set of properties that control the management of sessions, which are of course related to authentication.

Worthy of note, you can switch to server-side storage of user sessions (essentially similar to the V2 default) by setting the **idp.session.StorageService** property to **shibboleth.StorageService**, or an alternative defined by you. See StorageConfiguration for details.

The most important property to note, and the only one that **MUST** be set is the **idp.authn.flows** property. This is a regular expression that identifies the flows defined in *general-authn.xml* to enable. The expression applies only to the suffix of each flow ID (omitting the authn/ prefix), and the simplest way to express this is with a pipe (|) separated list of the flow names, e.g., flow1|flow2|flow3

Property	Type	Default	Function
idp.session.enabled	Boolean	true	Whether to enable the IdP's session tracking feature
idp.session.StorageService	Bean ID of StorageService	shibboleth. ClientSessionStorageService	Bean name of a storage implementation/configuration to use for IdP sessions
idp.session.idSize	Integer	32	Number of characters in IdP session identifiers
idp.session.consistentAddress	Boolean	true	Whether to bind IdP sessions to IP addresses
idp.session.timeout	Duration	PT60M	Inactivity timeout policy for IdP sessions (must be non-zero)
idp.session.slop	Duration	0	Extra time after expiration before removing SP sessions in case a logout is invoked
idp.session.maskStorageFailure	Boolean	false	Whether to hide storage failures from users during session cache reads/writes
idp.session.trackSPSessions	Boolean	false	Whether to save a record of every SP accessed during an IdP session (requires a server-side session store or HTML LocalStorage)

idp.session.secondaryServiceIndex	Boolean	false	Whether to track SPs on the basis of the SAML subject ID used, for logout purposes (requires SP session tracking be on)
idp.session.defaultSPlifetime	Duration	PT2H	Default length of time to maintain record of an SP session (must be non-zero), overridable by relying-party-specific setting
idp.authn.flows	Regular Expression		Expression that identifies the login flows to globally enable
idp.authn.flows.initial	Regular Expression		**DEPRECATED** Expression that identifies login flows used when no IdP session exists (see Advanced)
idp.authn.resolveAttribute	String		**DEPRECATED** Name of attribute (or a comma-sep'd list of names) to resolve before authentication (see Advanced)
idp.authn.filterActiveResultsByAttribute [3.2]	Boolean	true	Whether to apply the results of the attribute resolved by the previous property against active results as well as inactive flows (see Advanced)
idp.authn.defaultLifetime	Duration	PT60M	Default amount of time to allow reuse prior authentication flows, measured since first usage
idp.authn.defaultTimeout	Duration	PT30M	Default inactivity timeout to prevent reuse of prior authentication flows, measured since last usage
idp.authn.favorSSO	Boolean	false	Whether to prioritize prior authentication results when an SP requests more than one possible matching method (V2 behavior was to favor them)
idp.authn.rpui [3.3]	Boolean	true	Whether to populate information about the relying party into the tree for user interfaces during login and interceptors
idp.authn.identitySwitchIsError	Boolean	false	Whether to fail requests if a user identity after authentication doesn't match the identity in a pre-existing session.

V2 Compatibility

Authentication has been substantially redesigned to make customization easier and add more capability, but some compatibility has been maintained when using JAAS (UsernamePassword) or container-based (RemoteUser) authentication.

The V2 *handler.xml* file that defined login handlers and authentication "methods" is no longer supported. Compatibility is provided only for specific aspects of integrating authentication with the local environment.

If the V2 UsernamePassword login handler was used, the V3 equivalent is the Password flow with the JAAS back-end; a similar JAAS configuration can typically be used, but the most common case (LDAP) is an exception because the underlying LDAP library has changed. By convention this configuration is placed in *authn/jaas.config* and the legacy-matching "ShibUserPassAuth" login configuration name is used (though this can be changed). The UI for password-based login is no longer strictly JSP-based as in V2, but is now a Web Flow view: this can use Velocity, JSP, or potentially other view technologies. The default */login.vm* view provided uses Velocity. Using an older */login.jsp* file will require some changes, although the V2 JSP taglibs for metadata-driven UI information should still work. We recommend modernizing to the use of Velocity, as this is much simpler in most cases.

If the 2.x RemoteUser login handler was used, the V3 equivalent is the RemoteUser flow, and for compatibility this has been designed to use the same protected endpoint by default (/Authn/RemoteUser) as in V2. This allows for reuse of existing container authentication configuration.

So in short, activate flows with the **idp.authn.flows** property, transfer JAAS or web.xml and container configuration over, and you should have **basic** compatibility working, apart from the actual login UI for JAAS-based authentication.

In most cases, existing *relying-party.xml* `defaultAuthenticationMethod` settings will result in the expected login flows running, but in some cases you may need to alter the `supportedPrincipals` property of one or more flow descriptors in *authn/general-authn.xml*.

A distinction of note is that V2 did not properly separate the constants used for SAML 1.1 AuthenticationMethod from those used for SAML 2.0 AuthnContextClassRef. These tend to be different, at least for built-in values native to SAML, and the new defaults reflect this.

Notes