Spring Security SAML Extension

Reference Documentation

1.0.1.RELEASE

Vladimír Schäfer

Table of Contents

I. Getting Started ... 1
1. Introduction ... 2
1.1. What this manual covers ... 2
1.2. When to use Spring Security SAML Extension ... 2
1.3. Features and supported profiles ... 2
1.4. Requirements ... 3
1.5. Source code ... 3
1.6. Builds ... 3
1.7. License ... 3
1.8. Issue tracking ... 3
1.9. Contributions ... 3
1.10. Commercial support ... 3
1.11. Community support ... 4
1.12. Dependencies ... 4
2. What's new ... 5
2.1. New features, improvements and fixes in 1.0.1.FINAL ... 5
2.2. New features, improvements and fixes in 1.0.0.FINAL ... 5
2.3. Important code changes in 1.0.0.FINAL ... 6
3. Glossary ... 8
4. Quick start guide ... 10
4.1. Pre-requisites ... 10
4.2. Installation steps ... 10
Downloading sample application ... 10
Configuration of IDP metadata ... 10
Generation of SP metadata ... 11
Compilation ... 11
Deployment ... 11
Uploading of SP metadata to the IDP ... 11
4.3. Testing single sign-on and single logout ... 12
II. Configuring SAML Extension ... 13
5. Overview ... 14
6. Integration to applications ... 15
6.1. Maven dependency ... 15
6.2. Bean definitions ... 15
6.3. Java-based configuration ... 15
6.4. Spring Security integration ... 15
6.5. Error handling ... 16
6.6. Logging ... 16
7. Metadata configuration ... 17
7.1. Service provider metadata ... 17
Automatic metadata generation ... 17
Pre-configured metadata ... 19
Downloading metadata ... 20
7.2. Identity provider metadata ... 21
File-based metadata provider ... 21
HTTP-based metadata provider ... 21
HTTP-based metadata provider with SSL ... 21

Metadata signature verification 22
7.3. Extended metadata 22
7.4. Multi-tenancy and entity alias 24
8. Security configuration 26
8.1. Key management 26
Sample JKS keystore 26
Generating and importing private keys 26
Importing public keys 27
Loading SSL/TLS certificates 27
8.2. Security profiles 27
Metadata interoperability profile (MetaIOP) 28
PKIX profile 28
Custom profile 29
8.3. Hostname verification for HTTPS connections 29
9. Single sign-on configuration 30
9.1. IDP selection and discovery 30
9.2. Single sign-on process 31
Service provider initialized SSO 31
Identity provider initialized SSO 32
9.3. Logout process 33
Local logout 33
Global logout 33
9.4. Authentication object 34
9.5. Authentication assertion 35
9.6. Authentication log 35
10. Advanced configuration 37
10.1. Reverse proxies and load balancers 37
10.2. Context provider 37
10.3. Validity intervals 38
10.4. Enhanced client/proxy 40
10.5. Endpoint URLs 41
10.6. Artifact resolution 41
III. Sample application 43
11. Sample application 44
11.1. SAML login 44
11.2. Metadata administration 44
11.3. Metadata generation 44
IV. Integration guide 45
12. Integrating Identity Providers 46
12.1. Active Directory Federation Services 2.0 (AD FS) 46
Initialize IDP metadata 46
Initialize SP metadata 46
Test SSO 47
12.2. Okta 47
Deploy Spring SAML sample application 47
Configure Okta 47
Import Okta metadata to Spring SAML 48
Test SSO 49
13. Troubleshooting common problems 50

Part I. Getting Started

This chapter provides essential information needed to enable your application to act as a service provider and interact with identity providers using SAML 2.0 protocol. Later in this guide you can find information about detailed configuration options and additional use-cases enabled by this component.

1. Introduction

1.1 What this manual covers

This manual describes Spring Security SAML Extension component, its uses, installation, configuration, design and integartion possibilities.

1.2 When to use Spring Security SAML Extension

The extension enables both new and existing applications to act as a Service Provider in federations based on Web Single Sign-On and Single Logout profiles of SAML 2.0 protocol. The extension allows seamless combination of SAML 2.0 and other authentication and federation mechanisms in a single application. All products supporting SAML 2.0 in Identity Provider mode (e.g. ADFS, Okta, Shibboleth, OpenAM, Efecte EIM or Ping Federate) can be used with the extension.

The extension can also be used in applications which are not primarily secured using Spring Security. It can be adapted for both single and multi-tenant environments.

The extension can be either embedded inside your application and work along other authentication or single sign-on mechanisms, or it can be deployed separately and convey authentication information to applications using a custom mechanism.

The extension is probably the most complete open-source SAML 2.0 SP implementation with the widest feature-set and configuration possibilities. Other Java open-source alternatives are e.g. native SAML service providers integrating with IIS or Apache from Shibboleth (SAML processing is done on the web server and not on the application level) or OpenAM Fedlet.

1.3 Features and supported profiles

Current implementation should be conformant to SAML SP Lite and SAML eGovernment profile. The following profiles, bindings and features are supported as part of the product:

- Web single sign-on profile
- Web single sign-on holder-of-key profile
- IDP and SP initialized single sign-on
- Single logout profile
- Enhanced client/proxy profile
- Identity provider discovery profile and IDP selection
- Metadata interoperability and PKIX trust management
- Automatic service provider metadata generation
- Metadata loading from files, URLs, file-backed URLs
- Processing and automatic reloading of metadata with many identity providers
- Support for authentication contexts
- Logging for authentication events
- Customization of both SP and IDP metadata
- Processing of SAML attributes and user data using UserDetails interface
- Support for HTTP-POST, HTTP-Redirect, SOAP, PAOS and Artifact bindings
- Easy integration with applications using Spring Security
- Sample application with an user interface for quick configuration

You can use the following supported standards as a reference:

SAML 2.0 basic profiles

- http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf
- http://docs.oasis-open.org/security/saml/v2.0/saml-metadata-2.0-os.pdf
- http://docs.oasis-open.org/security/saml/v2.0/saml-profiles-2.0-os.pdf
- http://docs.oasis-open.org/security/saml/v2.0/saml-authn-context-2.0-os.pdf
- http://docs.oasis-open.org/security/saml/v2.0/saml-bindings-2.0-os.pdf
- http://docs.oasis-open.org/security/saml/v2.0/saml-conformance-2.0-os.pdf

SAML 2.0 additional profiles

- http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-holder-of-key-browser-sso.pdf
- http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-idp-discovery.pdf
- http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml2-holder-of-key.pdf
- http://docs.oasis-open.org/security/saml/Post2.0/sstc-metadata-iop.pdf

eGovernment profile

- http://kantarainitiative.org/confluence/download/attachments/42139782/kantara-egov-saml2-profile-2.0.pdf

1.4 Requirements

Spring Security SAML Extension requires as a minimum Java 1.6 and is known to work with most Java containers and application servers. It can also be used with PaaS providers, such as Google App Engine, please see https://github.com/vschafer/spring-security-saml-gae for details.

1.5 Source code

Source code for the project is maintained on Github.

1.6 Builds

Snapshot builds of the project are available in the SpringSource repository. We use Bamboo for continuous integration.

1.7 License

Source code of the module is licensed under the Apache License, Version 2.0. You may obtain copy of the license at http://www.apache.org/licenses/LICENSE-2.0.

1.8 Issue tracking

Please use Spring Security Extensions Jira for submitting of bugs and feature requests. Patches can be sent directly to GitHub as pull requests, but preferably open a Jira issue as well.

1.9 Contributions

Please send your pull requests directly to GitHub and preferably also open issue in Jira.

1.10 Commercial support

For commercial support and consulting services please contact sales@v7security.com

1.11 Community support

For community support please use Stack Overflow. The Spring Security forums contain some previously answered questions, but are now in read-only mode.

1.12 Dependencies

Internal processing of SAML messages, marshalling and unmarshalling is handled by OpenSAML.

Spring SAML has a transitive dependency to library not-yet-commons-ssl. Inside Spring SAML this library is only used for hostname verifications and will be removed in case OpenSAML removes the dependency.

2. What's new

This section contains overview of important changes for released versions of Spring SAML.

2.1 New features, improvements and fixes in 1.0.1.FINAL

Version 1.0.1.FINAL is fully backwards compatible with 1.0.0.FINAL and contains the following changes:

- Added support for Spring Security 4.0
- Added integration guide with Okta
- MaxAuthenticationAge time supports longer expiration times than 21 days
- Deployment without JKS keystore is now supported
- Service provider can now define multiple assertion consumer endpoints with same binding
- Minor fixes and documentation improvements

2.2 New features, improvements and fixes in 1.0.0.FINAL

Final release is not directly compatible with the previous RC versions, please make sure to migrate your code based on guidlines and changes below:

- Metadata signing now supports custom keyInfoGenerator and signingAlgorithm, signing can be enable per-entity
- SAMLContextProvider has new customization possibilities for PKIXTrustEvaluator, PKIXInformationResolver and MetadataResolver
- CertPathPKIXTrustEvaluator supports customization of security provider and explicit validation of certification path
- MetadataCredentialResolver can be configured to load data from XML metadata and/or ExtendedMetadata
- PKIXInformationResolver has an extension point for population of CRLs
- Improvements to logging and error handling, profile implementations now throw exceptions which are logged inside filter objects and fail with ServletExceptions, sample application newly shows handling of these errors
- Used OpenSAML version was updated to 2.6.1
- SAMLDefaultLogger now logs additional information such as NameID
- Enabled propagation of defaults (e.g. ProxySettings) set in the HttpClient object for ArtifactResolution
- JKSKeyManager now supports keystores without password
- SAMLContextProviderLB now supports empty contextPath and includes pathInfo data for requests
- Entity ID and EntityDescriptor ID can now be set separately in MetadataGenerator
- ECP now takes precedence over discovery in SAMLEntryPoint
- Signing of local metadata is now done before displaying, this enables manual modifications to metadata in local files
- ArtifactResolutionProfileImpl now support customization of used SocketFactory through extensions
- ID in generated metadata is now automatically created when null, ID is based on entityID cleaned in order to conform to xsd:ID (and xsd:NCName) type, EntityID is cleaned by replacing all illegal characters by underscores
- Support for hostname verification in artifact resolution
- Completed documentation
- Possibility to exclude the SAML Credential from the Authentication object
- Disabled deferred node expansion for ParserPool which improves performance in parsing of small XML documents

- HttpSessionStorage is now cleared after successful reception of a message in order to save memory
- Possibility to include attributes from only the authenticated Assertion, or from all
- New socket factory for trust verification during loading of metadata from HTTPS
- Possibility to disable support for IDP-initialized SSO
- Usage of metadata alias is now optional
- New look and feel of the sample application
- Cleanup of duplicate values in MetadataGenerator and ExtendedMetadata
- SAMLCredential now contains facility methods for handling of String SAML attributes

2.3 Important code changes in 1.0.0.FINAL

Below is an overview of major code and structure changes since Spring SAML 1.0 RC2 with possible effect on backwards compatibility.

Module names

- module saml2-core was renamed to core, jar and maven artifact names stay the same
- module saml2-sample was renamed to sample, jar and maven artifact names stay the same
- module src was renamed to docs, jar and mave artifact names stay the same

Descriptor securityContext.xml

- file saml2-sample/src/main/resources/security/securityContext.xml was moved to sample/src/main/webapp/WEB-INF/securityContext.xml
- administration part of the UI is now secured with username/password
- updated initialization of ParserPool to disable defer node expansion
- HttpClient in ArtifactResolution was made thread safe
- added new failure handler (failureRedirectHandler)
- MetadataGenerator bean now demonstrates usage of ExtendedMetadata
- FilesystemMetadataProvider was replaced with ResourceBackedMetadataProvider
- file sample/src/main/resources/security/idp.xml was moved to sample/src/main/resources/metadata/idp.xml

ArtifactResolutionProfileBase

- throws SAMLException instead of CredentialExpiredException on check of artifact response issue instant

HttpSessionStorage

- storage is now cleared on successful message reception

MetadataDisplayFilter

- new mandatory property KeyManager (autowired)

MetadataGenerator

- generated metadata is no longer signed by default (enable in ExtendedMetadata.signMetadata) and has disabled IDP discovery (enable in ExtendedMetadata.includeDiscovery)
- the following fields were moved from MetadataGenerator to ExtendedMetadata:
  - entityAlias -> alias
  - signMetadata -> signMetadata
  - signingKey -> signingKey
  - encryptionKey -> encryptionKey
  - tlsKey -> tlsKey
  - includeDiscovery -> idpDiscoveryEnabled

  - customDiscoveryURL -> idpDiscoveryURL
  - customDiscoveryResponseURL -> idpDiscoveryResponseURL
- removed methods signSAMLObject (moved to SAMLUtil) and getKeyInfoGeneratorName (moved to ExtendedMetadata)
- by default the first binding is now HTTP-POST instead of HTTP-Artifact, endpoint for Web SSO no longer includes PAOS binding, set property *bindingsSSO* with values "artifact", "post", "paos" for backwards compatibility
- by default endpoints for Web SSO holder of key are no longer included, set property *bindingsHoKSSO* with values "artifact" and "post" for backwards compatibility
- by default *MetadataGeneratorFilter* no longer sets property *entityAlias* to value *defaultAlias*, set the value manually for backwards compatibility

SAMLAuthenticationProvider

- property forcePrincipalAsString is now set to true by default

SAMLCredential

- method getAttributeByName was renamed to getAttribute

SAMLDiscovery

- fails with ServletException instead of SAMLRuntimeException

SAMLLogoutProcessingFilter

- throws ServletException on errors during acceptance of LogoutRequest instead of SAMLRuntimeException

SAMLUtil

- removed unused getDefaultBinding method

SingleLogoutProfileImpl

- sendLogoutResponse signature changed
- changed error handling, throws SAMLStatusException which is handled by Filter, logged and sends a SAML Response

WebSSOProfileImpl

- throws SAMLException instead of SAMLRuntimeException on missing data in context

WebSSOProfileConsumerImpl

- new property includeAllAttributes, set to true for original behavior
- throws SAMLException instead of CredentialExpiredException on check of resposne issue instant and assertion issue instant

3. Glossary

Table 3.1. Definitions of terms used within this manual

Term	Definition
Assertion	A part of SAML message (an XML document) which provides facts about subject of the assertion (typically about the authenticated user). Assertions can contain information about authentication, associated attributes or authorization decisions.
Artifact	Identifier which can be used to retrieve a complete SAML message from identity or service provider using a back-channel binding.
Binding	Mechanism used to deliver SAML message. Bindings are divided to front-channel bindings which use web-browser of the user for message delivery (e.g. HTTP-POST or HTTP-Redirect) and back-channel bindings where identity provider and service provider communicate directly (e.g. using SOAP calls in Artifact binding).
Discovery	Mechanism used to determine which identity provider should be used to authenticate user currently interacting with the service provider.
Metadata	Document describing one or multiple identity and service providers. Metadata typically includes entity identifier, public keys, endpoint URLs, supported bindings and profiles, and other capabilities or requirements. Exchange of metadata between identity and service providers is typically the first step for establishment of federation.
Profile	Standardized combination of protocols, assertions, bindings and processing instructions used to achieve a particular use-case such as single sign-on, single logout, discovery, artifact resolution.
Protocol	Definition of format (schema) for SAML messages used to achieve particular functionality such as requesting authentication from IDP, performing single logout or requesting attributes from IDP.
Identity provider (IDP)	Entity which knows how to authenticate users and provides information about their identity to service providers/relaying parties using federation protocols.
Service provider (SP)	Your application which communicates with the identity provider in order to obtain information about the user it interacts with. User information such as authentication state and user attributes is provided in form of security assertions.
Single Sign-On (SSO)	Process enabling access to multiple web sites without need to repeatedly present credentials necessary for authentication. Various federation protocols such as SAML, WS-Federation, OpenID or OAuth can be used to achieve SSO use-cases. Information such as means of authentication, user attributes, authorization decisions or security tokens are typically provided to the service provider as part of single sign-on.

Term	Definition
Single Logout (SLO)	Process terminating authenticated sessions at all resources which were accessed using single sign-on. Techniques such as redirecting user to each of the SSO participants or sending a logout SOAP messages are typically used.

4. Quick start guide

This chapter will guide you through steps required to easily integrate Spring Security SAML Extension with ssocircle.com's IDP service using SAML 2.0 protocol. When done you will have a working example of Web SSO against a single Identity Provider. The steps will guide you through deployment of the sample application, configuration of IDP metadata (XML document describing how to connect to the IDP server using SAML 2.0 protocol) and SP metadata (XML document describing your own service) and testing of web single sign-on and single logout.

Public demo of the sample application is available at saml-federation.appspot.com

4.1 Pre-requisites

Please make sure the following items are available before starting the installation:

- Java 1.6+ SDK
- Apache Maven

SAML Extension relies on XML processing capabilities of JAXP. Some older versions of JRE might require updating of the embedded JAXP libraries. In case you encounter XML processing exceptions please create folder *jdk/jre/lib/endorsed* in your JDK installation and include files in *lib/endorsed* from the latest OpenSAML archive available at http://shibboleth.net/downloads/java-opensaml/. The location of the endorsed folder may differ based on your application server or container.

Due to US export limitations Java JDK comes with a limited set of cryptography capabilities. Usage of the SAML Extension might require installation of the Unlimited Strength Jurisdiction Policy Files which removes these limitations.

4.2 Installation steps

Downloading sample application

Download the Spring SAML Extension either from sources or from one of the releases.

The Spring SAML Sample application is included in *sample* directory. We will be customizing content of the sample application in the following steps.

Configuration of IDP metadata

Modify file *sample/src/main/webapp/WEB-INF/securityContext.xml* of the sample application and replace *metadata* bean as follows:

```
<bean id="metadata" class="org.springframework.security.saml.metadata.CachingMetadataManager">
 <constructor-arg>
  <list>
   <bean class="org.opensaml.saml2.metadata.provider.HTTPMetadataProvider">
    <constructor-arg>
     <value type="java.lang.String">http://idp.ssocircle.com/idp-meta.xml</value>
    </constructor-arg>
    <constructor-arg>
     <value type="int">5000</value>
    </constructor-arg>
    <property name="parserPool" ref="parserPool"/>
   </bean>
  </list>
 </constructor-arg>
</bean>
```

The settings tell system to download IDP metadata from the given URL with timeout of 5 seconds. In production system metadata should be either stored as a local file or be downloaded from a source using SSL/TLS with configured trust or which provides digitally signed metadata.

Generation of SP metadata

Modify file *sample/src/main/webapp/WEB-INF/securityContext.xml* of the sample application, replace *metadataGeneratorFilter* bean as follows and make sure to replace the entityId value with a string which is unique within the SSO Circle service (e.g. urn:test:yourname:yourcity):

```
<bean id="metadataGeneratorFilter"
 class="org.springframework.security.saml.metadata.MetadataGeneratorFilter">
 <constructor-arg>
  <bean class="org.springframework.security.saml.metadata.MetadataGenerator">
   <property name="entityId" value="replaceWithUniqueIdentifier"/>
   <property name="extendedMetadata">
    <bean class="org.springframework.security.saml.metadata.ExtendedMetadata">
     <property name="signMetadata" value="false"/>
     <property name="idpDiscoveryEnabled" value="true"/>
    </bean>
   </property>
  </bean>
 </constructor-arg>
</bean>
```

Compilation

When building from sources compile whole project and install artifacts into your local Maven repository using:

```
gradlew build install
```

When using the release zip compile the sample application available in the *sample* directory using:

```
mvn package
```

You can find the compiled war archive *spring-security-saml2-sample.war* in directory *sample/build/libs/* (gradle) or *sample/target/* (maven).

Project files for your IDE can be created with *gradlew eclipse* or *gradlew idea*.

Deployment

You can start the application from the release *sample* directory using command:

```
mvn tomcat7:run
```

Same can be achieved using gradle with:

```
gradlew tomcatRun
```

After startup the Spring SAML sample application will be available at *http://localhost:8080/spring-security-saml2-sample*

Alternatively you can deploy the war archive to your application server or container.

Uploading of SP metadata to the IDP

Download current SP metadata:

- Open web browser to the URL of the deployed application.
- Select *Metadata information*.
- Select first item from category *Service providers*, e.g. *http://localhost:8080/spring-security-saml2-sample/saml/metadata*
- Copy content of the Metadata textarea to your clipboard.

Upload SP metadata to the IDP:

- Register yourself at www.ssocircle.com and login to the service.
- Select Metadata manager and click Add new Service Provider.
- Enter *entityId* configured in the section called “Generation of SP metadata” in the FQDN field.
- Paste content of clipboard into the metadata information textarea.
- Store metadata by pressing the Submit button.
- Logout from the SSOCircle service.

4.3 Testing single sign-on and single logout

Open the front page of your SP application, select *http://idp.ssocircle.com* IDP and press login. The system will generate a new authentication request using SAML 2.0 protocol, digitally sign it and send it to the IDP. After authentication at IDP with your account you will be redirected back to your application and automatically signed-in.

Pressing local logout will destroy local session and logout the user. While a session is still active at the IDP an attempt to reauthenticate will proceed without need to enter credentials.

Pressing global logout will destroy both local session and the session at IDP.

You can test IDP initialized single sign-on with URL *https://idp.ssocircle.com:443/sso/saml2/jsp/idpSSOInit.jsp?metaAlias=/ssocircle&spEntityID=replaceWithUniqueIdentifier*, after replacing the service provider identifier with the one configured as entityId in your securityContext.xml. It is possible to provide relayState data sent to your SP with parameter *RelayState*.

Part II. Configuring SAML Extension

This chapter provides information about configuration and customization options of the SAML extension. It will guide you through typical scenarios including problems you might encounter during integration with identity providers.

5. Overview

Spring Security SAML 2.0 library comprises three modules:

- *core* contains implementation of the WebSSO profiles of the SAML 2.0 protocol and is required for integration to target systems.
- *sample* contains example of Spring configuration used for integration to target systems. It also contains user interface for generation and management of metadata.
- *docs* contains this documentation.

Configuration of the library is done using Spring context XML. An example of configuration can be found under *sample/src/main/webapp/WEB-INF/securityContext.xml*. Setting up of the library typically involves these steps:

- integration to application using Spring XML configuration
- import, generation and customization of SP and IDP metadata
- configuration of signature, encryption and trust keys
- configuration of security profiles
- configuration of reverse proxy or load balancer
- configuration of IDP selection or discovery
- configuration of single sign-on process
- configuration of logout process
- configuration of authentication object
- configuration of authentication log

Additional steps such as customization of SAML 2.0 bindings, configuration of artifact resolution or configuration of time skews might be needed.

6. Integration to applications

SAML module can be directly embedded into new or existing Spring applications. In this case application itself includes the SAML library in WEB-INF/lib directory of the war archive and processes all SAML interactions. The other option of using the SAML library is deploying it as a stand-alone module which transfers information about the authenticated user to the target application using a custom mechanism. This chapter only discusses the first option.

6.1 Maven dependency

In order to include the library and all its dependencies add the following dependency to your pom.xml file:

```
<dependency>
 <groupId>org.springframework.security.extensions</groupId>
 <artifactId>spring-security-saml2-core</artifactId>
 <version>${version}</version>
</dependency>
```

The current version of SAML Extension has been tested to work with Spring 3.1.2, Spring Security 3.1.2 and OpenSAML 2.6.1. Later versions of these libraries are likely to be compatible without need for modifications.

6.2 Bean definitions

Configuration of the SAML library requires beans definitions included in the *sample/src/main/webapp/WEB-INF/securityContext.xml* configuration file. Include copy of the file in your own Spring application, either directly or with an inclusion. Configuration steps in the following chapters will be customizing beans included in the default context.

Beans of the SAML library are using auto-wiring and annotation-based configuration by default. Make sure that your Spring configuration contains e.g. the following settings in order to enable support for these features:

```
<context:annotation-config/>
<context:component-scan base-package="org.springframework.security.saml"/>
```

6.3 Java-based configuration

Spring SAML will include configuration classes for Spring Java-based configuration in future versions.

For an example of *securityContext.xml* translated into Java configuration in a Spring Boot application see project by Vincenzo De Notaris at https://github.com/vdenotaris/spring-boot-security-saml-sample.

6.4 Spring Security integration

Filters of the SAML module need to be enabled as part of the Spring Security settings. In case SAML authentication should be the default authentication mechanism of the application set bean *samlEntryPoint* as the default entry point. Make sure that filter *samlFilter* is included as one of the custom filters. In case SP metadata should be generated automatically during first request to the application include also filter *metadataGeneratorFilter*. The configuration directive may for example look as follows:

```
<security:http entry-point-ref="samlEntryPoint">
 <security:custom-filter before="FIRST" ref="metadataGeneratorFilter"/>
 <security:custom-filter after="BASIC_AUTH_FILTER" ref="samlFilter"/>
</security:http>
```

6.5 Error handling

Critical errors raised during processing of SAML messages are generally propagated as ServletExceptions to the Java container. In order to configure a custom error handling update your web.xml and provide a general handler for ServletExceptions:

```
<error-page>
 <exception-type>javax.servlet.ServletException</exception-type>
 <location>/error.jsp</location>
</error-page>
```

ServletException contains original reason for the failure as a cause. It is recommended that content of the exceptions is not displayed to end users, both for security and user experience reasons.

Errors produced during processing of the SAML AuthenticationResponse can be handled by plugging a custom implementation of the *org.springframework.security.web.authentication.AuthenticationFailureHandler* interface to the *samlWebSSOProcessingFilter* bean.

6.6 Logging

SAML Extension uses SLF4J framework for logging. The same applies to the underlaying OpenSAML library. The sample application by default uses log4j version 1.2 binding for SLF4J, configured with the following dependency:

```
<dependency>
 <groupId>org.slf4j</groupId>
 <artifactId>slf4j-log4j12</artifactId>
 <version>1.6.3</version>
 <scope>compile</scope>
</dependency>
```

In case you are using another logging library, make sure to change the dependency accordingly.

You can enable debug logging by modifying file *sample/src/main/resources/log4j.properties* and adding:

```
log4j.logger.org.springframework.security.saml=DEBUG
log4j.logger.org.opensaml=DEBUG
log4j.logger.PROTOCOL_MESSAGE=DEBUG
```

For details about using other logging frameworks please consult the SLF4J manual.

7. Metadata configuration

SAML metadata is an XML document which contains information necessary for interaction with SAML-enabled identity or service providers. The document contains e.g. URLs of endpoints, information about supported bindings, identifiers and public keys. Typically one metadata document will be generated for your own service provider and sent to all identity providers you want to enable single sign-on with. Similarly, each identity provider will make its own metadata available for you to import into your service provider application.

Each metadata document can contain definition for one or many identity or service providers and optionally can be digitally signed. Metadata can be customized either by direct modifications to the XML document, or using extended metadata. Extended metadata is added directly to the Spring configuration file and can contain additional options which are unavailable in the basic metadata document.

7.1 Service provider metadata

Service provider metadata contains keys, services and URLs defining SAML endpoints of your application. Metadata can be either generated automatically upon first request to the service, or it can be pre-created (see Chapter 11, *Sample application*). Once created metadata needs to be provided to the identity providers with whom we want to establish trust.

Automatic metadata generation

Automatic metadata generation is enabled by including the following filter in the Spring Security configuration:

```
<security:custom-filter before="FIRST" ref="metadataGeneratorFilter"/>
```

This filter is automatically invoked as part of the first request to a URL processed by Spring Security. In case there is no service provider metadata already specified (meaning property *hostedSPName* of the *metadata* bean is empty) filter will generate a new one.

By default metadata will be generated with the following values which can be customized by setting properties of the *metadataGenerator* bean:

Table 7.1. Metadata generator settings

Property	Description	Default value
`entityBaseURL`	Base URL to construct SAML endpoints from, needs to be a URL with protocol, server, port and context path.	Values from the request in format: *scheme://server:port/contextPath*
`entityId`	Unique identifier of the service provider.	<entityBaseUrl>/saml/metadata
`id`	XML identifier of the root metadata element referred in signature.	entityId with removed illegal characters (NCName)
`requestSigned`	Flag indicating whether this service signs authentication requests.	true
`wantAssertionSigned`	Flag indicating whether this service requires signed assertions.	true

Property	Description	Default value
`bindingsSSO`	Bindings to be included in the metadata for WebSSO profile. Supported values are: POST, Artifact and PAOS. Order of bindings in the property determines order of endpoints in the generated metadata.	POST, Artifact
`bindingsHoKSSO`	Bindings to be included in the metadata for WebSSO Holder-of-Key profile. Supported values are: POST and Artifact. Order of bindings in the property determines order of endpoints in the generated metadata.	
`bindingsSLO`	Bindings to be included in the metadata for Single Logout profile. Supported values are: POST and Redirect. Order of bindings in the property determines order of endpoints in the generated metadata.	POST, Redirect
`assertionConsumerIndex`	Index of assertion consumer point to be marked as default.	0
`includeDiscoveryExtension`	When true generated metadata will contain extension indicating that it's able to consume response from an IDP Discovery service.	false
`nameID`	Name identifiers to be included in the metadata. Supported values are: EMAIL, TRANSIENT, PERSISTENT, UNSPECIFIED and X509_SUBJECT. Order of NameIDs in the property determines order of NameIDs in the generated metadata.	EMAIL, TRANSIENT, PERSISTENT, UNSPECIFIED, X509_SUBJECT
`extendedMetadata`	Additional settings such as security keys, entity alias, metadata signing, IDP discovery, ECP settings, security profiles and signature requirements can be specified in the ExtendedMetadata, see Section 7.3, “Extended metadata” for details.	no extended metadata

In case property `entityBaseURL` is not specified, it will be automatically generated based on values in the first HTTP request. Generated value can be normalized to exclude standard 80/443 ports for http/https schemes by setting property `normalizeBaseUrl` of the MetadataGeneratorFilter to `true`. It is recommended to provide the value explicitly in the configuration.

Providing an empty collection or null value to properties *bindingsSSO*, *bindingsHoKSSO* and *bindingsSLO* will disable and remove the given profile. For example the following setting removes the holder-of-key profile from the generated metadata, forces artifact binding for single sign-on and redirect binding for single logout:

```
<bean class="org.springframework.security.saml.metadata.MetadataGenerator">
 <property name="bindingsSSO"><list><value>artifact</value></list></property>
 <property name="bindingsSLO"><list><value>redirect</value></list></property>
 <property name="bindingsHoKSSO"><list/></property>
</bean>
```

By default generated metadata will not be digitally signed. Digital signature can be enabled using property `signMetadata` of the *extendedMetdata* bean.

In case application is deployed behind a reverse-proxy or other mechanism which makes the URL at the application server different from the URL seen by client at least property `entityBaseURL` should be set to a value e.g. https://www.server.com:8080 For details about load balancing see Section 10.1, “Reverse proxies and load balancers”.

Pre-configured metadata

In some situations it is beneficial to provide static version of the metadata document instead of the automatic generation. Need for manual changes in the metadata or fixing of production settings are some of those. A custom metadata document describing local SP application can be added by updating the *metadata* bean with correct ExtendedMetadata. Please follow these steps in order to do so:

- Generate and download metadata, e.g. using the *Metadata Administration -> Generate new service provider metadata* option in the sample application's administration UI or using instructions in automatic metadata generator.

- Store the metadata file as part of your project classpath, e.g. in *WEB-INF/classes/metadata/localhost_sp.xml*.

- Disable the automatic metadata generator by removing the following custom filter from the *securityContext.xml*:

  ```
  <security:custom-filter before="FIRST" ref="metadataGeneratorFilter"/>
  ```

- Include the SP metadata in the *metadata* bean and mark the entity as *local* in the extended metadata. Make sure to specify the *alias* property in case it was used during metadata generation.

  It is recommended to use the administration UI which also generates all the Spring declarations ready for inclusion in your *securityContext.xml*.

  Configuration for pre-configured local metdadata can look for example like this:

```
<bean class="org.springframework.security.saml.metadata.ExtendedMetadataDelegate">
 <constructor-arg>
  <bean class="org.opensaml.saml2.metadata.provider.ResourceBackedMetadataProvider">
   <constructor-arg>
    <bean class="java.util.Timer"/>
   </constructor-arg>
   <constructor-arg>
    <bean class="org.opensaml.util.resource.ClasspathResource">
     <constructor-arg value="/metadata/localhost_sp.xml"/>
    </bean>
   </constructor-arg>
   <property name="parserPool" ref="parserPool"/>
  </bean>
 </constructor-arg>
 <constructor-arg>
  <bean class="org.springframework.security.saml.metadata.ExtendedMetadata">
   <property name="local" value="true"/>
   <property name="securityProfile" value="metaiop"/>
   <property name="sslSecurityProfile" value="pkix"/>
   <property name="signMetadata" value="true"/>
   <property name="signingKey" value="apollo"/>
   <property name="encryptionKey" value="apollo"/>
   <property name="requireArtifactResolveSigned" value="false"/>
   <property name="requireLogoutRequestSigned" value="false"/>
   <property name="requireLogoutResponseSigned" value="false"/>
   <property name="idpDiscoveryEnabled" value="true"/>
   <property name="idpDiscoveryURL"
    value="https://www.server.com:8080/context/saml/discovery"/>
   <property name="idpDiscoveryResponseURL"
    value="https://www.server.com:8080/context/saml/login?disco=true"/>
  </bean>
 </constructor-arg>
</bean>
```

Same instance of your application can include multiple statically declared local service providers each differentiated by its own unique alias and entity ID, see Section 7.4, “Multi-tenancy and entity alias” for details. In case your application defines multiple local service providers, set property *hostedSPName* of the *metadata* bean to the entity ID of the default one.

The file with pre-configured metadata doesn't need to include digital signature. Metadata will be automatically signed during runtime when property *signMetadata* is set to *true*.

For details about available settings of the ExtendedMetadata see Section 7.3, “Extended metadata”.

Downloading metadata

Metadata describing the default local application can be downloaded from URL:

```
https://www.server.com:8080/context/saml/metadata
```

In case the application is configured to contain multiple service providers metadata for each can be loaded by adding the alias:

```
https://www.server.com:8080/context/saml/login/alias/defaultAlias
```

URL for metadata download can be disabled by removing filter *metadataDisplayFilter* from the *securityContext.xml*.

Metadata is also available in the sample application's administration UI under *Metadata information -> selected SP*.

7.2 Identity provider metadata

Metadata for identity providers is imported to the *metadataManager* in a similar way as pre-configured SP metadata. Metadata containing one or many identity providers can be added by providing an URL or a file. Processing of metadata and processing of SAML messages can be customized using properties of ExtendedMetadataDelegate and ExtendedMetadata.

File-based metadata provider

File-based provider loads metadata from a file available in the filesystem or classpath.

```
<bean class="org.springframework.security.saml.metadata.ExtendedMetadataDelegate">
 <constructor-arg>
  <bean class="org.opensaml.saml2.metadata.provider.FilesystemMetadataProvider">
   <constructor-arg>
    <value type="java.io.File">classpath:security/idp.xml</value>
   </constructor-arg>
   <property name="parserPool" ref="parserPool"/>
  </bean>
 </constructor-arg>
 <constructor-arg>
  <bean class="org.springframework.security.saml.metadata.ExtendedMetadata"/>
 </constructor-arg>
</bean>
```

Metadata is automatically refreshed in intervals specified by properties *minRefreshDelay* and *maxRefreshDelay* of the *MetadataProvider* bean.

HTTP-based metadata provider

HTTP-based provider loads metadata from an URL.

```
<bean class="org.springframework.security.saml.metadata.ExtendedMetadataDelegate">
 <constructor-arg>
  <bean class="org.opensaml.saml2.metadata.provider.HTTPMetadataProvider">
   <constructor-arg>
    <value type="java.lang.String">http://idp.ssocircle.com/idp-meta.xml</value>
   </constructor-arg>
   <constructor-arg>
    <!-- Timeout for metadata loading in ms -->
    <value type="int">5000</value>
   </constructor-arg>
   <property name="parserPool" ref="parserPool"/>
  </bean>
 </constructor-arg>
 <constructor-arg>
  <bean class="org.springframework.security.saml.metadata.ExtendedMetadata"/>
 </constructor-arg>
</bean>
```

Metadata is automatically refreshed in intervals specified by properties *minRefreshDelay* and *maxRefreshDelay* of the *MetadataProvider* bean.

Alternatively class *org.opensaml.saml2.metadata.provider.FileBackedHTTPMetadataProvider* can be used to provide a backup in case URL is temporarily unavailable. File to use as backup is specified as third argument in the *MetadataProvider* bean constructor.

HTTP-based metadata provider with SSL

By default, loading of metadata using the HTTP-based provider over HTTPS performs trust verification configured in your JDK. In case you'd like to use certificates in your *keyStore*, add the following bean which changes the socketFactory used by the HTTP Client:

```
<bean class="org.springframework.security.saml.trust.httpclient.TLSProtocolConfigurer"/>
```

The *TLSProtocolConfigurer* instantiates *TLSProtocolSocketFactory* and registers is as a default socket factory for https protocol inside the HTTP Client used for metadata loading. The socket factory uses all public certificates present in the *keyStore* as trust anchors for PKIX validation. The used keys can be constrained with property *trustedKeys*.

The socket factory configured in this fashion is used for all metadata providers. It is possible to customize metadata loading on a per-provider basis by adding a configured HttpClient instance to the HTTPMetadataProvider constructor.

Metadata signature verification

Importing of digitally signed metadata requires verification of signature's validity and trust. Metadata is not required to be signed by default. When present, signature is verified with PKIX algorithm and uses all public keys present in the configured *keyManager* as trust anchors. Make sure to include root CA certificate and intermediary CA certificates of the signature in your *keyStore*. For details see the section called “Importing public keys”.

You can limit certificates used to peform the verification by setting property *metadataTrustedKeys* of the ExtendedMetadataDelegate bean. The provided collection should contain aliases of keys to be used as trust anchors.

Signature verification can be disabled by setting property *metadataTrustCheck* to false in the ExtendedMetadataDelegate bean. Setting *metadataRequireSignature* to true will reject metadata unless it's digitally signed.

7.3 Extended metadata

Extended metadata provides additional settings for customization of SAML exchanges between SP and IDP which are not supported in the standard SAML 2.0 metadata documents. Examples of such settings are requirements for message signing, IDP discovery and security profiles.

Extended metadata is defined using *org.springframework.security.saml.metadata.ExtendedMetadata* beans embedded inside ExtendedMetadataDelegate for each SP or IDP metadata definition. In case a single metadata document contains multiple identity providers (in multiple EntityDescriptor elements), extended metadata can be set separately for each of them using a map with entity IDs as keys, e.g.:

```
<bean class="org.springframework.security.saml.metadata.ExtendedMetadataDelegate">
 <constructor-arg>
  metadata provider bean
 </constructor-arg>
 <constructor-arg>
  <!-- Default extended metadata for entities not specified in the map -->
  <bean class="org.springframework.security.saml.metadata.ExtendedMetadata"/>
 </constructor-arg>
 <constructor-arg>
  <!-- Extended metadata for specific IDPs -->
  <map>
   <entry key="http://idp.ssocircle.com">
    <bean class="org.springframework.security.saml.metadata.ExtendedMetadata"/>
   </entry>
  </map>
 </constructor-arg>
</bean>
```

The following table summarizes settings available in the extended metadata. The same class is used for both local service providers and remote identity providers; each value contains information about the entities it's valid for.

Table 7.2. Extended metadata settings

Property	**Default**	**Entities**	**Description**
local	false	local and remote	True for metadata of a local service provider. False for remote identity providers.
alias		local only	Unique alias used to identify the selected local service provider based on used URL. See Section 7.4, “Multi-tenancy and entity alias”.
signMetadata	false	local only	When true generated metadata will be signed using XML Signature using certificate with alias of `signingKey`.
idpDiscoveryEnabled	false	local only	When true system will initialize IDP discovery when no IDP is selected during SSO initialization. See Section 9.1, “IDP selection and discovery”.
idpDiscoveryURL	internal discovery URL	local only	URL of the IDP discovery service. Only used when discovery is enabled.
idpDiscoveryResponseURL	internal discovery URL	local only	URL expecting response from the IDP discovery service. Only used when discovery is enabled.
ecpEnabled	false	local only	Property enables support for the SAML 2.0 ECP profile. See Section 10.4, “Enhanced client/proxy”.
securityProfile	metaiop	local only	Security profile for verification of message signatures. See Section 8.2, “Security profiles”.
sslSecurityProfile	pkix	local only	Security profile for vericiation of SSL/TLS endpoint trust. See Section 8.2, “Security profiles”.
sslHostnameVerification	default	local only	Verification of hostnames for HTTPS calls (e.g. in Artifact resolution). Allowed values are *default*, *defaultAndLocalhost*, *strict* and *allowAll*. Value *allowAll* effectively disables hostname verification. All values are case-insensitive. For more details on the supported hostname verifications see Commons-SSL JavaDoc.
signingAlgorithm	-	local only	Algorithm used to create digital signature on the metadata object. Typical values are *http://*

Property	Default	Entities	Description
			www.w3.org/2000/09/xmldsig#rsa-sha1, *http://www.w3.org/2001/04/xmldsig-more#rsa-sha256* and *http://www.w3.org/2001/04/xmldsig-more#rsa-sha512*.
signingKey	-	local and remote	For local entities alias of private key used to create signatures. The default private key is used when no value is provided. For remote identity providers defines an additional public key used to verify signatures.
encryptionKey	-	local and remote	For local entities alias of private key used to encrypt data. The default private key is used when no value is provided. For remote identity providers defines an additional public key used to decrypt data.
tlsKey	-	local and remote	For local entities alias of private key used for SSL/TLS client authentication. No client authentication is used when value is not specified. For remote identity providers defines an additional public key used for trust resolution.
trustedKeys	-	remote	Keys included as trusted anchors during PKIX evaluation. All keys in the keyStore are used as trust anchors with null value. Keys are only used with PKIX security profile.
requireLogoutRequestSigned	true	local and remote	For local entities enables requirement of signed logout requests. For remote entities enables signing of requests sent to the IDP.
requireLogoutResponseSigned	false	local and remote	For local entities enables requirement of signed logout responses. For remote entities enables signing of responses sent to the IDP.
requireArtifactResolveSigned	true	remote only	Enables signing of artifact resolution requests sent to the remote identity providers.
supportUnsolicitedResponse	true	remote only	Enables support for Unsolicited Responses (IDP-Initialized SSO) sent from this remote entity.

For additional examples on setting up metadata and extended metadata see Section 7.1, “Service provider metadata” for local SP, and Section 7.2, “Identity provider metadata” for remote IDPs.

7.4 Multi-tenancy and entity alias

Spring SAML contains limited support for multi-tenancy. It is possible to define configuration for multiple instances of local service providers, where each can have different URLs and security settings. System is differentiating between the service provider instances using *entity alias* which is a unique identifier within deployment of Spring SAML.

Entity alias is appended to URLs of SAML endpoints and used by Spring SAML to identify the correct instance. For example for local service provider with *entity alias customer123* the standard URL *scheme://server:port/contextPath/saml/login* becomes *scheme://server:port/contextPath/saml/login/alias/customer123*.

The entity alias functionality can only be used together with pre-configured metadata (see the section called “Pre-configured metadata”). The *entity alias* is specified in the extended metadata of each of the configured service providers.

Spring SAML doesn't enforce any limitations on which Identity Provider can be deliver messages to which of the local Service Providers. In case your application requires similar rules (for example only certain tenants can authenticate using a specific IDP), make sure to implement them for example in your *SAMLUserDetailsService* (for single sign-on).

Selection of the correct Service Provider instance based on URL is performed inside *SAMLContextProviderImpl* class.

8. Security configuration

SAML Extension requires configuration of security settings which include cryptographic material used for digital signatures and encryption, security profiles for configuration of trusted cryptographic material provided by remote entities and verifications of HTTPS connections.

8.1 Key management

SAML exchanges involve usage of cryptography for signing and encryption of data. All interaction with cryptographic keys is done through interface *org.springframework.security.saml.key.KeyManager*. The default implementation *org.springframework.security.saml.key.JKSKeyManager* relies on a single JKS key store which contains all private and public keys. KeyManager should contain at least one private key which should be marked as default by using the alias of the private key as part of the *JKSKeyManager* constructor.

In case your application doesn't need to create digital signatures and/or decrypt incoming messages, it is possible to use an empty implementation of the keystore which doesn't require any JKS file - *org.springframework.security.saml.key.EmptyKeyManager*. This can be the case for example when using only IDP-Initialized single sign-on. Please note that when using the *EmptyKeyManager* some of Spring SAML features will be unavailable. This includes at least SP-initialized Single Sign-on, Single Logout, usage of additional keys in *ExtendedMetadata* and verification of metadata signatures. Use the following bean in order to initialize the *EmptyKeyManager*:

```
<bean id="keyManager" class="org.springframework.security.saml.key.EmptyKeyManager"/>
```

Sample JKS keystore

Sample application contains a default JKS key store with a sample private certificate usable for test purposes. The key store is defined as:

```
<bean id="keyManager" class="org.springframework.security.saml.key.JKSKeyManager">
 <constructor-arg value="classpath:security/samlKeystore.jks"/>
 <constructor-arg type="java.lang.String" value="nalle123"/>
 <constructor-arg>
 <map>
  <entry key="apollo" value="nalle123"/>
 </map>
 </constructor-arg>
 <constructor-arg type="java.lang.String" value="apollo"/>
</bean>
```

The first argument points to the used key store file, second contains password for the keystore, third then map with passwords for private keys with alias-password value pairs. Alias of the default certificate is the last parameter.

Generating and importing private keys

Private keys (with either self-signed or CA signed certificates) are used to digitally sign SAML messages, encrypt their content and in some cases for SSL/TLS Client authentication of your service provider application. SAML Extension ships with a default private key in the *samlKeystore.jks* with alias *apollo* which can be used for initial testing, but for security reason should be replaced with your own key in early development stages.

In case your IDP doesn't require keys signed by a specific certification authority you can generate your own self-signed key using the Java utility *keytool*, e.g. with:

```
keytool -genkeypair -alias some-alias -keypass changeit -keystore samlKeystore.jks
```

The keystore will now contain additional PrivateKeyEntry with alias *mykey* which can be imported to the *keyManager* in your *securityContext.xml*.

Keys signed by certification authorities are typically provided in .p12/.pfx format (or can be converted to such using OpenSSL) and imported to Java keystore with, e.g.:

```
keytool -importkeystore -srckeystore key.p12 -srcstoretype PKCS12 -srcstorepass password \
 -alias some-alias -destkeystore samlKeystore.jks -destalias some-alias \
 -destkeypass changeit
```

The following command can be used to determine available alias in the p12 file:

```
keytool -list -keystore key.p12 -storetype pkcs12
```

Importing public keys

Cryptographic material used to decrypt incoming data and verify trust of signatures in SAML messages and metadata is stored either in metadata of remote entities or in the *keyManager*. In order to import additional trusted key to the keystore run, e.g.:

```
keytool -importcert -alias some-alias -file key.cer -keystore samlKeystore.jks
```

Imported keys can be referenced in ExtendedMetadataDelegate and ExtendedMetadata beans, for details see the section called “Metadata signature verification” and Section 8.2, “Security profiles”.

Loading SSL/TLS certificates

Direct SSL/TLS connections (used with HTTP-Artifact binding) require verification of the public key presented by the server. The SSL Extractor utility can be used to extract certificates presented by an SSL/TLS endpoint, e.g. with:

```
java -jar sslextractor-0.9.jar www.google.com 443
```

The certificates are stored as .cer files and can be imported to the keystore as a usual public key. For details about configuring of trust for SSL/TLS connections see Section 8.2, “Security profiles”.

8.2 Security profiles

Exchanges of messages between identity providers and service providers with SAML protocol involves usage of digital signatures. Signatures are typically constructed using means of asymetric cryptography and public key infrastructure with public and private keys signed by trusted certification authorities. Signatures are either applied directly to parts of XML representation of SAML messages using XML Signature or are part of the transport layer used to deliver the message like SSL/TLS.

Verification of signatures is executed in two phases. Signature is first checked for validity by comparing digital hash included as part of the signature with value calculated from the content. Subsequently it is verified whether party who created the signature is trusted by the recipient. Spring Security SAML provides two mechanisms for defining which signatures should be accepted - metadata interoperability mode and PKIX mode.

Security profiles are defined in Extended Metadata of your local SP. Profile can be defined separately for XML Signatures using property *securityProfile* and for SSL/TLS Signatures using property*sslSecurityProfile*. Value of both properties can be either *metaiop* or *pkix*. For details about using Extended Metadata see Chapter 7, *Metadata configuration*, for reference of allowed values see Section 7.3, “Extended metadata”.

Metadata interoperability profile (MetaIOP)

With MetaIOP mode certificates are not checked for expiration or revocation and certificate paths are not verified. This means that it does not matter which certification authority issued the certificate, as the fact whether the certificate is trusted or not is conveyed using other mechanisms (e.g. by secure metadata exchange or digital signature of metadata itself).

Signature is deemed trusted when the certificate used to create it is included in one of the following places:

- Key with usage of signing or unspecified in entity metadata of a remote entity
- Signing key specified in property *signingKey* of extended metadata of a remote entity

MetaIOP is the default profile for verification of XML signatures. For details about this profile see the specification.

PKIX profile

With PKIX profile trust of signature certificates is verified based on a certificate path between trusted CA certificates and the certificate in question. Certificate is trusted when it's possible to construct path from a trusted certificate to the validated one. With this profile certificate expiration and revocation can be checked.

Trusted keys (anchors) for PKIX verification of signatures are combined from the following places:

- Key with usage of signing or unspecified in entity metadata of a remote entity
- Signing key specified in property *signingKey* of extended metadata of a remote entity
- All keys specified in *trustedKeys* set of extended metadata of a remote entity, or all keys available in the key store when the property is null (default value)

Please note that trust anchors are treated as automatically trusted and are not necessarily subject to all checks as leaf certificates are (depending on your security provider implementation). You should preferably use only your CA and intermediary CA certificates as trust anchors. In case you want to ignore certificates available in your XML metadata and only use settings from your manually set ExtendedMetadata, set property *useXmlMetadata* of your *metadataResolver* to false:

```
<bean id="contextProvider" class="org.springframework.security.saml.context.SAMLContextProviderImpl">
 <property name="metadataResolver">
  <bean class="org.springframework.security.saml.trust.MetadataCredentialResolver">
   <constructor-arg index="0" ref="metadata"/>
   <constructor-arg index="1" ref="keyManager"/>
   <property name="useXmlMetadata" value="false"/>
  </bean>
 </property>
</bean>
```

PKIX verification supports checking of CRLs (certificate revocation lists) using the default underlaying Java Security Provider (e.g. Sun JCE, BouncyCastle JCE).

The PKIX algorithm needs to be advised that the revocation checking is enabled. You can do so by customizing the *pkixTrustEvaluator* inside *SAMLContextProvider*, see an example with properties *forceRevocationEnabled* and *revocationEnabled* bellow.

By default the validation algorithm only uses the *CertPathBuilder*. Some Java security implementations do not support full feature set of revocation checking in this class and only implement them in the *CertPathValidator* (e.g. Sun provider only supports OCSP in CertPathBuilder since Java 1.8). You can instruct system to use both *CertPathBuilder* and *CertPathValidator* by setting property *validateCertPath* to *true* on bean *CertPathPKIXTrustEvaluator*.

The security provider used for loading of PKIX verification factories can be customized using property *securityProvider*.

```
<bean id="contextProvider" class="org.springframework.security.saml.context.SAMLContextProviderImpl">
 <property name="pkixTrustEvaluator">
  <bean class="org.springframework.security.saml.trust.CertPathPKIXTrustEvaluator">
   <property name="PKIXValidationOptions">
    <bean class="org.opensaml.xml.security.x509.CertPathPKIXValidationOptions">
     <property name="forceRevocationEnabled" value="true"/>
     <property name="revocationEnabled" value="true"/>
    </bean>
   </property>
   <property name="validateCertPath" value="true"/>
   <property name="securityProvider" value="SUN"/>
  </bean>
 </property>
</bean>
```

Spring SAML uses standard CertPath verification API. The default Sun JCE provider supports automatic revocation checking based on the certificate's CRL Distribution Points Extension (by setting system property *com.sun.security.enableCRLDP* to true), CRL point defined using certificate's Authority Information Access (AIA) Extension (by setting system property *com.sun.security.enableAIAcaIssuers* to true) and OCSP (by setting system property *ocsp.enable* to true). For details see the Java PKI Programmer's Guide. In case you are using another security provider, please consult its manual for functionality related to *CertPathBuilder* and *CertPathValidator* with the *PKIX* algorithm.

You can also manually populate CRLs by extending class *org.springframework.security.saml.trust.PKIXInformationResolver* and overriding method *populateCRLs* with your own CRL population logic. Populated CRLs are automatically added to the PKIX verification mechanism. The customized class needs to be set to property *pkixResolver* in the *contextProvider* bean.

Custom profile

Engine used to verify trust of signatures for given combination of SP/IDP is created in methods *populateTrustEngine* and *populateSSLTrustEngine* of interface *org.springframework.security.saml.context.SAMLContextProvider* and can be overriden with custom implementation. See Section 10.2, “Context provider” for details on context customization.

8.3 Hostname verification for HTTPS connections

Connections to HTTPS services (e.g. during Artifact resolution) require verification that the connected hostname corresponds with the hostname defined in the service's public certificate. Hostname verification is enabled by default.

Verification can be disabled by setting ExtendedMetadata property *sslHostnameVerification* of the local SP entity to *allowAll*. For details on using the ExtendedMetadata see Section 7.3, “Extended metadata”.

All supported values can be found in the ExtendedMetadata reference Section 7.3, “Extended metadata”.

9. Single sign-on configuration

9.1 IDP selection and discovery

Discovery helps your Service Provider determine which Identity Provider should be used for authentication of the current user. It is automatically initialized during calls to single sign-on endpoint at *scheme://server:port/contextPath/saml/login*. SAML Extension supports multiple modes of discovery including the Identity Provider Discovery Service Protocol and Profile.

IDP discovery modes can always be skipped during SSO initialization by specifying HTTP request parameter *idp* with the entityId of the required IDP, e.g. *scheme://server:port/contextPath/saml/login?idp=mySelectedIDP*.

The URL where local SP expects discovery response can be included in the SP metadata as one of the extensions. The feature can be enabled by setting property *includeDiscoveryExtension* to true on bean *MetadataGenerator* inside *MetadataGeneratorFilter*, e.g.:

```
<bean id="metadataGeneratorFilter"
 class="org.springframework.security.saml.metadata.MetadataGeneratorFilter">
 <constructor-arg>
  <bean class="org.springframework.security.saml.metadata.MetadataGenerator">
   <property name="includeDiscoveryExtension" value="true"/>
  </bean>
 </constructor-arg>
</bean>
```

Default IDP without discovery

The mode is enabled by default and automatically selects the default IDP without performing discovery.

The default IDP can be configured using property *defaultIDP* on bean *metadata* in the Spring Security configuration. In case the property isn't set, system will automatically use the first available IDP.

Local discovery service

SAML Extension includes a local IDP discovery service which presents user with an IDP selection page. This mode can be enabled by setting property *includeDiscovery* in the local SP extended metadata to *true*.

The selection page can be customized using property *idpSelectionPath* on bean *samlIDPDiscovery*. System forwards to this page wih a discovery request which includes the following request attributes:

- *idpDiscoReturnURL* - URL to send the IDP selection result to using GET action
- *idpDiscoReturnParam* - name of the GET parameter to include the entity ID of the selected IDP

See the default implementation in *sample/src/main/webapp/WEB-INF/security/idpSelection.jsp* for an example.

Remote discovery service

In order to enable external IDP discovery service configure property *idpDiscoveryURL* in your local SP extended metadata to the external discovery URL. Make sure property *idpDiscoveryEnabled* is set to true. The remote discovery service needs to support the Identity Provider Discovery Service Protocol and Profile.

9.2 Single sign-on process

Spring SAML Extension supports both SP-initialized and IDP-initialized single sign-on.

Service provider initialized SSO

SP initialized SSO process can be started in two ways:

- User accesses a resource protected by Spring Security which initializes SAMLEntryPoint
- User is redirected to the SSO endpoint at e.g. https://www.server.com/context/saml/login

After identification of IDP to use for authentication (for details see Section 9.1, “IDP selection and discovery”), SAML Extension creates an AuthnRequest SAML message and sends it to the selected IDP. Both construction of the AuthnRequest and binding used to send it can be customized using *WebSSOProfileOptions* object. SAMLEntryPoint determines *WebSSOProfileOptions* configuration to use by calling method *getProfileOptions*. The default implementation returns the value specified in property *defaultOptions*. The method can be overriden to provide custom logic for SSO initialization.

Default settings for *WebSSOProfileOptions* can be specified in bean *samlEntryPoint* of your *securityContext.xml*, e.g.:

```
<bean id="samlEntryPoint" class="org.springframework.security.saml.SAMLEntryPoint">
 <property name="defaultProfileOptions">
  <bean class="org.springframework.security.saml.websso.WebSSOProfileOptions">
   <property name="binding" value="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"/>
   <property name="includeScoping" value="false"/>
  </bean>
 </property>
</bean>
```

WebSSOProfileOptions supports the following settings:

Table 9.1. org.springframework.security.saml.websso.WebSSOProfileOptions parameters

Property	Description
binding	Default: binding of the first declared SingleSignOnService in IDP metadata. Binding used to send message to IDP. Supported values depend on the SP configuration, typically "urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST", "urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Redirect", "urn:oasis:names:tc:SAML:2.0:bindings:PAOS" and "urn:oasis:names:tc:SAML:2.0:profiles:holder-of-key:SSO:browser".
providerName	Default: empty. Human readable name of the local SP sent with the authentication request.
assertionConsumerIndex	Default: empty. When set determines where should IDP send response and which binding to use. Otherwise system uses the default assertion consumer service marked as default, or first applicable. Available indexes can be found in metadata of this service provider.

Property	Description
nameID	Default: empty. Name ID to request from IDP in the NameIDPolicy. No NameIDPolicy is sent when not specified. Typical values are "urn:oasis:names:tc:SAML:1.1:nameid-format:emailAddress", "urn:oasis:names:tc:SAML:2.0:nameid-format:transient", "urn:oasis:names:tc:SAML:2.0:nameid-format:persistent", "urn:oasis:names:tc:SAML:2.0:nameid-format:encrypted".
allowCreate	Default: empty. Only applicable when nameID is specified, when true instructs IDP that it is allowed to create new user based on the authentication request.
passive	Default: false. Sets whether the IdP should refrain from interacting with the user during the authentication process.
forceAuthn	Default: false. When true IDP is required to re-authenticate user and not rely on previous authentication events.
includeScoping	Default: true. When true request will include Scoping element.
allowedIDPs	Default: empty. Values to be included in the Scoping element on top of the IDP message is sent to. Only applicable when includeScoping is set to true.
proxyCount	Default: 2. Determines value to be used in the proxyCount attribute of the scope in the AuthnRequest. Use zero to disable proxying or value >0 to specify how many hops are allowed. Only applicable when includeScoping is set to true.
authnContexts	Default: empty. Authentication contexts IDP is allowed to use when authenticating user. See the specification for details.
authnContextComparison	Default: AuthnContextComparisonTypeEnumeration.EXACT. Mechanism used by IDP to determine authentication method to use. See the specification for details.
relayState	Default: empty. Value is sent to IDP and provided back to SP as part of the authentication response.

The AuthnRequest message is sent unencrypted on message level. If needed, encryption should be provided by SSL/TLS on transport layer.

Identity provider initialized SSO

Spring SAML supports reception of Unsolicited Response messages (so called IDP-initialized SSO). In this scenario IDP creates a Response object in the same way as if it was replying to an AuthnRequest message sent from SP, but it omits the InResponseTo parameter. Message is then sent to the AssertionConsumerURL of Spring SAML (typically *scheme://server:port/contextPath/saml/SSO*) using one of the supported bindings. List of all available endpoints and bindings can be found in the metadata of the Spring SAML application.

Received Unsolicited Respose message is processed and validated in exactly the same way as with SP-Initialized SSO.

Support for unsolicited messages can be disabled in the ExtendedMetadata of remote entities using property *supportUnsolicitedResponse*.

9.3 Logout process

Spring SAML Extension supports both Local Logout and Single Logout mechanisms.

Local logout

Local logout terminates only the local session and doesn't affect neither session at IDP, nor sessions at other SPs where user logged in using single sign-on. Local logout can be initialized at *scheme://server:port/contextPath/saml/logout?local=true*. Call is intercepted by bean *samlLogoutFilter* which can be configured with the following settings:

- Instance of interface *org.springframework.security.web.authentication.logout.LogoutSuccessHandler* (constructor index 0) which determines operation to perform after successful logout (e.g. redirect to a logout landing page). By default user gets redirected to page *logout.jsp*.

- Instances of interface *org.springframework.security.web.authentication.logout.LogoutHandler* (constructor index 1) which are responsible for destruction of user's session. The default handler *org.springframework.security.web.authentication.logout.SecurityContextLogoutHandler* logs the user out by removing the Authentication object, but leaves the HTTP session opened.

It is also possible to configure local logout using standard Spring Security element *<security:logout>* inside *<security:http>* block. For example:

```
<security:http>
 <security:logout logout-url="/j_logout" logout-success-url="/logout.jsp"/>
</security:http>
```

Global logout

Global logout implements the SAML 2.0 Single Logout profile which terminates both session at the current SP, the IDP session and sessions at other SPs connected to the same IDP session. Single Logout can be initialized from any of the participaing SPs or from the IDP.

Single Logout is currently supported with HTTP-Redirect and HTTP-POST bindings. SOAP binding is not available.

Global logout can be initialized at *scheme://server:port/contextPath/saml/logout*. System automatically determines which IDP to send the request to based on the currently authenticated user. Single logout can be configured using beans *samlLogoutFilter* and *samlLogoutProcessingFilter* with the following options:

- Bean *samlLogoutFilter* can be provided with instances of interface *org.springframework.security.web.authentication.logout.LogoutHandler* (constructor index 3). The handlers are called before sending SAML 2.0 LogoutRequest to the IDP when initializing Single Logout from the current SP.

- Bean *samlLogoutProcessingFilter* can be provided with instance of interface *org.springframework.security.web.authentication.logout.LogoutSuccessHandler* (constructor index 0). Handler is called after successful finalization of Single Logout process (reception of LogoutResponse from IDP) and determines operation to perform after logout (e.g. redirect to a logout landing page). By default user gets redirected to page *logout.jsp*.

- Bean *samlLogoutProcessingFilter* can be provided with instances of interface *org.springframework.security.web.authentication.logout.LogoutHandler* (constructor index 1). The handlers are called after successful reception of SAML 2.0 LogoutRequest or LogoutResponse from the IDP.

Spring SAML correctly handles SAML 2.0 LogoutRequest messages sent from the IDP and performs logout in case the message is valid. In case of invalid data (missing signature, invalid issuer, invalid issue time, invalid destination, invalid session index, invalid name ID, no user logged in) system responds with SAML 2.0 LogoutResponse with an error Status code.

9.4 Authentication object

Successful authentication using SAML token results in creation of an *Authentication* object by the *SAMLAuthenticationProvider*. By default instance of *org.springframework.security.providers.ExpiringUsernameAuthenticationToken* is created. Content of the resulting object can be customized by setting properties of the *samlAuthenticationProvider* bean in the *securityContext.xml*. An instance of *org.springframework.security.saml.userdetails.SAMLUserDetailsService* can be provided to supply application-specific information about the authenticated user.

The *Authentication* object will by default include string version of the *NameID* included in the SAML Assertion as its*principal*. Property *forcePrincipalAsString* can be used to change this to include the raw *NameID* element.

The Authentication object is available in pages secured with Spring Security using *SecurityContextHolder.getContext().getAuthentication()* and is populated with the following values:

Table 9.2. ExpiringUsernameAuthenticationToken values.

Property	**Value**
Principal	When forcePrincipalAsString = true (default) - *String* value of *NameID* included in the SAML Assertion (*credential.getNameID().getValue()* of type java.lang.String)
Principal	When forcePrincipalAsString = false AND userDetail = null (default) - *NameID* object included in the SAML Assertion (*credential.getNameID()* of type *org.opensaml.saml2.core.NameID*)
Principal	When forcePrincipalAsString = false AND userDetail != null - *UserDetail* object returned from the *SAMLUserDetailsService*
Credentials	SAML authentication object including entity ID of local and remote entity, name ID, assertion and relay state (*org.springframework.security.saml.SAMLCredential*)
Authorities	Result of *getAuthorities()* call on the *UserDetails* object returned from *SAMLUserDetailsService*, empty list when there's no *UserDetail* object available.
Expiration	Value of *SessionNotOnOrAfter* in the SAML Assertion when avaialble, null otherwise. *Authentication* object will start returning false on the *isAuthenticated()* after the expiration time.

Custom implementation of the *SAMLUserDetailsService* can be provided as property *userDetails* of the *SAMLAuthenticationProvider*. Implementation can perform operation such as parsing of attributes present in the SAML Assertion, e.g.:

```
package fi.schafer.test.saml;

import org.opensaml.saml2.core.Attribute;
import org.opensaml.xml.XMLObject;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.security.saml.SAMLCredential;
import org.springframework.security.saml.userdetails.SAMLUserDetailsService;

public class TestUserDetails implements SAMLUserDetailsService {
 @Override
 public Object loadUserBySAML(SAMLCredential cred) throws UsernameNotFoundException {
  return cred.getAttributeAsString("accountID");
 }
}
```

Population of the authentication object can be further customized by overriding of the *getUserDetails*, *getPrincipal*, *getEntitlements* and *getExpirationDate* methods in the *SAMLAuthenticationProvider*.

9.5 Authentication assertion

Assertion used to authenticate user is stored in the *SAMLCredential* object under property *authenticationAssertion*. By default the original content (DOM) of the assertion is discarded and system only keeps an unmarshalled version which might slightly differ from the original, e.g. in white-spaces. In order to instruct Spring SAML to keep the assertion in the original form (keep its DOM) set property *releaseDOM* to *false* on bean *WebSSOProfileConsumerImpl*.

Assertion can be serialized to String using the following call:

```
XMLHelper.nodeToString(SAMLUtil.marshallMessage(credential.getAuthenticationAssertion()))
```

9.6 Authentication log

Key events such as single sign-on and single logout initialization, success or failure can be logged for creation of an audit trail. A custom logger can be created by implementing interface *org.springframework.security.saml.log.SAMLLogger* and including its bean in the *securityContext.xml*, e.g.:

```
<bean id="samlLogger" class="org.springframework.security.saml.log.SAMLDefaultLogger"/>
```

Two basic implementations are provided by default:

- org.springframework.security.saml.log.SAMLEmptyLogger

  Doesn't perform any logging, simply ignores all events.

- org.springframework.security.saml.log.SAMLDefaultLogger

  Logs events as INFO level messages to the log name *org.springframework.security.saml.log.SAMLDefaultLogger* configurable as described in Section 6.6, “Logging”. Setting property *logMessages* to *true* will include content of the SAML messages as part of the log. Logging of exceptions can be disabled by setting *logErrors* to *false*. Fields are semicolon separated with the following values:

  - type of SAML message (AuthNRequest, AuthNResponse, LogoutRequest or LogoutResponse)
  - result of processing (SUCCESS or FAILURE)
  - IP address of the peer who made the current request to SP
  - entity ID of the local SP

- entity ID of the remote IDP
- identifier of the authenticated user
- SAML message (when logMessages is enabled)
- text of the error (only for failures, when logErrors is enabled)

The logger is only called for messages which can be correctly received and parsed. For errors which occur before correct parsing see Section 6.5, “Error handling”.

10. Advanced configuration

10.1 Reverse proxies and load balancers

SAML Extension can be deployed in scenarios where mutliple back-end servers process SAML requests forwarded by a reverse-proxy or a load balancer. SSL termination proxies which communicate using an unencrypted channel between the proxy and back-end servers are also supported. In order to configure SAML Extension for deployment behind a load balancer or a reverse-proxy please follow these steps:

- Make sure that your reverse-proxy or load-balancer is configured to use sticky sessions. Information about e.g. sent requests is stored within a user's HTTP session and sending of response to another back-end node would make the original request data unavailable and fail the validation. Sticky session are not necessary in case only IDP-initialized SSO is used or when sessions are replicated to all nodes.
- Provide information about front-end URL to the back-end servers by changing the *contextProvider* bean implementation in your *securityContext.xml* to class *org.springframework.security.saml.context.SAMLContextProviderLB*:

  ```
  <bean id="contextProvider" class="org.springframework.security.saml.context.SAMLContextProviderLB">
   <property name="scheme" value="https"/>
   <property name="serverName" value="www.myserver.com"/>
   <property name="serverPort" value="443"/>
   <property name="includeServerPortInRequestURL" value="false"/>
   <property name="contextPath" value="/spring-security-saml2-sample"/>
  </bean>
  ```

  This setting enables the extension to correctly form all generated URLs and verify endpoints of the incoming SAML messages.
- In case you use automatically generated metadata make sure to configure *entityBaseURL* matching the front-end URL in your *metadataGeneratorFilter* bean:

  ```
  <bean id="metadataGeneratorFilter"
    class="org.springframework.security.saml.metadata.MetadataGeneratorFilter">
   <constructor-arg>
    <bean class="org.springframework.security.saml.metadata.MetadataGenerator">
     <property name="entityBaseURL"
      value="https://www.myserver.com/spring-security-saml2-sample"/>
    </bean>
   </constructor-arg>
  </bean>
  ```

10.2 Context provider

Context provider populates information about the local service provider (your application) such as entityId, role, metadata, security keys, SSL credentials and trust engines for verification of signatures and SSL/TLS connections. Once populated context is made available to all components participating in processing of the incoming or outgoing SAML messages. ContextProvider can customized to alter behavior of the SAML Extension. The default implementation *org.springframework.security.saml.context.SAMLContextProviderImpl* relies on information available in the ExtendedMetadata and performs the following steps for creation of the context:

- Locate entityId of the local SP by parsing part of the URL after */alias/* (e.g. myAlias in https://www.myserver.com/saml_extension/saml/sso/alias/myAlias?idp=myIdp) and matching it with property *alias* specified in the ExtendedMetadata. In case the URL doesn't contain any alias part the default service provider configured with property *hostedSPName* on the *metadata* bean is used.

- Populate credential used to decrypt data sent to this service provider. In case ExtendedMetadata specifies property *encryptionKey* it will be used as an alias to lookup a private key from the *keyManager* bean. Otherwise defaultKey of the *keyManager* bean will be used.
- Populate credential used for SSL/TLS client authentication. In case ExtendedMetadata specifies property *tlsKey* it will be used as an alias to lookup key from *keyManager* bean. Otherwise no credential will be provided for client authentication.
- Populate trust engine for verification of signatures. Depending on *securityProfile* setting in the ExtendedMetadata trust engine based on either the section called “Metadata interoperability profile (MetaIOP)” or the section called “PKIX profile” is created.
- Populate trust engine for verification of SSL/TLS connections. Depending on *sslSecurityProfile* setting in the ExtendedMetadata trust engine based on either the section called “Metadata interoperability profile (MetaIOP)” or the section called “PKIX profile” is created.

During initialization of SSO ContextProvider is also requested to provide metadata of the peer IDP. System performs these steps to locate peer IDP to use:

- Load parameter *idp* of the HttpRequest object and try to locate peer IDP by the entityId. When there's no *idp* parameter provided system will either start IDP discovery process (when enabled in the ExtendedMetadata of the local SP) or use the default IDP specified in the *metadata* bean.

10.3 Validity intervals

For security reasons system limits the time window enabling processing of SAML messages and assertions. The time window parameters can be customized with the following settings.

Validity of assertions processed during the signle sign-on process is limited to 3000 seconds. Value can be customized with property *maxAssertionTime* of the *WebSSOProfileConsumerImpl* bean.

System allows users to single sign-on for up to 7200 seconds since their initial authentication with the IDP (based on value AuthInstance of the Authentication statement). Some IDPs allow users to stay authenticated for longer periods than this and you might need to change the default value by setting *maxAuthenticationAge* of the *WebSSOProfileConsumerImpl* bean.

As clocks between IDP and SP machines may not be perfectly synchronized a tolerance of 60 seconds is applied for time comparisons. The tolerance value (time skew) can be customized by settings property *responseSkew* in beans *WebSSOProfileConsumerImpl* and *SingleLogoutProfileImpl*.

The following tables summarize all checks for time validity during processing of incoming SAML messages. Response skew refers to property *responseSkew* set on profile beans. Past indicates that validity window for checking of the value will be extended by *responseSkew* seconds to the past and correspondingly with the future value. Nullable values can be missing from the incoming messages.

Table 10.1. Time checks during processing of incoming SAML Response in WebSSO and WebSSO HoK profiles

response.issueInstant	
Applied skew:	responseSkew (past + future)
Nullable:	No

Fails with:	Throws SAMLException
Description:	Time when SAML response message was created.
response.assertion.issueInstant	
Applied skew:	responseSkew (past + future) + maxAssertionTime (future)
Nullable:	No
Fails with:	Throws SAMLException
Description:	Time when SAML assertion was created, allows validity extension as assertion might be re-used by the caller.
response.assertion.subject.subjectConfirmation.notOnOrAfter	
Applied skew:	responseSkew (future)
Nullable:	No
Fails with:	Throws SAMLException
Description:	Time when subject can no longer be confirmed.
response.assertion.authnStatement.authnInstant	
Applied skew:	responseSkew (past + future) + maxAuthenticationAge (future)
Nullable:	No
Fails with:	Throws CredentialsExpiredException
Description:	Time when user authenticated to IDP, typically differs from time or response or assertion creation time.
response.assertion.authnStatement.sessionNotOnOfAfter	
Applied skew:	no skew
Nullable:	Yes
Fails with:	Throws CredentialsExpiredException
Description:	Time when user's session expires and requires re-authentication, sessions are typically valid for longer period and therefore do not suffer from time synchronization problems.
response.assertion.condition.notBefore	
Applied skew:	responseSkew (past)
Nullable:	Yes

Fails with:	Throws SAMLException
Description:	Time limit on validity of assertion.
response.assertion.condition.notOnOrAfter	
Applied skew:	responseSkew (future)
Nullable:	Yes
Fails with:	Throws SAMLException
Description:	Time limit on validity of assertion.

Table 10.2. Time checks during processing of incoming SAML LogoutRequest in Single Logout profile

response.issueInstant	
Applied skew:	responseSkew (past + future)
Nullable:	No
Fails with:	Sends LogoutResponse with error Status "urn:oasis:names:tc:SAML:2.0:status:Requester"
Description:	Time when SAML LogoutRequest message was created.

Table 10.3. Time checks during processing of incoming SAML LogoutResponse in Single Logout profile

response.issueInstant	
Applied skew:	responseSkew (past + future)
Nullable:	No
Fails with:	Throws SAMLException
Description:	Time when SAML LogoutResponse message was created.

Table 10.4. Time checks during processing of incoming SAML ArtifactResponse in Artifact Resolution profile

response.issueInstant	
Applied skew:	responseSkew (past + future)
Nullable:	No
Fails with:	Throws MessageDecodingException
Description:	Time when SAML LogoutResponse message was created.

10.4 Enhanced client/proxy

Support for enhanced client/proxy can be configured using property *ecpEnabled* of the service provider's extended metadata. Once enabled, ECP profile is automatically activated with requests containing HTTP headers *Accept: application/vnd.paos+xml* and *PAOS: ver='urn:liberty:paos:2003-08'; 'urn:oasis:names:tc:SAML:2.0:profiles:SSO:ecp'*. Binding used to server ECP profile is always automatically set to PAOS.

ECP can be enabled in combination with the automatic metadata generation using the following settings:

```
<bean class="org.springframework.security.saml.metadata.MetadataGenerator">
 <property name="extendedMetadata">
  <bean class="org.springframework.security.saml.metadata.ExtendedMetadata">
   <property name="ecpEnabled" value="true"/>
  </bean>
 </property>
</bean>
```

10.5 Endpoint URLs

By default Spring SAML uses the following endpoints, which can optionally also contain information about entity alias of the local Service Provider:

Table 10.5. Endpoint overview

Profile	**Binding**	**Endpoint**
Web Single Sign-on	HTTP-POST, HTTP-Artifact, PAOS	scheme://server:port/contextPath/saml/SSO
Web Single Sign-on Holder of Key	HTTP-POST, HTTP-Artifact	scheme://server:port/contextPath/saml/HoKSSO
Single Logout	HTTP-POST, HTTP-Redirect	scheme://server:port/contextPath/saml/SingleLogout

The default URLs can be altered with these steps:

- change property *filterProcessesUrl* on the corresponding processing bean (*samlWebSSOProcessingFilter*, *samlWebSSOHoKProcessingFilter*, *samlLogoutProcessingFilter* or *samlIDPDiscovery*) to the new URL, for example */samlResponse*
- update the *samlFilter* bean and make sure that the modified processing filter is mapped to the correct pattern, for example */samlResponse/***, the */*** part is only needed in case you're using the entity alias feature
- re-generate metadata for your service provider, in case you are using automatic metadata generator the endpoints will be automatically generated with the new URLs
- in case you are using pre-configured metadata you can perform changes manually in your existing metadata file

Endpoints of filters *samlEntryPoint*, *samlLogoutFilter* and *metadataDisplayFilter* can be changed using the same process and without need to re-generate the metadata.

10.6 Artifact resolution

Usage of HTTP-Artifact binding requires Spring SAML to make a direct SOAP call to the Identity Provider. Sometimes it's necessary to configure correct HTTP proxy for the call. This can be achieved by setting property *hostConfiguration* on *HttpClient* plugged to the *artifactBinding* bean. The following configuration demonstrates creation of the bean for the *hostConfiguration*:

```
<bean id="hostConfiguration" class="org.apache.commons.httpclient.HostConfiguration"/>
<bean class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
 <property name="targetObject" ref="hostConfiguration"/>
 <property name="targetMethod" value="setProxy"/>
 <property name="arguments">
  <list>
   <value>testHost</value>
   <value>8080</value>
  </list>
 </property>
</bean>
```

Another common use-case is situation when artifact resolution endpoint at IDP is secured using HTTP-Basic authentication. Authentication can be configured by setting *HTTPClient's* property *state* with the following bean:

```
<bean id="state" class="org.apache.commons.httpclient.HttpState"/>
<bean class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
 <property name="targetObject" ref="state"/>
 <property name="targetMethod" value="setCredentials"/>
 <property name="arguments">
  <list>
   <util:constant static-field="org.apache.commons.httpclient.auth.AuthScope.ANY"/>
   <bean class="org.apache.commons.httpclient.UsernamePasswordCredentials">
    <constructor-arg value="username"/>
    <constructor-arg value="password"/>
   </bean>
  </list>
 </property>
</bean>
```

Part III. Sample application

Chapter provides reference for the sample application and its administration user interface.

11. Sample application

Spring SAML includes a sample application which demonstrates key capabilities of this product. For details on compilation and deployment of the sample application please see Chapter 4, *Quick start guide*.

Public demo of the sample application is available at saml-federation.appspot.com

11.1 SAML login

Sample application demonstrates usage of IDP discovery which is automatically invoked on access to the application root. Discovery presents selection of all available Identity Providers and initiates SAML 2.0 single sign-on with the selected IDP after clicking on the *"Start single sign-on"* button.

After authentication at IDP, sample application displays information about the received and validated assertion, or displays errors encountered during validation.

Clicking buttons *"Global Logout"* and *"Local Logout"* initializes the logout process as described in Section 9.3, “Logout process”.

11.2 Metadata administration

Sample application contains an administration UI which enables simple monitoring and administrative use-cases. You can access the UI by clicking on *"Metadata Administration"* button.

Administration part is secured with role *ROLE_ADMIN* and uses local authentication with default username *admin* and password *admin*. As Spring Security allows only one authentication to be currently active, authenticating to administration UI will remove any previous SAML authentication from the security context.

Metadata administration enables the following operations:

- Displaying of existing identity provider and service provider entities by clicking on their identifier. Information includes content of the metadata and extended metadata for the entity.
- Displaying of existing metadata providers and possibility to remove them.
- Refreshing of all metadata providers by clicking on button *"Refresh metadata"*.
- Generation of new metadata by clicking on *"Generate new service provider metadata"*.

11.3 Metadata generation

Metadata generator allows dynamic creation of service provider metadata based on values provided in the UI. Metadata can be immediately applied to the currently running instance by setting *"Store for current session"* option to *"Yes"*.

Options available in the interface are discussed in the section called “Automatic metadata generation” and Section 7.3, “Extended metadata”. The generated values can be used as input for pre-configured metadata described in the section called “Pre-configured metadata”.

Part IV. Integration guide

This chapter includes step-by-step instructions on basic steps required for enabling single sign-on with common identity providers.

12. Integrating Identity Providers

Section provides additional information regarding integration of Spring SAML with popular Identity Providers.

12.1 Active Directory Federation Services 2.0 (AD FS)

AD FS 2.0 supports SAML 2.0 in IDP mode and can be easily integrated with SAML Extension for both SSO and SLO. Before starting with the configuration make sure that the following pre-requisites are satisfied:

- Install AD FS 2.0 (http://www.microsoft.com/en-us/download/details.aspx?id=10909)
- Run AD FS 2.0 Federation Server Configuration Wizard in the AD FS 2.0 Management Console
- Make sure that DNS name of your Windows Server is available at your SP and vice versa
- Install a Java container (e.g. Tomcat) for deployment of the SAML 2 Extension
- Configure your container to use HTTPS, this is required by AD FS (http://tomcat.apache.org/tomcat-6.0-doc/ssl-howto.html)

Initialize IDP metadata

- Download AD FS 2.0 metadata from e.g. *https://adfsserver/FederationMetadata/2007-06/FederationMetadata.xml*
- Store the downloaded content to *sample/src/main/resources/metadata/FederationMetadata.xml*
- Modify bean *metadata* in *sample/src/main/webapp/WEB-INF/securityContext.xml* and replace *classpath:security/idp.xml* with *classpath:security/FederationMetadata.xml* and add property *metadataTrustCheck* to *false* to skip signature validation:

```
<bean class="org.springframework.security.saml.metadata.ExtendedMetadataDelegate">
 <constructor-arg>
  <bean class="org.opensaml.saml2.metadata.provider.ResourceBackedMetadataProvider">
   <constructor-arg>
    <bean class="java.util.Timer"/>
   </constructor-arg>
   <constructor-arg>
    <bean class="org.opensaml.util.resource.ClasspathResource">
     <constructor-arg value="/metadata/FederationMetadata.xml"/>
    </bean>
   </constructor-arg>
   <property name="parserPool" ref="parserPool"/>
  </bean>
 </constructor-arg>
 <constructor-arg>
  <bean class="org.springframework.security.saml.metadata.ExtendedMetadata"/>
 </constructor-arg>
 <property name="metadataTrustCheck" value="false"/>
</bean>
```

Initialize SP metadata

- Deploy SAML 2 Extension war archive from *sample/target/spring-security-saml2-sample.war*, or use embedded Tomcat with command: *mvn tomcat7:run*
- Open Spring SAML in browser, e.g. at https://localhost:8443/spring-security-saml2-sample, making sure to use HTTPS protocol
- Click Metadata Administration, login and select item with your server name from the Service providers list

- Store content of the Metadata field to a document metadata.xml and upload it to the AD FS server
- In AD FS 2.0 Management Console select "Add Relying Party Trust"
- Select "Import data about the relying party from a file" and select the metadata.xml file created earlier. Select Next
- The wizard may complain that some content of metadata is not supported. You can safely ignore this warning
- Continue with the wizard. On the "Ready to Add Trust" make sure that tab endpoints contains multiple endpoint values. If not, verify that your metadata was generated with HTTPS protocol URLs
- Leave "Open the Edit Claim Rules dialog" checkbox checked and finish the wizard
- Select "Add Rule", choose "Send LDAP Attributes as Claims" and press Next
- Add NameID as "Claim rule name", choose "Active Directory" as Attribute store, choose "SAM-Account-Name" as LDAP Attribute and "Name ID" as "Outgoing claim type", finish the wizard and confirm the claim rules window, in ADFS 3.0 you might need to configure the Name ID as a Pass Through claim
- Open the provider by double-clicking it, select tab Advanced and change "Secure hash algorithm" to SHA-1

Test SSO

Open the Spring SAML sample application at e.g. https://localhost:8443/spring-security-saml2-sample, select your AD FS server and press login. In case Artifact binding is used and SSL/TLS certificate of your AD FS is not already trusted, import it to your samlKeystore.jks by following instructions in the error report.

12.2 Okta

Okta supports single sign-on to customer specified SAML 2.0 Service Provider applications, such as Spring SAML Extension. Before starting with the configuration make sure that the following pre-requisites are satisfied:

- Have an Okta instance and administration account ready, Okta license must allow you to add custom applications
- Install a Java container (e.g. Tomcat) for deployment of the SAML 2 Extension

Deploy Spring SAML sample application

- Deploy SAML 2 Extension war archive from *sample/target/spring-security-saml2-sample.war*, or use embedded Tomcat with command: *mvn tomcat7:run*
- Open Spring SAML in browser, e.g. at http://localhost:8080/spring-security-saml2-sample
- Click Metadata Administration, login and select item with your server name from the Service providers
- Note the *Entity ID field*, and *Assertion Consumer Service URL (ACS)* from the metadata XML, e.g. *http://localhost:8080/spring-security-saml2-sample/saml/SSO*

Information such as entity ID and URLs of your Spring SAML can be customized, see Section 7.1, “Service provider metadata” for details.

Configure Okta

- Login to Okta as an administrator, select *Admin*, select *Applications* and click *Create New App*
- From the list of supported protocols select SAML 2.0 and press *Create*

- Define app name (e.g. Spring SAML), optionally define app image and press *Next*
- Configure SAML with the following settings:

Table 12.1.

Single Sign on URL	Use value noted during Spring SAML initialization, e.g. *http://localhost:8080/spring-security-saml2-sample/saml/SSO*
Audience URI (SP Entity ID)	Use value noted during Spring SAML initialization, e.g. *http://localhost:8080/spring-security-saml2-sample/saml/metadata*
Default RelayState	Leave empty, unless you require Okta to provide a value to Spring SAML
Name ID format	Select any of the available options, depending on your requirements
Application username	Select any of the available options, depending on your requirements
Response (advanced settings)	Select "signed"
Assertion (advanced settings)	Select "signed"
Authentication context class (advanced settings)	Select any of the available options
Request compression (advanced settings)	Select "Uncompressed"

- Optionally define attributes to be sent to Spring SAML after single sign-on, and press *Next*
- On Feedback page select "This is an internal application that we created" and press *Finish*
- Make sure to distribute the newly created application to users you want to use for testing

Import Okta metadata to Spring SAML

- In Okta click link "Identity provider metadata" and store the downloaded content to *sample/src/main/resources/metadata/okta.xml*
- In Spring SAML modify bean *metadata* in *sample/src/main/webapp/WEB-INF/securityContext.xml* and replace *classpath:security/idp.xml* with *classpath:security/okta.xml*:

```
<bean class="org.springframework.security.saml.metadata.ExtendedMetadataDelegate">
    <constructor-arg>
        <bean class="org.opensaml.saml2.metadata.provider.ResourceBackedMetadataProvider">
            <constructor-arg>
                <bean class="java.util.Timer"/>
            </constructor-arg>
            <constructor-arg>
                <bean class="org.opensaml.util.resource.ClasspathResource">
                    <constructor-arg value="/metadata/okta.xml"/>
                </bean>
            </constructor-arg>
            <property name="parserPool" ref="parserPool"/>
        </bean>
    </constructor-arg>
    <constructor-arg>
        <bean class="org.springframework.security.saml.metadata.ExtendedMetadata"/>
    </constructor-arg>
</bean>
```

- Restart Spring SAML for the changes to get applied

Test SSO

Open the Spring SAML sample application at e.g. http://localhost:8080/spring-security-saml2-sample, select your Okta server and press login. Alternatively start IDP-initialized single sign-on using *App Embed Link* provided by Okta in application configuration, e.g. *https://v7security.okta.com/home/v7security_springsaml_1/0oa4vkeakAsUtZ8AI0y6/39139*.

13. Troubleshooting common problems

Time synchronization

Processing of SAML messages and assertions is often limited to a specific time window which e.g. prevents possibilities of replay attacks. Validation of messages can fail when internal clocks of the IDP and SP machines are not synchronized. Make sure to use a time synchronization service on all systems in the federation.

Error 'InResponseToField doesn't correspond to sent message' during SSO

Make sure that application uses the same HttpSession during sending of the request and reception of the response. Typically, this problem arises when the auhentication request is initialized from localhost address or http scheme, while response is received at a public host name or https scheme. E.g., when initializing authentication from URL https://host:port/app/saml/login, the response must be received at https://host;port/app/saml/SSO, not http://host:port/app/saml/SSO or https://localhost:port/app/saml/SSO.

The checking of the InResponseToField can be disabled by re-configuring the context provider as follows:

```
<bean id="contextProvider" class="org.springframework.security.saml.context.SAMLContextProviderImpl">
  <property name="storageFactory">
    <bean class="org.springframework.security.saml.storage.EmptyStorageFactory"/>
  </property>
</bean>
```

System is redirecting to e.g. localhost address when public facing URL is different

In case you use automatic metadata generation make sure to set property entityBaseURL on bean MetadataGenerator to e.g. http://server:port/yourapp or use pre-generated metadata.

System fails during decryption or encryption of fields, e.g. with 'Failed to decrypt EncryptedData'

Make sure the *Unlimited Strength Jurisdiction Policy Files* are correctly installed in your JDK. See Section 4.1, “Pre-requisites” for details.

My system fails during validation of certificates with errors similar to "PKIX path building failed"

This is typically caused by misconfiguration of certificates. Either your metadata or keyStore do not contain the correct leaf certificates or CA certificates, or your certificates are invalid. You can get additional information by starting your application with flag *-Djavax.net.debug=all*.