MetadataDrivenConfiguration

This feature is available in V3.4 and later of the software.

Overview

One of the challenges of dealing with interoperability issues with SPs is maintaining all the custom configuration rules needed to drive the IdP's behavior in all kinds of different ways. Traditionally most of these kinds of settings are concentrated in the *relying-party.xml* file and involve defining "overrides" for SPs or groups of SPs, often through manually maintained lists of entityIDs.

As discussed in the RelyingPartyConfiguration topic in the section on dynamically-driven configuration, V3.3 introduced an advanced capability to produce these settings on the fly at runtime, which creates opportunities for more flexible configuration, at some cost in complexity. One of the techniques currently described in that section involves the use of metadata "tags" (a shorthand name for the `<mdattr:EntityAttributes>` metadata extension feature) to embed signals in metadata that can drive configuration. In the longer term, any hope for a simplified approach to configuration or even a GUI probably rests on a strategy of this general type.

V3.4 introduces a built-in layer of code and Spring wiring to enable the use of the metadata-driven approach to configuration in a systematic way that provides a set of conventions for tag names and values to use to drive a significant range of configuration settings. With a small amount of up-front adjustment to the *relying-party.xml* file, it's possible to enable comprehensive support for metadata-driven configuration that can be freely intermixed in most cases with more static settings as required.

Bear in mind that enabling this feature requires a total degree of trust and control over one's sources of metadata, because the information in the metadata can have a wide-ranging impact on the behavior of the IdP, by design. It essentially off-loads pieces of your configuration to the metadata, with all that that entails.

Relying Party Configuration

Support for this feature has been added to the runtime environment in a way that avoids overhead (and risk) if not used but allows for straightforward enablement on a per-profile basis. The existing parent beans used for defining relying party defaults, overrides, and profile configurations all have analogs with a ".MDDriven" suffix in their names. Changing the existing parent bean name by adding the suffix and reloading the configuration will cause the runtime to evaluate most of the commonly-used configuration settings by checking for appropriately-named "tags" in the SP's metadata before falling back to the default, or statically configured, value if no tag is found.

For example, if you would like to enable tag-driven configuration for SAML 2.0 Browser SSO, you can replace any reference to the **SAML2.SSO** bean with **SAML2.SSO.MDDriven,** as in this example:

Enable tag-driven features in relying-party.xml

```
<bean id="shibboleth.DefaultRelyingParty" parent="RelyingParty">
        <property name="profileConfigurations">
                <list>
                        <ref bean="Shibboleth.SSO" />
                        <ref bean="SAML1.AttributeQuery" />
                        <ref bean="SAML1.ArtifactResolution" />

                        <ref bean="SAML2.SSO.MDDriven" />

                        <ref bean="SAML2.ECP" />
                        <ref bean="SAML2.Logout" />
                        <ref bean="SAML2.AttributeQuery" />
                        <ref bean="SAML2.ArtifactResolution" />
                </list>
        </property>
</bean>
```

Attribute / Property Convention

The "bridge" between the metadata and this feature is the naming and syntax of the attribute tags used to control the configuration properties in the IdP. As an implementation strategy this could be "brute forced" with arbitrary mapping dictionaries, but the supplied implementation takes the approach of automating a mapping between the names of settings and the identifiers used internally for the various profiles, and the names of the corresponding SAML Attributes in metadata for those profile/setting combinations.

Each setting has a "base" name matching its Java bean property name as would be used if the setting were explicitly configured in Spring. For example, the property set via `p:defaultAuthenticationMethods` is named "defaultAuthenticationMethods".

The corresponding SAML Attribute for a setting is named by suffixing the "base" name of the setting to a profile URL that is defined by the Shibboleth software for each of the supported profiles.

The URLs are as follows:

Profile	Profile URL
Shibboleth.SSO	`http://shibboleth.net/ns/profiles/saml1/sso/browser`
SAML1.AttributeQuery	`http://shibboleth.net/ns/profiles/saml1/query/attribute`
SAML1.ArtifactResolution	`http://shibboleth.net/ns/profiles/saml1/query/artifact`
SAML2.SSO	`http://shibboleth.net/ns/profiles/saml2/sso/browser`
SAML2.ECP	`http://shibboleth.net/ns/profiles/saml2/sso/ecp`
SAML2.AttributeQuery	`http://shibboleth.net/ns/profiles/saml2/query/attribute`
SAML2.ArtifactResolution	`http://shibboleth.net/ns/profiles/saml2/query/artifact`
SAML2.Logout	`http://shibboleth.net/ns/profiles/saml2/logout`
Liberty.SSOS	`http://shibboleth.net/ns/profiles/liberty/ssos`
CAS.LoginConfiguration	`https://www.apereo.org/cas/protocol/login`
CAS.ProxyConfiguration	`https://www.apereo.org/cas/protocol/proxy`
CAS.ValidateConfiguration	`https://www.apereo.org/cas/protocol/serviceValidate`

It follows that the `includeAttributeStatement` property of the "Shibboleth.SSO" profile configuration can be set via a metadata Attribute named "`http://shibboleth.net/ns/profiles/saml1/sso/browser/includeAttributeStatement`".

As an additional convention, a setting can be configured for all profiles simultaneously by prefixing it with the URL "`http://shibboleth.net/ns/profiles`".

We reserve the right to define behavior for any current or future SAML Attributes named in the `shibboleth.net` domain or any other URI we own and control, so if any developers wish to develop general purpose extensions or behavior based on such tags, you should either rely on your own tag names or seek permission from the project.

Type Conversion

The supplied implementations support various built-in type conversions supporting a natural mapping between simple XML syntax and Java data types. The only XML syntaxes supported are "simple content" models involving an `<AttributeValue>` containing only text content, but it is possible to apply specific `xsi:type` designations that trigger more precise handling (such as enforcing numeric or boolean data). Different kinds of settings support particular XML syntaxes as described below.

Setting Data Type	Supported XML Conversions	Notes
String	Untyped, string, boolean, integer, dateTime, base64binary	Booleans are mapped to "0" or "1". Dates are mapped to the Unix epoch, converted to String.
Boolean	Untyped, string, boolean, integer	Strings are processed as a valid XML boolean value (0, 1, true, false) or treated as false. Non-zero integers are true, zero is false.
Integer	Untyped, string, boolean, integer	Strings are decoded via Integer.decode() method. Booleans are mapped to 0 or 1.
Long	Untyped, string, boolean, integer, dateTime	Strings are decoded via Long.decode() method. Booleans are mapped to 0 or 1. Dates are mapped to the Unix epoch, converted to a Long.
Double	Untyped, string, boolean, integer	Strings are decoded via Double.valueOf() method. Booleans are mapped to 0.0 or 1.0.
Duration	Untyped, string, integer	Strings are converted from the ISO Duration notation used throughout the software. Integers are treated as a millisecond duration.
List<?>	Untyped, string, boolean, integer, dateTime, base64binary	Supports multiple values and each value is converted to a String and then used to construct an object of the type specified for the property via a String-based single-argument constructor.
Set<?>	Untyped, string, boolean, integer, dateTime, base64binary	Supports multiple values and each value is converted to a String and then used to construct an object of the type specified for the property via a String-based single-argument constructor.
Bean	Untyped, string	Converted to a String used as a name of a Spring bean to build or access

Applying Tags

There are two ways to apply tags to metadata: directly and indirectly.

The direct method applies to cases in which you control the metadata or when the source of the metadata supports the inclusion of the necessary extensions. The tags simply appear within the metadata being loaded into the IdP using an `<mdattr:EntityAttributes>` extension element:

Direct embedding of configuration tags

```
<EntityDescriptor xmlns="urn:oasis:names:tc:SAML:2.0:metadata"
                  xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
                  xmlns:mdattr="urn:oasis:names:tc:SAML:metadata:attribute"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                  entityID="https://sp.example.org">
    <Extensions>
        <mdattr:EntityAttributes>
            <saml:Attribute Name="http://shibboleth.net/ns/profiles/defaultAuthenticationMethods"
                NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
                <saml:AttributeValue>http://example.org/ac/classes/mfa</saml:AttributeValue>
            </saml:Attribute>
            <saml:Attribute Name="http://shibboleth.net/ns/profiles/saml2/sso/browser/signResponses"
                NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
                <saml:AttributeValue xsi:type="xsd:boolean">false</saml:AttributeValue>
            </saml:Attribute>
        </mdattr:EntityAttributes>
    </Extensions>
    <SPSSODescriptor protocolSupportEnumeration="urn:oasis:names:tc:SAML:2.0:protocol">
        <!-- Content omitted -->
    </SPSSODescriptor>
</EntityDescriptor>
```

The indirect method uses the EntityAttributesFilter to attach the tags at runtime after loading and verifying the metadata. This is useful for cases in which you want to rely on externally supplied metadata but still organize parts of your configuration around the metadata. One use for this is simply consistency: if you have both external and local metadata, you can drive the configuration with both.

Indirect example of applying tags using a filter

```
<MetadataProvider id="InCommonMD" xsi:type="FileBackedHTTPMetadataProvider"
        metadataURL="http://md.incommon.org/InCommon/InCommon-metadata.xml"
        backingFile="%{idp.home}/metadata/InCommon-metadata.xml"
        failFastInitialization="false">
        <MetadataFilter xsi:type="RequiredValidUntil" maxValidityInterval="P14D" />
        <MetadataFilter xsi:type="SignatureValidation" requireSignedRoot="true"
                certificateFile="%{idp.home}/credentials/incommon.pem" />
        <MetadataFilter xsi:type="EntityRoleWhiteList">
                <RetainedRole>md:SPSSODescriptor</RetainedRole>
        </MetadataFilter>
        <MetadataFilter xsi:type="EntityAttributes">
                <saml:Attribute Name="http://shibboleth.net/ns/profiles/defaultAuthenticationMethods"
                        NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
                        <saml:AttributeValue>http://example.org/ac/classes/mfa</saml:AttributeValue>
                </saml:Attribute>
                <saml:Attribute Name="http://shibboleth.net/ns/profiles/saml2/sso/browser/signResponses"
                        NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
                        <saml:AttributeValue xsi:type="xsd:boolean">false</saml:AttributeValue>
                </saml:Attribute>
                <Entity>https://sp.example.org</Entity>
        </MetadataFilter>
</MetadataProvider>
```

Relying Party Configuration Examples

The following examples illustrate possible `<Attribute>` syntaxes that can be used to configure a variety of common features. The examples apply equally to either the direct or indirect methods of attaching the attributes to the metadata.

Signed Assertions vs. Responses

Enabling signed assertions for a particular SP is advisedly handled by setting the `WantAssertionsSigned` XML attribute in metadata, but isn't always possible and sometimes has to be combined with disabling signed responses (or just for efficiency).

SAML 2 Browser SSO: Sign assertions and not responses

```
<saml:Attribute Name="http://shibboleth.net/ns/profiles/saml2/sso/browser/signResponses"
        NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
        <saml:AttributeValue xsi:type="xsd:boolean">false</saml:AttributeValue>
</saml:Attribute>
<saml:Attribute Name="http://shibboleth.net/ns/profiles/saml2/sso/browser/signAssertions"
        NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
        <saml:AttributeValue xsi:type="xsd:boolean">true</saml:AttributeValue>
</saml:Attribute>
```

Disabling Encryption

Setting **idp.encryption.optional** is usually a workaround for handling the majority of SPs without encryption support, but there are a couple of scenarios in which it's useful to be able to manually disable it. Some federations (InCommon for one, at least for the time being) have limitations such that SPs without encryption support are stuck registering keys they don't support. Some SPs support encryption but build in time-bombs by forcing flag day key rotations on all IdPs that cause outages or manual work. Or you may simply want to drive this more explicitly for only a subset of SPs and not as a global change.

Note that it may be helpful to make sure encryption also doesn't get used during logout, so the second tag may be useful.

All profiles: turn off encryption

```
<saml:Attribute Name="http://shibboleth.net/ns/profiles/encryptAssertions"
        NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
        <saml:AttributeValue xsi:type="xsd:boolean">false</saml:AttributeValue>
</saml:Attribute>
<saml:Attribute Name="http://shibboleth.net/ns/profiles/encryptNameIDs"
        NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
        <saml:AttributeValue xsi:type="xsd:boolean">false</saml:AttributeValue>
</saml:Attribute>
```

Algorithms

There are metadata extensions that are meant to be used to signal algorithm support, but they're not widely used at this point.

An occasional scenario is to force SHA-1 for older systems. This relies on identifying the name of a Spring bean to use to supply an object of the appropriate type, in this case, a SecurityConfiguration object.

All profiles: use SHA-1

```
<saml:Attribute Name="http://shibboleth.net/ns/profiles/securityConfiguration"
        NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
        <saml:AttributeValue>shibboleth.SecurityConfiguration.SHA1</saml:AttributeValue>
</saml:Attribute>
```

Another scenario that will be increasingly important is to upgrade to AES-GCM encryption. This is similar:

All profiles: use AES-GCM

```
<saml:Attribute Name="http://shibboleth.net/ns/profiles/securityConfiguration"
        NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
        <saml:AttributeValue>shibboleth.SecurityConfiguration.GCM</saml:AttributeValue>
</saml:Attribute>
```

For other algorithm use cases, you would need to wire up an appropriate SecurityConfiguration bean to identify with the tag. Note that if you identify a bean of the wrong type, the system will detect and reject this.

NameID Format Selection

Using `<NameIDFormat>` elements in metadata (which can also be added at runtime with the NameIDFormatFilter) is the normal way to use metadata to select the Format to use, but the "unspecified" Format has to be triggered with a profile setting. That's not commonly needed, but it's easy to define a tag for in such cases (and if you prefer you can obviously designate any Format this way).

All profiles: use the "unspecified" NameID Format

```
<saml:Attribute Name="http://shibboleth.net/ns/profiles/nameIDFormatPrecedence"
        NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
        <saml:AttributeValue>urn:oasis:names:tc:SAML:1.1:nameid-format:unspecified</saml:AttributeValue>
</saml:Attribute>
```

One concern is that any two SPs using it are only coincidentally going to want the same data, so this isn't always solely a matter of Format selection. In such cases, it may be necessary to add activationCondition properties based on SP to specific generators that are handling the same Format.

Bug Workarounds

Some of the other profile settings are workarounds for bugs, e.g., omitting the `NotBefore` attribute. This is not very common but easily tag-driven.

All profiles: omit the NotBefore attribute

```
<saml:Attribute Name="http://shibboleth.net/ns/profiles/includeConditionsNotBefore"
        NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
        <saml:AttributeValue xsi:type="xsd:boolean">false</saml:AttributeValue>
</saml:Attribute>
```

A more obscure example would be an SP that requires additional `<Audience>` values in addition to its own entityID. This mainly demonstrates that some settings may be multi-valued.

All profiles: add additional Audience values

```
<saml:Attribute Name="http://shibboleth.net/ns/profiles/assertionAudiences"
        NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
        <saml:AttributeValue>https://example.org/bogusAudience1</saml:AttributeValue>
        <saml:AttributeValue>https://example.org/bogusAudience2</saml:AttributeValue>
</saml:Attribute>
```

Forcing MFA

Handling SPs that require MFA but can't request it requires IdP-side configuration, usually involving a couple of settings. The URL below is just an example, it has to be replaced by whatever `<AuthnContextClassRef>` constant is used in a deployment to signal MFA, or possibly more than one.

All profiles: forcing MFA

```
<saml:Attribute Name="http://shibboleth.net/ns/profiles/defaultAuthenticationMethods"
        NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
        <saml:AttributeValue>http://example.org/ac/classes/mfa</saml:AttributeValue>
</saml:Attribute>
<!-- The disallowedFeatures setting is a bitmask, and 0x1 blocks SPs requesting authentication types. -->
<saml:Attribute Name="http://shibboleth.net/ns/profiles/disallowedFeatures"
        NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
        <saml:AttributeValue>0x1</saml:AttributeValue>
</saml:Attribute
```

Interceptor Flows

Triggering consent based on the SP is pretty common.

SAML and CAS SSO: enable attribute consent

```
<saml:Attribute Name="http://shibboleth.net/ns/profiles/postAuthenticationFlows"
        NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
        <saml:AttributeValue>attribute-release</saml:AttributeValue>
</saml:Attribute>
```

The authorization intereptor flow is another case, though the checking logic would have to be extended for each different service/scenario. In the example, both flows are enabled.

SAML and CAS SSO: enable authorization checking and attribute consent

```
<saml:Attribute Name="http://shibboleth.net/ns/profiles/postAuthenticationFlows"
        NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri">
        <saml:AttributeValue>context-check</saml:AttributeValue>
        <saml:AttributeValue>attribute-release</saml:AttributeValue>
</saml:Attribute
```

Legacy Profiles

A common use case is enabling SAML 1 for legacy systems, often combined with either enabling queries or attribute push (to eliminate the queries). This is not handled as well via this mechanism because one generally enables most profiles by default, and it's not practical to enable them and then try and disable them for all but a few SPs. This goal is the opposite, disable by default and enable the profiles for a few exceptions.

Therefore, the built-in wiring cannot accomodate this use case, but it's possible to do this in a couple of different ways, one that works in pre-3.4 versions by defining a relying party override, and one that avoids the need for an override but requires API changes specific to V3.4. A common tag can be used in both cases for consistency.

First, an example making use of the support for profile configuration activation conditions in V3.4. This is the same wiring that is provided by the system configuration for many of the other properties but is supplied here "by hand" in order to adjust the default value of the condition to "false" instead of "true".

V3.4+: Enabling SAML 1.1 SSO conditionally using a tag

```
<bean id="shibboleth.DefaultRelyingParty" parent="RelyingParty.MDDriven">
    <property name="profileConfigurations">
        <list>
            <bean parent="Shibboleth.SSO.MDDriven">
                <property name="activationCondition">
                    <bean class="net.shibboleth.utilities.java.support.logic.PredicateSupport" factory-method="
fromFunction">
                        <constructor-arg>
                            <bean parent="shibboleth.MDDrivenBoolProperty" p:propertyName="activationCondition"
/>
                        </constructor-arg>
                        <constructor-arg value="false" />
                    </bean>
                </property>
            </bean>
            <ref bean="SAML2.SSO.MDDriven" />
            <ref bean="SAML2.ECP.MDDriven" />
            <ref bean="SAML2.Logout.MDDriven" />
            <ref bean="SAML2.AttributeQuery.MDDriven" />
            <ref bean="SAML2.ArtifactResolution.MDDriven" />
        </list>
    </property>
</bean>
```

Second, the override example that works for older versions using the same attribute tag for consistency.

V3.3: Enabling SAML 1.1 SSO conditionally using a tag

```
<!-- Assumption is that the Shibboleth.SSO profile bean is left out of the DefaultRelyingParty. -->

<util:list id="shibboleth.RelyingPartyOverrides">
        <bean parent="RelyingPartyByTag">
                <constructor-arg name="candidates">
                        <list>
                                <bean parent="TagCandidate"
                                        c:name="http://shibboleth.net/ns/profiles/saml1/sso/browser
/activationCondition"
                                        p:values="#{{'1', 'true'}}"/>
                        </list>
                </constructor-arg>
                <property name="profileConfigurations">
                        <list>
                                <ref bean="Shibboleth.SSO" />
                        </list>
                </property>
        </bean>
</util:list>
```

Attribute Filter Examples

The AttributeFilterConfiguration has had support for metadata-driven configuration for a while now, but it hasn't been extensively used. A logical approach is to align usage with the property-driven model outlined above and move to a per-attribute policy model, in contrast to the fairly common model today of defining policies around services. Since the filter layer operates by iterating over all policies to determine if they apply, it may be more efficient for larger policy sets to redesign policies around tags that signal release of each individual attribute.

To facilitate the sharing of examples, building of tools, and a more useful set of default rules in the software, we have agreed to reserve the following SAML Attribute for use in constructing filter policies:

Name: `http://shibboleth.net/ns/attributes/releaseAllValues`

NameFormat: `urn:oasis:names:tc:SAML:2.0:attrname-format:uri`

Values: Each value is an internal Attribute ID whose values should be released to an SP whose metadata contains the relevant tag/value.

Example

Here's an example policy (more or less matching an example in the default file) that applies this tag test to a couple of attributes. A couple of subtle points here.

One is that this approach is really a purely "local" one because the IdP Attribute names are local/internal only. You might assume most people use similar conventions and you'd be right, but you can't expect that to be true universally, so it is **not** appropriate to ever try and use this kind of metadata tag outside of a scenario that does not involve control of the IdP. It's not a fix for the brokenness of the more general SAML `<RequestedAttribute>` metadata element, which is unusable for a variety of reasons.

Less critically, note that this policy example demonstrates an optimization in that it applies the tag check in the `<AttributeRule>`(s) and not up in the `<PolicyRequirementRule>`. This has the advantage of requiring less XML to express and is about the same or better in performance than running multiple policies. It's not as good for an attribute that might have 3-5 or more values, since that would force the cost of checking for the tag to be paid for each value.

```
<AttributeFilterPolicy id="Per-Attribute-singleValued">
        <PolicyRequirementRule xsi:type="ANY"/>

        <AttributeRule attributeID="eduPersonPrincipalName">
                <PermitValueRule xsi:type="EntityAttributeExactMatch"
                        attributeName="http://shibboleth.net/ns/attributes/releaseAllValues"
                        attributeNameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri"
                        attributeValue="eduPersonPrincipalName" />
        </AttributeRule>

        <AttributeRule attributeID="mail">
                <PermitValueRule xsi:type="EntityAttributeExactMatch"
                        attributeName="http://shibboleth.net/ns/attributes/releaseAllValues"
                        attributeNameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri"
                        attributeValue="mail" />
        </AttributeRule>
</AttributeFilterPolicy>
```

Reference

Bean ID	Type	Function
RelyingParty. MDDriven	RelyingPartyConfiguration	A template bean for use in defining metadata-driven RelyingParty overrides by hand
RelyingPartyByName. MDDriven	RelyingPartyConfiguration	A template bean for defining metadata-driven RelyingParty overrides based on matching by name
RelyingPartyByGroup. MDDriven	RelyingPartyConfiguration	A template bean for defining metadata-driven RelyingParty overrides based on matching by `<EntitiesDescriptor>` groups
RelyingPartyByTag. MDDriven	RelyingPartyConfiguration	A template bean for defining metadata-driven RelyingParty overrides based on matching `<EntityAttributes>` extension content
Shibboleth.SSO. MDDriven	BrowserSSOProfileConfiguration	Default metadata-driven configuration for SAML 1.1 SSO profile
SAML1. AttributeQuery. MDDriven	AttributeQueryProfileConfiguration	Default metadata-driven configuration for SAML 1.1 Attribute Query profile
SAML1. ArtifactResolution. MDDriven	ArtifactResolutionProfileConfiguration	Default metadata-driven configuration for SAML 1.1 Artifact Resolution profile
SAML2.SSO. MDDriven	BrowserSSOProfileConfiguration	Default metadata-driven configuration for SAML 2.0 SSO profile
SAML2.ECP. MDDriven	ECPProfileConfiguration	Default metadata-driven configuration for SAML 2.0 Enhanced Client/Proxy profile
SAML2.Logout. MDDriven	SingleLogoutProfileConfiguration	Default metadata-driven configuration for SAML 2.0 Single Logout profile
SAML2. AttributeQuery. MDDriven	AttributeQueryProfileConfiguration	Default metadata-driven configuration for SAML 2.0 Attribute Query profile
SAML2. ArtifactResolution. MDDriven	ArtifactResolutionProfileConfiguration	Default metadata-driven configuration for SAML 2.0 Artifact Resolution profile
Liberty.SSOS. MDDriven	SSOSProfileConfiguration	Default metadata-driven configuration for Liberty ID-WSF Delegated SSO profile
CAS. LoginConfiguration. MDDriven	LoginConfiguration	Default metadata-driven configuration for CAS login prototol
CAS. ProxyConfiguration. MDDriven	ProxyConfiguration	Default metadata-driven configuration for CAS proxy login protocol

CAS.ValidateConfiguration.MDDriven	ValidateConfiguration	Default metadata-driven configuration for CAS ticket validation protocol
shibboleth.DefaultMDProfileAliases	List<String>	A built-in list of alternate URL "prefixes" to property names, this is used to automate the generation of property tag names that apply to all profiles at the same time.
shibboleth.MDProfileAliases	List<String>	An optional user-supplied list of additional URL prefixes to support custom property tag names
shibboleth.MDDrivenStringProperty	StringConfigurationLookupStrategy	Parent bean for defining new lookup strategies for string settings
shibboleth.MDDrivenBoolProperty	BooleanConfigurationLookupStrategy	Parent bean for defining new lookup strategies for boolean settings
shibboleth.MDDrivenIntProperty	IntegerConfigurationLookupStrategy	Parent bean for defining new lookup strategies for integer settings
shibboleth.MDDrivenLongProperty	LongConfigurationLookupStrategy	Parent bean for defining new lookup strategies for long integer settings
shibboleth.MDDrivenDoubleProperty	DoubleConfigurationLookupStrategy	Parent bean for defining new lookup strategies for double settings
shibboleth.MDDrivenDurationProperty	DurationConfigurationLookupStrategy	Parent bean for defining new lookup strategies for Duration settings
shibboleth.MDDrivenListProperty	ListConfigurationLookupStrategy	Parent bean for defining new lookup strategies for List settings
shibboleth.MDDrivenSetProperty	SetConfigurationLookupStrategy	Parent bean for defining new lookup strategies for Set settings
shibboleth.MDDrivenBeanProperty	BeanConfigurationLookupStrategy	Parent bean for defining new lookup strategies for arbitrary Spring bean settings