A Guide to OpenSAML V3

By

Stefan Rasmusson

DISCLAIMER

While all attempts have been made to verify information provided in this publication, the author assumes no responsibility for errors, omissions, or contrary interpretation of the subject matter herein.

Any perceived slights of specific people or organizations are unintentional. The images in this book have been used solely for educational purposes and remain the intellectual property of respective owners.

The information included in this book cannot be compared with the guidelines provided in other books related to OpenSAML. Readers are advised to look for professional guidance if necessary.

About the Author

My first real encounter with IT and software development was when my grandpa taught me programming in the old classic language, Pascal, at the age of 14. Although this didn't turn out to be the highlight of my career, it was, nevertheless a humble beginning to what would later become my lifelong passion and desire for computers.

Several years later, I pursued this interest in higher education, studying Computer Engineering at Umeå University in my hometown in Northern Sweden. After finishing my studies, I was hired as a Java consultant at a firm where I spent five years working on security as my major focus. During this time, I encountered interesting challenges and my interest for security only grew stronger over time.

Most recently, I was hired as a principal consultant for Oracle to design solutions using their security products. Simultaneously, I am completing my master's degree in Information Security. This has given me a chance to indulge in numerous developing projects carried out on a small-scale.

I am not an author. However, my enthusiasm and motivation in this subject has enabled me to write a book that can engage like-minded individuals and those who have an insatiable appetite for computers and information security.

- Stefan Rasmusson

About this Book

Motivation

With several years of experience on the OpenSAML framework, I was stunned to notice the lack of reliable sources on this subject. I had to take the hard way of carrying out extensive research, gathering relevant information by clicking through endless Java docs and hunting among the source codes. The project finally turned out to be a success and I managed to use OpenSAML to integrate with a governmental identity provider.

While I was working and hitting my head against the wall, I started my own blog site on this subject and found that I was not the only one facing this problem. The response of my blog site was overwhelming and many people working with OpenSAML found it extremely informative and useful.

Keeping this in mind, I decided to write a book that would serve as an extensive resource for people who are keen on using the best possible information on OpenSAML framework. This book is a step ahead for people looking for reliable and relevant information on OpenSAML framework.

Aim

The main purpose of this book is to provide a step-by-step guide for users on how to authenticate Web Browser SSO (Single Sign-On) profile using OpenSAML in Java. The scope of this book will only cover authentication from the view of the Service Providers, the party that is requesting to authenticate a user. SAML is a very versatile framework and can be used in many different ways. However, it is important to note that this book focuses on one of many possible ways to use it.

This book will show SAML usage with the following configurations:

- SP initiated Single Sign-On
- Authentication request using HTTP Redirect Binding
- Assertion transported using HTTP Artifact Binding
- SAML Artifact transported using HTTP Redirect Binding
- Artifact resolution using SOAP Binding

As cryptography is a central part of protecting SAML communication, the following cryptographic functions will be used and explained:

- Retrieving credentials to use with cryptographic functions

- Signing AuthnRequest using HTTPRedirectDeflateEncoder
- Signing Assertion using Signer utility class
- Verifying Assertion using signature validators
- Encrypting and decrypting Assertion

Summary of Contents

Part I - Introduction

This part introduces you to the book and focuses on how to make the best use of the information provided. It also introduces the SAML protocol and its main building blocks and guides you on how to set up the basic boilerplate needed to begin using OpenSAML.

Part II - Implementing a Service Provider

Here is where the magic happens. In this part the authentication process of intercepting, authenticating and getting relevant information from the user is described. The entire process is illustrated using the code from the sample project presented in Part I.

Part III - Security Features

This part shows how to use some of the cryptographic functions in OpenSAML.

Summary of the Migration Process

Those of you who had bought the previous edition of the book might be interested to know about changes since the last edition. In this chapter, we will try to provide summary of changes in the OpenSAML framework between versions 2 and 3.

Migration Summary

To help you with the migration process from version 2 to version 3, a rudimentary migration guide is provided in this section. The migration guide is based on the steps needed to migrate the sample project of this book. As a result, the guide will serve as an acceptable starting point for other projects.

Some steps in the migration process are more extensive than others. These are explained in more detail in the following sections.

- The updated sample project for OpenSAML version 3 is available at https://bitbucket.org/srasmusson/webprofile-ref-project-v3
- The structure of the OpenSAML dependencies has changed between version 2 and 3. The functionality is not split over multiple jars.
- The MessageContext has changed to become more modular. MessageHandlers has been added to process the messages using message context.
- The syntax for using the HTTPRedirectDeflateEncoder has changed to a small degree, mainly due to the new design of the message context.
- The syntax for sending SOAP messages has changed substantially.
- In OpenSAML version 2 the Configuration class was used to get a builder factory to build SAML objects and marshaller to marshall SAML messages to text. In version 3 the Configuration class is replaced with XMLObjectProviderRegistrySupport for this purpose, using the methods XMLObjectProviderRegistrySupport.getBuilderFactory and XMLObjectProviderRegistrySupport.getMarshallerFactory().getMarshaller(object)
- The EntityIdCriteria class used for resolving credentials is renamed to EntityIdCriterion.
- The StatusCode.SUCCESS_URI used in the ArtifactResponse has been renamed to StatusCode.SUCCESS
- A lot of package names have changed from version 2 to 3. The easiest way to know about the changes is to search for the classes in an IDE.

- The EncryptionParameters class used for specifying encryption options is renamed to DataEncryptionParameters.
- The SecureRandomIDGenerator class used to generate IDs for SAML messages is renamed to RandomIdentifierGenerationStrategy.
- The Configuration.validateNonSunJAXP method has been removed as it is not longer necessary to validate this. Configuration.validateJCEProviders method has been replaced by the init method of JavaCryptoValidationInitializer.
- The SecurityHelper.generateKeyPair method has been moved to the KeySupport class and the SecurityHelper.getSimpleCredential method has been moved to the CredentialSupport class.

Dependencies

The structure of the OpenSAML dependencies has changed a lot between version 2 and version 3. In version 2 all the functionality of OpenSAML was contained in one jar file. In version 3, on the other hand, it is organized as a Maven multi-module Maven project with different functionality in its own module.

For now, users are not provided with a single dependency for the whole of OpenSAML, so each required module must be added as its own dependency. The following modules exist in the latest version of OpenSAML.

- opensaml-core
- opensaml-profile-api
- opensaml-profile-impl
- opensaml-soap-api
- opensaml-soap-impl
- opensaml-saml-api
- opensaml-saml-impl
- opensaml-xacml-api
- opensaml-xacml-impl
- opensaml-xacml-saml-api
- opensaml-xacml-saml-impl
- opensaml-messaging-api
- opensaml-messaging-impl
- opensaml-storage-api
- opensaml-storage-impl
- opensaml-security-api
- opensaml-security-impl
- opensaml-xmlsec-api
- opensaml-xmlsec-impl

This modularity enables the project to only include the dependencies that are most needed. The dependencies can be added using the following syntax in Maven

```
<dependency>
<groupId>org.opensaml</groupId>
<artifactId>opensaml-core</artifactId>
<version>3.2.0</version>
</dependency>
```

Note that the following modules are used in the sample project.

- opensaml-core
- opensaml-saml-api
- opensaml-saml-impl
- opensaml-messaging-api
- opensaml-messaging-impl
- opensaml-soap-api
- opensaml-soap-impl

Message Contexts

When sending a message back in version 2 of OpenSAML, a BasicSAMLMessageContext was used to contain all the information about the SAML message, where it was going to be sent, signing information etc. In the current version of OpenSAML, however, the MessageContext structure has been split up into many separate classes, each containing a specific part of information. Below are some examples.

- MessageContext - Main context.
- SAMLPeerEntityContext - A context containing information about a peer entity, in other words, the IdP for the SP and vice versa. This context itself does not contain much information, but usually contain one or more sub-contexts.
- SAMLEndpointContext - This context contains information about a specific endpoint of the peer entity.
- SecurityParametersContext - The context is used for holding information about the signature and/or encryption.
- SAMLMessageInfoContext - The context is used for holding basic info about issue instant and id.

The contexts are created as sub-contexts of the main message context or each other. The following is an example:

```
MessageContext context = new MessageContext();

SAMLPeerEntityContext peerEntityContext
    = context.getSubcontext(SAMLPeerEntityContext.class, true);

SAMLEndpointContext endpointContext
    = peerEntityContext.getSubcontext(SAMLEndpointContext.class,
        true);
```

The last parameter of the getSubcontext method is used to instruct the method to create the sub-context if it doesn't already exist. Once the context is created it is populated using the available setters, as shown below.

```
endpointContext.setEndpoint(getIPDEndpoint());
```

Message Handlers

Another exciting new feature in version 3 is a collection of message handlers. The message handlers process the messages providing among other things message validation, signature verification and signing. This is done usually before an encoder or decoder encodes or decodes the message onto a channel. Below are some of the available handlers.

- MessageLifetimeSecurityHandler – lifetime validation
- SAMLOutboundProtocolMessageSigningHandler – signing outbound messaged
- ReceivedEndpointSecurityHandler – validation of destination

When invoking the handlers they use context information needed by that specific handler. Using the above handlers as an example:

- MessageLifetimeSecurityHandler will require SAMLMessageInfoContext, containing information about issue time etc.
- SAMLOutboundProtocolMessageSigningHandler will require SecurityParametersContext, containing singing parameters
- ReceivedEndpointSecurityHandler only needs the base message context, containing the SAML message. The required information is extracted directly from the message.

Message handlers can be invoked either directly as shown below.

```
SAMLOutboundProtocolMessageSigningHandler handler
= new SAMLOutboundProtocolMessageSigningHandler();
handler.initialize();
handler.invoke(context);
```

This can also be done indirectly by a client implementation, for instance with PipelineHttpSOAPClient, when sending SOAP messages. Example of this is shown in the next section.

As is often the case, there is usually more than one handler that should be used on a message. Instead of initializing and invoking each handler they can be invoked one after the other using a BasicMessageHandlerChain.

```
List handlers = new ArrayList<MessageHandler>();
```

```
handlers.add(handler1);
handlers.add(handler2);

BasicMessageHandlerChain<ArtifactResponse> handlerChain
    = new BasicMessageHandlerChain<ArtifactResponse>();
handlerChain.setHandlers(handlers);
```

Below is an example of message handler usage for validating the destination endpoint and the lifetime of the message. The handlers are called using a handler chain.

The main message context is created and the message is added as shown below.

```
MessageContext context = new MessageContext<ArtifactResponse>();
context.setMessage(artifactResponse);
```

The SAMLMessageInfoContext is populated with general information about the message, in this case the issue instant as depicted below.

```
SAMLMessageInfoContext messageInfoContext
    = context.getSubcontext(SAMLMessageInfoContext.class, true);
messageInfoContext.setMessageIssueInstant(
    artifactResponse.getIssueInstant());
```

The MessageLifetimeSecurityHandler used for validation of message lifetime is populated with the application requirements for message lifetime as shown here.

```
MessageLifetimeSecurityHandler lifetimeSecurityHandler
    = new MessageLifetimeSecurityHandler();

lifetimeSecurityHandler.setClockSkew(1000);
lifetimeSecurityHandler.setMessageLifetime(2000);
lifetimeSecurityHandler.setRequiredRule(true);
```

The ReceivedEndpointSecurityHandler is populated with the servlet request. The handler extracts the required destination from the servlet request object as can be seen below.

```
ReceivedEndpointSecurityHandler receivedEndpointSecurityHandler
    = new ReceivedEndpointSecurityHandler();
receivedEndpointSecurityHandler.setHttpServletRequest(request);
```

The handlers are added to a handler chain, as shown below.

```
List handlers = new ArrayList<MessageHandler>();
handlers.add(lifetimeSecurityHandler);
handlers.add(receivedEndpointSecurityHandler);

BasicMessageHandlerChain<ArtifactResponse> handlerChain
    = new BasicMessageHandlerChain<ArtifactResponse>();
handlerChain.setHandlers(handlers);
```

Finally, the handler chain is initialized and invoked.

```
handlerChain.initialize();
handlerChain.doInvoke(context);
```

Sending Messages using HTTPRedirectDeflateEncoder

The syntax for using the HTTPRedirectDeflateEncoder has changed some, mostly due to the new design of message context.

At first, the main message context is created.

```
MessageContext context = new MessageContext();
```

Secondly, set the message to be sent.

```
context.setMessage(authnRequest);
```

Thirdly, the SAMLEndpointContext inside the SAMLPeerEntityContext is created, containing the destination endpoint.

```
SAMLPeerEntityContext peerEntityContext
    = context.getSubcontext(SAMLPeerEntityContext.class, true);
SAMLEndpointContext endpointContext
    = peerEntityContext.getSubcontext(SAMLEndpointContext.class,
        true);
endpointContext.setEndpoint(getIPDEndpoint());
```

Fourthly, the SecurityParametersContext is optional. However, here it is created and populated with singing parameters to provide a signature to the message.

```
SignatureSigningParameters signatureSigningParameters
    = new SignatureSigningParameters();
signatureSigningParameters.setSigningCredential(
    SPCredentials.getCredential());
signatureSigningParameters.setSignatureAlgorithm(
    SignatureConstants.ALGO_ID_SIGNATURE_RSA_SHA256);

context.getSubcontext(SecurityParametersContext.class,
true).setSignatureSigningParameters(signatureSigningParameters);
```

Fifthly, create the encoder and populate it with the message context and the servlet response

```
HTTPRedirectDeflateEncoder encoder
    = new HTTPRedirectDeflateEncoder();

encoder.setMessageContext(context);
encoder.setHttpServletResponse(httpServletResponse);
```

Lastly, the encoder is initialized and the message is encoded

```
encoder.initialize();
encoder.encode();
```

Sending and Receiving Messages using SOAP

The syntax for sending SOAP messages has changed quite drastically since version 2 much due to the new design of message context and handlers etc.

First the main message context is created

```
MessageContext<ArtifactResolve> contextout
    = new MessageContext<ArtifactResolve>();

contextout.setMessage(artifactResolve);
```

There are no mandatory contexts needed for sending SOAP messages, but as it is often used the security parameters context is included to provide information needed for signing the message.

```
SignatureSigningParameters signatureSigningParameters
    = new SignatureSigningParameters();
signatureSigningParameters.setSignatureAlgorithm(
    SignatureConstants.ALGO_ID_SIGNATURE_RSA_SHA256);
signatureSigningParameters.setSigningCredential(
    SPCredentials.getCredential());
signatureSigningParameters.setSignatureCanonicalizationAlgorithm(
    SignatureConstants.ALGO_ID_C14N_EXCL_OMIT_COMMENTS);

SecurityParametersContext securityParametersContext
    = contextout.getSubcontext(SecurityParametersContext.class,
        true);
securityParametersContext.setSignatureSigningParameters(
        signatureSigningParameters);
```

As this operation is a two way communication, an InOutOperationContext is used to hold the outgoing and incoming message context. The InOutOperationContext is created and the outgoing context is added.

```
InOutOperationContext<ArtifactResponse, ArtifactResolve> context
    = new ProfileRequestContext<ArtifactResponse, ArtifactResolve>();
context.setOutboundMessageContext(contextout);
```

In order to send the SOAP message, a soap client is needed. The client invokes message handlers, encoder and decoder in order to transform and send the message. The handlers, encoder and decoder are contained inside of a pipeline.

The way that a client is created is by extending AbstractPipelineHttpSOAPClient and implementing the newPipeline method. This method should return the pipeline that should be used by the client when sending the message.

```
AbstractPipelineHttpSOAPClient<SAMLObject, SAMLObject> soapClient
    = new AbstractPipelineHttpSOAPClient() {
       protected HttpClientMessagePipeline newPipeline() throws
           SOAPException {
```

Create the encoder and decoder used to transport the message

```
HttpClientRequestSOAP11Encoder encoder
    = new HttpClientRequestSOAP11Encoder();
HttpClientResponseSOAP11Decoder decoder
    = new HttpClientResponseSOAP11Decoder();
```

Create the pipeline

```
BasicHttpClientMessagePipeline pipeline
    = new BasicHttpClientMessagePipeline(
       encoder,
       decoder
);
```

The SAMLOutboundProtocolMessageSigningHandler is set as an outbound message handler to sign the outgoing message.

```
pipeline.setOutboundPayloadHandler(
    new SAMLOutboundProtocolMessageSigningHandler());
```

A HttpClient is provided to the SOAP client to facilitate encoding and decoding to HTTP.

```
HttpClientBuilder clientBuilder = new HttpClientBuilder();
soapClient.setHttpClient(clientBuilder.buildClient());
```

Finally the SAML message is sent using the clients send method, specifying the destination endpoint and the message context.

```
soapClient.send(IDPConstants.ARTIFACT_RESOLUTION_SERVICE, context);
```

The method waits until the message is delivered and the response is received or a timeout is reached. When the response is received the SAML message can be retrieved from the incoming message context.

```
return context.getInboundMessageContext().getMessage();
```

Table of Contents

PART-I

Chapter 1: All you Need to Know about OpenSAML

I. What is OpenSAML?

OpenSAML V2 end of life
At July 31 of 2016 version 2 of OpenSAML officially reached its end of life. This means that the library will not be receiving any new updates, not even critical security updates.
Because of this, everyone currently using version 2 of OpenSAML is strongly advised to migrate to OpenSAML version 3 or look into other solutions. This book is written for OpenSAML version 3.

OpenSAML is a library to facilitate working with SAML messages. Below are some of the functions that OpenSAML provides:

- Creating SAML messages
- Parsing and exporting SAML objects as XML
- Signing and encryption
- Encoding and message transport

Internet2 provides and supports the library. Shibboleth products, produced by internet2, are one of the examples of identity solutions that utilize the OpenSAML library. The OpenSAML library is available in Java and C++, however; not all functions are provided in both versions. OpenSAML is licensed under Apache 2.0 and the latest version of OpenSAML supports SAML 2.0, 1.1 and 1.0.

Myths and Realities of OpenSAML

***Myth*:** There seems to be a common misunderstanding that OpenSAML is a Single Sign-On (SSO) product. To save yourself a lot of work, it is very important to understand what OpenSAML does not provide.

***Reality*:** OpenSAML is a low-level library for working with SAML. Low level, in this case, means that the library does not provide any complete service and is not by itself a service- or identity-provider. Instead, it provides the functionality to help you build your own service. This characteristic makes it very suitable for building custom implementations that use SAML. However, it falls short as far as standard cases are concerned. For standard cases, there are a lot of products that solve the problem in a much better way, saving lot of time and effort. Shibboleth is a perfect example of products where the OpenSAML library is used to provide solutions for commonly used

cases. As we advance towards further chapters and in the closing pages of the book, there are more examples of products that solve common use cases for SAML.

It is important to note that using OpenSAML in a secure and correct way requires a good understanding of the SAML specifications. Although, this book guides you with a brief explanation to fulfill this requirement, detailed explanation of every specification is beyond the scope of this book. Therefore, it is recommended to read the reference section for additional information.

II. What you Need to Know

OpenSAML is a Java library that helps to build and transport messages over the HTTP protocol. For this purpose, it is important to have basic experience and understanding of Java, web applications and the HTTP protocol.

In addition, OpenSAML is a set of libraries meant to support developers working with SAML protocol. Therefore; it is good to have an understanding of SAML itself before working with the library. In this book, it is not assumed that you have this knowledge. However, to build a secure and acceptable product using OpenSAML, it is recommended to have a good understanding of the SAML protocol.

This book will introduce the SAML protocol, its parts, and the SAML Web Browser SSO Profile to help you better understand the contents that follow.

Recommended Reading

A recommended reading is the SAML technical overview from OASIS, this gives a detailed introduction to the SAML specifications and how to use it.

During your work with OpenSAML, you will probably need to read some of the actual SAML specifications.

Both, the technical overview and the specifications are referenced in the chapter; Further Reading, at the back of the book.

III. SAML in Short

SAML is an XML framework for exchanging security information and defines the communications protocols and format for doing this in a secure manner.

SAML is a centralized authentication scheme and defines two main entities that are communicating:

- The service provider (SP) is the entity (or entities) that provides an actual business service to the user and is often the one to request authentication of a user.

- The identity provider (IdP) provides authentication of a user, thus ensuring that the user is who he or she claims to be.

Centralized scheme means that the SPs typically rely on one IdP to provide authentication of all users. In contrast, a decentralized authentication scheme, like OpenID; typically allows for many different IdPs, depending on the preference of the user.


SAML SPs can also be setup to use multiple IdPs, but this is generally not the case.

SAML is created to fulfill three main purposes:

1. Single Sign-On

Single Sign-On is to allow a user, with access to multiple separate systems, to sign in once and access all the systems, without having to sign in to every single one. Single Sign-On between different systems in different domains, is the most important goal of the SAML specification.

2. Federated Identity

Federated identity is when business partners in different domains agree to share a common identifier for the identity of the user. When a user is authenticated using SAML, the shared identifier is sent to the service provider and the user can be linked to a local profile.

3. Use of SAML in other Frameworks

The SAML assertion is a very flexible way of describing an authentication. One of the main drivers for SAML is the ability to use the SAML Assertion and its flexibility in other frameworks and contexts that is not part of the original specification. For example, the WS-security framework specifies how to use SAML Assertions to secure web services. Another example is the XACML framework (eXtensible Access Control Markup Language) that is used to communicate authorization information. One of the key goals of XACML is the ability to integrate with SAML.

IV. The SAML Specifications

SAML is built up of many specifications that together form the SAML framework. For the purpose of this book five of them are of interest.

1. Assertions
2. Protocols
3. Bindings
4. Profiles
5. Metadata.

1. *Assertions - The Information*

This is a specification of the SAML Assertion; the actual object that is used to describe an authentication in SAML. An assertion can typically contain information about; when and how a user was authenticated. This can include some extra information about the user, for example; email addresses and telephone numbers. The specification describes, in detail, the content and use of the different XML (Extensible Markup Language) elements used. Below is an example showing an Assertion whose implementation is specified in the assertion specification.

```
<saml:Assertion
    ID="560f83e350ff2cabfa02345ee59153ba"
    IssueInstant="2010-11-22T14:30:30.728Z"
    Version="2.0">
  <saml:Issuer>TestIDP</saml:Issuer>
  <saml:Subject>
    <saml:NameID>
      harold_dt
    </saml:NameID>
  </saml:Subject>
  <saml:AuthnStatement
      AuthnInstant="2010-11-04T14:04:30Z"
      SessionIndex="s22428b07e56ce0dbd3f72237ce29c585541db5d01">
    <saml:AuthnContext>
      <saml:AuthnContextClassRef>
urn:oasis:names:tc:SAML:2.0:ac:classes:PasswordProtectedTransport
      </saml:AuthnContextClassRef>
    </saml:AuthnContext>
  </saml:AuthnStatement>
</saml:Assertion>
```

2. *Protocol -The Communication*

The protocol specification specifies how to perform different actions. These actions are broken up into a series of request and response objects that contain information specific to the action. The creation and interpretation is specified in great detail in this specification. One example of a protocol is the authentication request protocol that specifies; how a service provider can request to have a user authenticated.

Below is an example of AuthnRequest object. This is the first request that is sent when using the authentication request protocol and indicates how the SP wants the user authenticated.

```
<saml2p:AuthnRequest
    AssertionConsumerServiceURL=http://localhost:8080/webprofile-ref-
project/sp/consumer
    Destination="http://localhost:8080/webprofile-ref-
project/idp/singleSignOnService"
    ID="_52c9839568ff2e5a10456dfefaad0555"
    IssueInstant="2014-05-13T17:34:37.810Z"
    ProtocolBinding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-
Artifact"
    Version="2.0">
```

```
  <saml2:Issuer>
    TestSP
  </saml2:Issuer>
  <saml2p:NameID
      PolicyAllowCreate="true"
      Format="urn:oasis:names:tc:SAML:2.0:nameid-format:transient"/>
  <saml2p:RequestedAuthnContext Comparison="minimum">
    <saml2:AuthnContextClassRef>
      urn:oasis:names:tc:SAML:2.0:ac:classes:Password
    </saml2:AuthnContextClassRef>
  </saml2p:RequestedAuthnContext>
</saml2p:AuthnRequest>
```

3. Binding- The Transportation

Binding defines how SAML messages are transported using lower-level communication protocols. Examples of such bindings are; the HTTP Redirect Binding, which specifies how messages are to be transported using HTTP Redirect messages and the SAML SOAP Binding, which specifies how messages are transported using SOAP. Below is an example of SAML message transported inside of a SOAP envelope using the SAML SOAP Binding.

```
<soap-env:Envelope>
  <soap-env:Body>
    <!- SAML message -->
  </soap-env:Body>
</soap-env:Envelope>
```

4. Profiles- Putting it Together

The profiles are the specifications that organize everything together and describe on a higher level how the assertions, protocols and bindings are used to solve a specific case. For example, the Web Browser SSO Profile describes how a user, using a browser, can be authenticated. The Web Browser SSO Profile will be discussed more in the following chapters.

5. Metadata

SAML Metadata is configuration data that contains information about a party in a SAML communication. Among many other things, the metadata can contain the id of the other party, web service end points to be used, supported bindings and cryptographic keys, to be used for communication.

OpenSAML provides metadata providers to help with SAML metadata. These provide great help when reading and processing the configuration data.

It is out of the scope of this book to discuss the structure and parsing of the metadata in detail. However, for more information, refer to the metadata providers in the OpenSAML documentation and the SAML metadata specification.

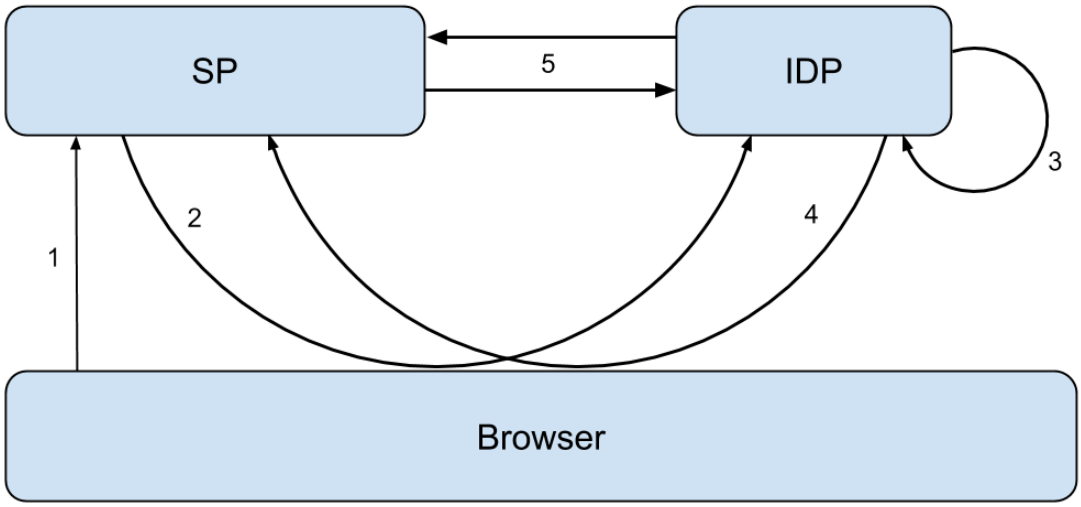
An example of what a SAML metadata XML could look like is provided in the Appendix in the end of the book.

Relevant references to metadata and the complete specification of SAML Metadata are provided in the subsequent chapters of this book.

V. The SAML Web Browser SSO Profile

SAML Web Browser SSO Profile is arguably the most important of the profiles in the SAML specification. It specifies how to solve the use-case where a user with a web browser is authenticated using an external identity provider.

In this book, I will show a usual configuration using the web browser profile with HTTP artifact binding. This is not the only configuration that can be used for the web browser profile, but a common one. The flow in the authentication process is illustrated in the figure below.



1. User Tries to Get Access

The process begins with an unauthenticated user, trying to get access to a protected part of the application (SP). Some form of filter is put in place to check if the user is authenticated or not. This is not really a part of the SAML process but merely a step to decide if authentication is required.

2. The User is Redirected to the Identity Provider (IdP)

When the filter detects a user who is not authenticated, the user is automatically redirected to the IdP for authentication.

3. The User is Authenticated

In this step, the user is authenticated. This step does not involve any interaction with the SP. The IdP has full responsibility for authenticating the user in a secure way.

4. The Authenticated User is Sent Back to the SP

When the authentication is successfully completed, the user is sent back to the SP together with a SAML artifact. The artifact is more or less an identifier for the authentication information, the assertion, at the IdP. This information could contain sensitive information and is therefore not sent via the browser.

5. Request Authentication Information

When the SP receives the artifact, it is sent to a web service at the IdP. The web service returns the authentication information through SAML Artifact Response.

The artifact response contains the SAML Assertion, which is the actual proof of authentication. The assertion can, among other things, contain information on how and when the user was authenticated. The assertion can also contain different identity attributes with information about the user, for example, username, address and phone number.

Part I: Chapter 2

Getting started

This chapter will walk you through the process of setting up your environment and the OpenSAML library to start development. It also presents the sample project that is going to be referred to and used later in the book.

I. Getting the OpenSAML Libraries

The OpenSAML libraries are available from the OpenSAML homepage.

For Maven users the dependencies are provided in the Shibboleth repository: https://build.shibboleth.net/nexus/content/repositories/releases/org/opensaml/

OpenSAML 3 is organized as a Maven multi-module Maven project, with different functionality in its own module. For now users are not provided with a single dependency for the whole of OpenSAML, so each required module must be added as its own dependency. The following modules exist in the latest version of OpenSAML.

- opensaml-core
- opensaml-profile-api
- opensaml-profile-impl
- opensaml-soap-api
- opensaml-soap-impl
- opensaml-saml-api
- opensaml-saml-impl
- opensaml-xacml-api
- opensaml-xacml-impl
- opensaml-xacml-saml-api
- opensaml-xacml-saml-impl
- opensaml-messaging-api
- opensaml-messaging-impl
- opensaml-storage-api
- opensaml-storage-impl
- opensaml-security-api
- opensaml-security-impl
- opensaml-xmlsec-api
- opensaml-xmlsec-impl

This modularity enables the project to only include the required dependencies. Which module are required, will vary from project to project. Below are the required dependency declarations for the latest version for the sample project used throughout the book:

```
<dependencies>
  <dependency>
```

```
    <groupId>org.opensaml</groupId>
    <artifactId>opensaml-core</artifactId>
    <version>3.2.0</version>
  </dependency>
  <dependency>
    <groupId>org.opensaml</groupId>
    <artifactId>opensaml-saml-api</artifactId>
    <version>3.2.0</version>
  </dependency>
  <dependency>
    <groupId>org.opensaml</groupId>
    <artifactId>opensaml-saml-impl</artifactId>
    <version>3.2.0</version>
  </dependency>
  <dependency>
    <groupId>org.opensaml</groupId>
    <artifactId>opensaml-messaging-api</artifactId>
    <version>3.2.0</version>
  </dependency>
  <dependency>
    <groupId>org.opensaml</groupId>
    <artifactId>opensaml-messaging-impl</artifactId>
    <version>3.2.0</version>
  </dependency>
  <dependency>
    <groupId>org.opensaml</groupId>
    <artifactId>opensaml-soap-api</artifactId>
    <version>3.2.0</version>
  </dependency>
  <dependency>
    <groupId>org.opensaml</groupId>
    <artifactId>opensaml-soap-impl</artifactId>
    <version>3.2.0</version>
  </dependency>
 </dependencies>
<repositories>
  <repository>
    <id>Shibboleth repo</id>
    <url>
https://build.shibboleth.net/nexus/content/repositories/releases
    </url>
  </repository>
</repositories>
```

II. Ensuring the correct implementations of JCE

OpenSAML uses JCE for cryptographic functionality. According to the OpenSAML team, some of the implementations on JCE does not support the entire required set of cryptographic functions and it is recommended to use the Bouncy Castle implementation for JCE.

To help confirm that your JCE implementations have everything needed by OpenSAML, the following method is provided:

```
JavaCryptoValidationInitializer javaCryptoValidationInitializer =
    new JavaCryptoValidationInitializer();
javaCryptoValidationInitializer.init();
```

This should be run before the initialization of OpenSAML, described below, to ensure that all the required prerequisites are met.

The following can be used to log the installed JCE providers:

```
for (Provider jceProvider : Security.getProviders()) {
  logger.info(jceProvider.getInfo());
}
```

When using Maven to acquire the OpenSAML dependencies, the Bouncy Castle provider is automatically included in the class path. When downloading OpenSAML manually, these are included in the package. However, they have to be manually added to the class path.

III. Logging

OpenSAML uses the SLF4J logging facade. SLF4J does not have any logging capability by it self, but relies instead on logging implementation to do the actual logging. The OpenSAML developer team uses Logback, but any implementation compatible with SLF4J will suffice.

To enable logging with Logback, one needs the Logback libraries: logback-core and logback-classic and apache commons logging. The libraries can be downloaded from the Logback and Apache homepage:

http://logback.qos.ch/download.html,

https://commons.apache.org/proper/commons-logging/,

or fetched with Maven using the following dependency declaration:

```
<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-core</artifactId>
  <version>1.0.13</version>
</dependency>
<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-classic</artifactId>
  <version>1.0.13</version>
</dependency>
<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-classic</artifactId>
  <version>1.0.13</version>
</dependency>
<dependency>
    <groupId>commons-logging</groupId>
    <artifactId>commons-logging</artifactId>
    <version>1.2</version>
</dependency>
```

IV. The OpenSAML Initialization process

OpenSAML depends on a collection of configuration files to work. OpenSAML ships with a default set of configuration files. These are recommended for most purposes but can be changed if necessary. One example of this, provided by the OpenSAML team, is if you intend to write your own SAML extension.

The configuration files must be loaded before the OpenSAML library can be used. To load the default configuration files, run:

```
InitializationService.initialize();
```

Now the OpenSAML library is ready for use.

If this is not done, or the initialization process could not finish, OpenSAML will not return certain objects, which will result in NullPointerException when trying to use the library. This is a common error for developers just starting out with OpenSAML.

Creating SAML objects

SAML objects are created using builders. The steps for creating an SAML object involve a lot of jumping through hoops, casting and boilerplate code.

This is the way to create a SAML Assertion object using OpenSAML:

```
XMLObjectBuilderFactory builderFactory =
    XMLObjectProviderRegistrySupport.getBuilderFactory();

Assertion assertion = (Assertion) builderFactory
    .getBuilder(Assertion.DEFAULT_ELEMENT_NAME)
    .buildObject(Assertion.DEFAULT_ELEMENT_NAME);
```

To avoid a lot of unnecessary code it can be a good idea to use the following utility method. The method uses generics to allow an easier creation of objects.

```
public static <T> T buildSAMLObject(final Class<T> clazz) {
  XMLObjectBuilderFactory builderFactory =
      XMLObjectProviderRegistrySupport.getBuilderFactory();

  QName defaultElementName = (QName) clazz.getDeclaredField(
      "DEFAULT_ELEMENT_NAME").get(null);
  T object = (T) builderFactory.getBuilder(defaultElementName)
      .buildObject(defaultElementName);
  return object;
}
```

Using this method, the Assertion and other objects can now be created using a single line:

```
OpenSAMLUtils.buildSAMLObject(Assertion.class);
```

VI. Sample Project and Code Samples

Part II and III of this book include detailed examples using a sample project and its code.

Project

The sample project is a web application implementation of a Service Provider. The project also contains skeleton of an Identity Provider to show the communication between the two.

The sample project is publicly available on Bitbucket and can be downloaded using the following Git command:

```
git clone https://bitbucket.org/srasmusson/webprofile-ref-project-v3
```

The project is built using Apache Maven, which will need to be installed before running, and can be started by running the following command in a terminal:

```
mvn tomcat:run
```

When the embedded tomcat container has started up, you can se the following message in the terminal

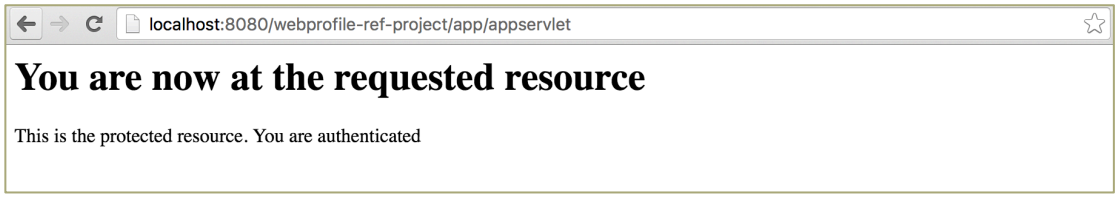
```
INFO: Starting Coyote HTTP/1.1 on http-8080
```

The application is now ready to use and you can open a web browser and browse the URL:

http://localhost:8080/webprofile-ref-project/app/appservlet.

The “appservlet”, on the above URL, is a web page that is protected by the Service Provider. Browsing to this protected webpage for the first time will start the authentication process and redirect the user for authentication.

The user authenticates by clicking the “Authenticate”-button.



When the authentication is done, the user is redirected back to the “appservlet” page and is now allowed access.

All the relevant SAML XML is logged in the window where the startup command was executed.

Use of the Sample Project

The sole purpose of the sample project is to illustrate the function of OpenSAML. Although, it can be a very good place to start when experimenting with OpenSAML, it should not be used for production purposes.

Code Samples

The supporting content features systematic illustrations of the project code for better understanding. Be prepared to come across a lot of code in these parts of the book. The information is customized to make it readable for the reader in several ways:

- Wherever the code extends the desirable length, it is divided into smaller sections of a couple of lines each.
- The code is supported with additional subtexts; explaining certain parts of the code, whenever necessary.
- When the code is easy to infer, it is usually preceded by the description.
- To sustain the attention of the reader, only selected lines of the code in the project are included that need explanation.

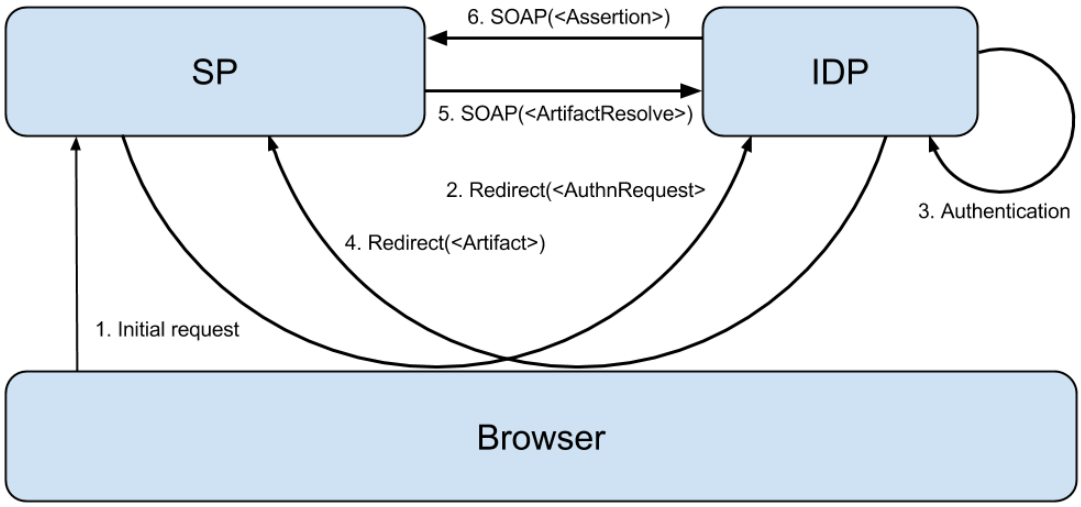
The entire code can be found at Bitbucket:

https://bitbucket.org/srasmusson/webprofile-ref-project-v3

PART II

Chapter 1: The Authentication Process

As mentioned in Heading V of Part I: Chapter 1, authentication with SAML is done in five major steps described below. Please note that this is just a technical overview of the process. In the following chapters, we will look more closely on how to accomplish these steps using OpenSAML.



1. User tries to get access

The user makes a request to a resource on the Service Provider(SP). In this step, the decision is made if an authentication is required or not. Generally authentication is considered required if the user does not already have an authenticated session on the SP.

2. The user is redirected to the Identity Provider (IdP)

If authentication is required, the SP creates an AuthnRequest object that specifies the requirements for how the user should be authenticated. The AuthnRequest is encoded as URL parameter and sent to the IdP with a HTTP Redirect via the browser.

3. The user is authenticated

The IdP decodes the AuthnRequest and authenticates the user based on the requirements in the request.

4. Authenticated user is sent back to the SP

If the authentication is successful, the IdP links this authentication information with a unique identifier called an SAML Artifact. The Artifact is encoded as a URL parameter and the user and artifact is sent back to the SP using HTTP Redirect.

5. SP Requests authentication information

The SP creates an ArtifactResolve object and packages the Artifact inside. The ArtifactResolve is sent to the IdP using a SOAP web service call.

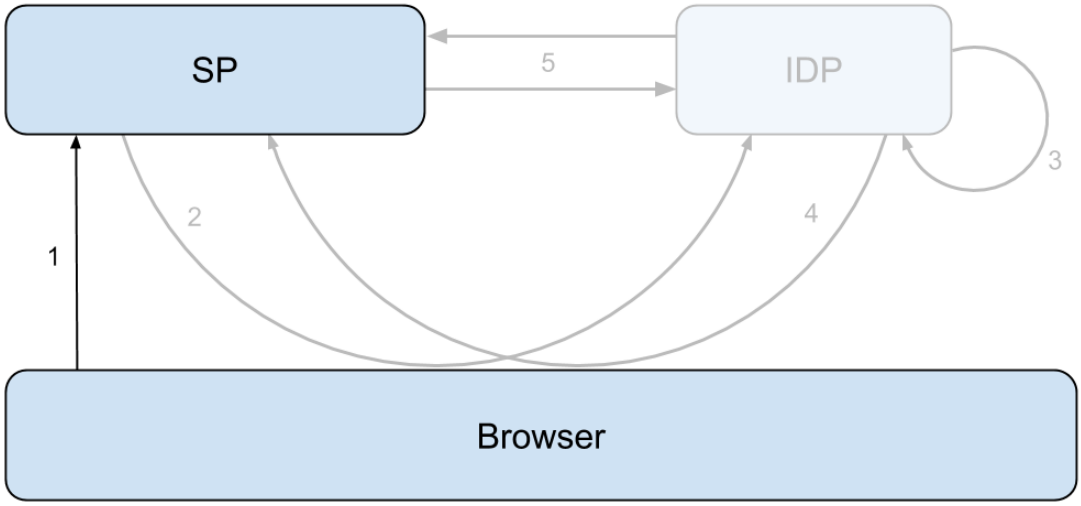
6. IdP responds with authentication information

The IdP receives the ArtifactResolve and extracts the Artifact. The IdP responds to the SOAP request with an ArtifactResponse including the authentication information, in the form of a SAML Assertion contained inside it.

Examples of the XML messages sent in the authentication process are provided in the Appendix in the end of the book

Part II: Chapter 2

Step 1: User Interception



This chapter covers step 1 in the authentication process. User interception is not really a part of the authentication process, but is used to decide if the authentication is needed.

The user's interaction starts with the interception of an unauthenticated user trying to access a resource. In a java web application, a good place to do this interception is in a servlet filter. The filter checks if the user has an authenticated session. If it does, it lets the user through to the resource. However, if it does not, it starts the authentication process. Below is an example of a typical interception filter:

```
public class AccessFilter implements Filter {
  private static Logger logger = LoggerFactory
      .getLogger(AccessFilter.class);

  @Override
  public void init(FilterConfig filterConfig) throws ServletException
{
    JavaCryptoValidationInitializer javaCryptoValidationInitializer =
        new JavaCryptoValidationInitializer();


    try {
      javaCryptoValidationInitializer.init();
    } catch (InitializationException e) {
      e.printStackTrace();
    }

    }

  try {
    InitializationService.initialize();
```

```
  } catch (InitializationException e) {
    new RuntimeException("Initialization failed");
  }
}
```

As the filter is the first class that is run in the authentication, this is a good place to run the initialization process for OpenSAML.

```
public void doFilter(ServletRequest request,
    ServletResponse response, FilterChain chain) throws IOException,
    ServletException {
  HttpServletRequest httpServletRequest =
      (HttpServletRequest) request;
  HttpServletResponse httpServletResponse =
      (HttpServletResponse) response;

  if (httpServletRequest.getSession().getAttribute(
      SPConstants.AUTHENTICATED_SESSION_ATTRIBUTE) != null) {
    chain.doFilter(request, response);
```

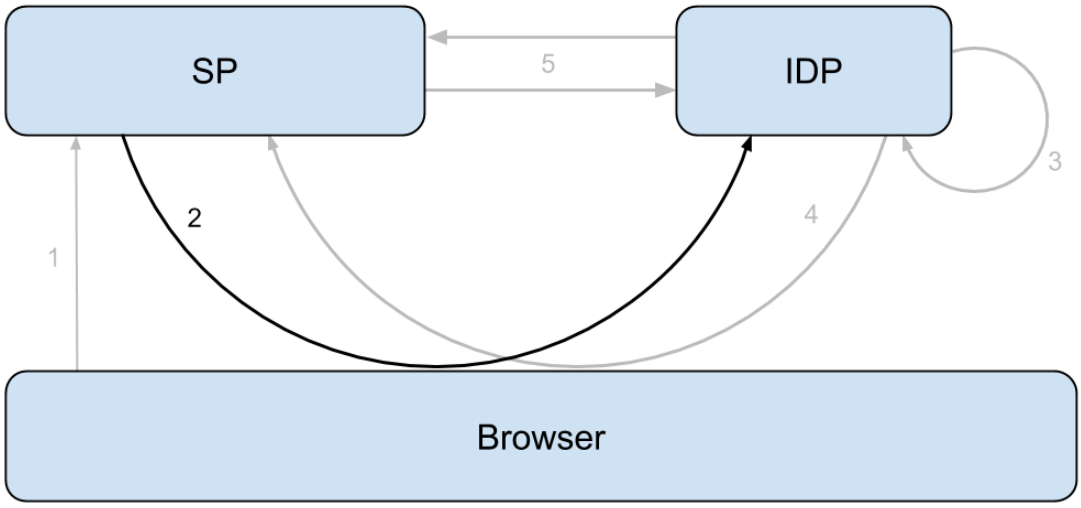
When a user is authenticated, the attribute AUTHENTICATED_SESSION_ATTRIBUTE is set on the session to indicate that the user is authenticated. If the attribute is present the user is already authenticated and the filter takes no further action.

```
  } else {
    setGotoURLOnSession(httpServletRequest);
    redirectUserForAuthentication(httpServletResponse);
  }
```

If the attribute is not present, the user is not yet authenticated. The filter then saves the target URL to the user session and starts the authentication of the user.

Part II: Chapter 3

Step 2: Authentication Request



This chapter covers the actual start of the SAML authentication process. It begins with the SP requesting a user to be authenticated by sending an SAML AuthnRequest object to the IdP. There are different ways to do this; however, in this book we will cover a common way for this purpose. This method utilizes the SAML HTTP Redirect binding to redirect the user to the IdP along with an AuthnRequest as a URL parameter.

I. Building the AuthnRequest object

To build the authentication request, start with instantiating the AuthnRequest object using the helper function presented in Heading V of Chapter 2. This will be the root object of the request.

```
AuthnRequest authnRequest = OpenSAMLUtils
    .buildSAMLObject(AuthnRequest.class);
```

Next the properties of the request are set.

Time of the Request: The receiving end checks the time when the request was created to ensure that the request is not too old.

```
authnRequest.setIssueInstant(new DateTime());
```

The destination URL: This is controlled at the receiver to ensure that the request was actually meant for it.

```
authnRequest.setDestination(getIPDDestination());
```

The binding required to transmit the resulting SAML Assertion: Bindings are briefly explained in Heading IV of Part I: Chapter 1. The artifact binding works by the IdP sending an Artifact to the SP as a URL parameter in a HTTP Redirect. The Artifact is simply put an identifier for the real message that the IdP wants to send. When the SP receives the Artifact, it exchanges it for the real message over a direct channel with the IdP. The direct channel in this case is a SOAP web service. This process is explained in more detail in Heading II of Part II: Chapter 4

```
authnRequest
    .setProtocolBinding(SAMLConstants.SAML2_ARTIFACT_BINDING_URI);
```

SP Address: This is the address at the SP that receives the SAML Assertion, after a successful authentication at the IdP. In our case, this is where the user will be redirected after authenticating.

```
authnRequest
    .setAssertionConsumerServiceURL(getAssertionConsumerEndpoint());
```

Setting ID of the request: This is used in general to identify the request and it plays a role validating the response for the request. This is generally a secure random number.

```
authnRequest.setID(OpenSAMLUtils.generateSecureRandomId());
```

A random secure identifier can be generated using a RandomIdentifierGenerationStrategy. En example of the usage of the generator is shown in below.

```
RandomIdentifierGenerationStrategy secureRandomIdGenerator
    = new RandomIdentifierGenerationStrategy();
String id = secureRandomIdGenerator.generateIdentifier();
```

Issuer: Issuer is a common object that is present in every SAML request and response. This is the identification of the sender and in this case, it only contains the entity ID of the sending SP. The entity ID is an identifying string for the SP. It can be set to any string but it is often practical to use the URL of the SP. For example: http://www.example.com/sp

```
Issuer issuer = OpenSAMLUtils.buildSAMLObject(Issuer.class);
issuer.setValue(SPConstants.SP_ENTITY_ID);
authnRequest.setIssuer(issuer);
```

NameID: The NameID is the IdP identifier for the user. The NameID policy is a specification from the SP on how it wants the NameID to be created. The format indicates what type of identifier the SP wants for the user. The flag AllowCreate indicates if the receiving IdP is allowed to create a user account if one does not already exist.

```
NameIDPolicy nameIDPolicy =
    OpenSAMLUtils.buildSAMLObject(NameIDPolicy.class);

nameIDPolicy.setFormat(NameIDType.TRANSIENT);
nameIDPolicy.setAllowCreate(true);
authnRequest.setNameIDPolicy(nameIDPolicy);
```

NameID Formats

There are numerous formats defined in the SAML specification such as Kerberos, email and Windows Domain Qualified Name. Two special formats are:

- Persistent and
- transient identifier.

Persistent Identifier: The persistent identifier is a random identifier that is linked to a user at the IdP. This is a mechanism to ensure that the real identity of the user is not disclosed. Whenever a user is signed in and the persistent identifier is used, the same identifier is returned. The SP can use this identifier to link the user to a local user account.

Transient Identifier: The transient identifier is a random identifier that does not have any connection to the user. A transient identifier will be different for every time the user signs in.

```
authnRequest.setRequestedAuthnContext(buildRequestedAuthnContext());
```

The requested authentication context is the SP's requirement for the authentication, which includes; how the SP wants the IdP to authenticate the user.
Below is an explanation of the authentication context used in this case:

```
RequestedAuthnContext requestedAuthnContext = OpenSAMLUtils
    .buildSAMLObject(RequestedAuthnContext.class);
```

The context object is instantiated.

```
AuthnContextClassRef passwordAuthnContextClassRef = OpenSAMLUtils
    .buildSAMLObject(AuthnContextClassRef.class);

passwordAuthnContextClassRef
    .setAuthnContextClassRef(AuthnContext.PASSWORD_AUTHN_CTX);

requestedAuthnContext.getAuthnContextClassRefs().add(
    passwordAuthnContextClassRef);

requestedAuthnContext
   .setComparison(AuthnContextComparisonTypeEnumeration.MINIMUM);
```

The authnContextClassRef refers to one authentication alternative. Examples of other authentication alternatives are; password authentication and Kerberos. For

a complete reference of the available authentication contexts, refer to the SAML authentication context document found in the chapter on Further Reading in the back of this book.

One or more authentication contexts can be defined. In cases where there is more than one, the list of contexts is based on the highest to lowest authentication preference.

The comparison tells the IdP how to evaluate the user based on the authentication alternatives given. The comparison can be on the basis of one of the following:

- **Minimum**

If minimum is set as comparison, the user is authenticated using a method that is equal to, or greater than any of the authentication contexts specified. In our case, this would mean that the user could be authenticated using password or a method considered equally strong or stronger, for example, smart card authentication.

- **Better**

If better is set as comparison the user must be authenticated with a method that is considered stronger than any of the specified authentication contexts. In our case, this would mean that the user must be authenticated using a method that is considered stronger than password.

- **Exact**

If the comparison is set to exact, the user must be authenticated using a method that corresponds to exactly one of the specified authentication contexts. In our case, this would mean that the user must be authenticated using password, nothing else.

- **Maximum**

If maximum is set as comparison, the user must be authenticated with a method that is as strong as possible, without exceeding any of the specified authentication contexts. In our case, this would mean that the user must be authenticated using password or a weaker method like IP address comparison.

It is up to the IdP implementation to rate how strong an authentication method is compared to another one.

II. Sending the AuthnRequest

To send the authentication request to the IdP, HTTP Redirect binding is used.

When using the HTTP Redirect binding, the AuthnRequest message is transmitted as deflate encoded URL parameter. It is not a requirement that the message be signed, but it can be done in order to maintain message integrity. In the case that the message is signed and the signature is attached as a separate URL parameter. As the message is transmitted as a URL parameter through the

browser, it is open for manipulation. It is recommended that the redirect be done over HTTPS to ensure confidentiality and integrity of the message.

To help with signing, deflating and sending the redirect, OpenSAML provides the HTTPRedirectDefalteEncoder. Below, I will show how to use the encoder to deflate and sign the AuthnRequest and redirect user to the IdP with the message.

The OpenSAML message encoders are an abstraction for the transportation of SAML messages. In the case of HTTPRedirectDefalteEncoder, it abstracts the details of the HTTP Redirect binding.

The encoders work with a data object called a MessageContext, where all the information about the message and the transportation details of the message are saved.

Message Contexts

When sending a message back in version 2 of OpenSAML, a BasicSAMLMessageContext was used to contain all the information about the SAML message, where it was going to be sent, signing information etc. In the current version of OpenSAML, however, the MessageContext structure has been split up into many separate classes, each containing a specific part of information. Below are some examples.

- MessageContext - Main context
- SAMLPeerEntityContext - A context containing information about a peer entity, in other words, the IdP for the SP and vice versa. This context itself does not contain much information, but usually contain one or more sub-context
- SAMLEndpointContext - This context contains information about a specific endpoint of the peer entity.
- SecurityParametersContext – The context is used for holding information about the signature and/or encryption.
- SAMLMessageInfoContext – The context is used for holding basic info about issue instant and id.

The contexts are created as sub-contexts of the main message context or each other. The following is and example:

```
MessageContext context = new MessageContext();

SAMLPeerEntityContext peerEntityContext
    = context.getSubcontext(SAMLPeerEntityContext.class, true);

SAMLEndpointContext endpointContext
    = peerEntityContext.getSubcontext(SAMLEndpointContext.class, true);
```

The last parameter of the getSubcontext method is used to instruct the method to create the sub-context if it doesn't already exist. Once the context is created it is populated using the available setters, as shown below.

```
endpointContext.setEndpoint(getIPDEndpoint());
```

The SAMLEndpointContext message context is required to send an AuthnRequest and is used to specify the destination of the message. SAMLEndpointContext is a sub-context of SAMLPeerEntityContext and is created and populated as shown below.

At first, the main message context is created.

```
MessageContext context = new MessageContext();
```

Secondly, set the SAML message to be sent on the main context. In this case the AuthnRequest.

```
context.setMessage(authnRequest);
```

Thirdly, the SAMLPeerEntityContext and SAMLEndpointContext are created.

```
SAMLPeerEntityContext peerEntityContext
    = context.getSubcontext(SAMLPeerEntityContext.class, true);
SAMLEndpointContext endpointContext
    = peerEntityContext.getSubcontext(SAMLEndpointContext.class,
        true);
```

Fourthly, the destination endpoint is created and set as destination on the SAMLEndpointContext.

```
SingleSignOnService endpoint
    = OpenSAMLUtils.buildSAMLObject(SingleSignOnService.class);

endpoint.setBinding(SAMLConstants.SAML2_REDIRECT_BINDING_URI);
endpoint.setLocation(getIPDSSODestination());
context.setPeerEntityEndpoint(endpoint);

endpointContext.setEndpoint(endpoint);
```

Fifthly, the SecurityParametersContext is optional. However here it is created and populated with singing parameters to provide a signature to the message.

```
SignatureSigningParameters signatureSigningParameters
    = new SignatureSigningParameters();
signatureSigningParameters.setSigningCredential(
    SPCredentials.getCredential());
signatureSigningParameters.setSignatureAlgorithm(
    SignatureConstants.ALGO_ID_SIGNATURE_RSA_SHA256);

context.getSubcontext(
    SecurityParametersContext.class, true)
        .setSignatureSigningParameters(signatureSigningParameters);
```

While preparing the message for transport, the HTTPRedirectDefalteEncoder will automatically sign the AuthnRequest and add the signature and signature algorithm as URL parameters, along with the AuthnRequest itself. More information on the cryptographic signature functions and credentials of OpenSAML is found in Part III, Chapters 1 and 2.

Sixthly, the HTTPRedirectDefalateEncoder is created and populated with the message context and the HTTPServletResponse where the message will be encoded.

```
HTTPRedirectDeflateEncoder encoder
    = new HTTPRedirectDeflateEncoder();

encoder.setMessageContext(context);
encoder.setHttpServletResponse(httpServletResponse);
```

Lastly, the encoder is initialized and the encode method is called to send the message. The encode method deflates the message, generates a signature, adds the result as URL parameters and redirects the user to the IdP.

```
encoder.initialize();
encoder.encode();
```

An example of the resulting redirect URL and the AuthnRequest XML is shown below.

```
http://localhost:8080/webprofile-ref-project/idp/singleSignOnService
    ?SAMLRequest=<The deflated AuthnRequest>
    &SigAlg=http%3A%2F%2Fwww.w3.org%2F2000%2F09%2Fxmldsig%23rsa-sha1
    &Signature=<The signature>
```

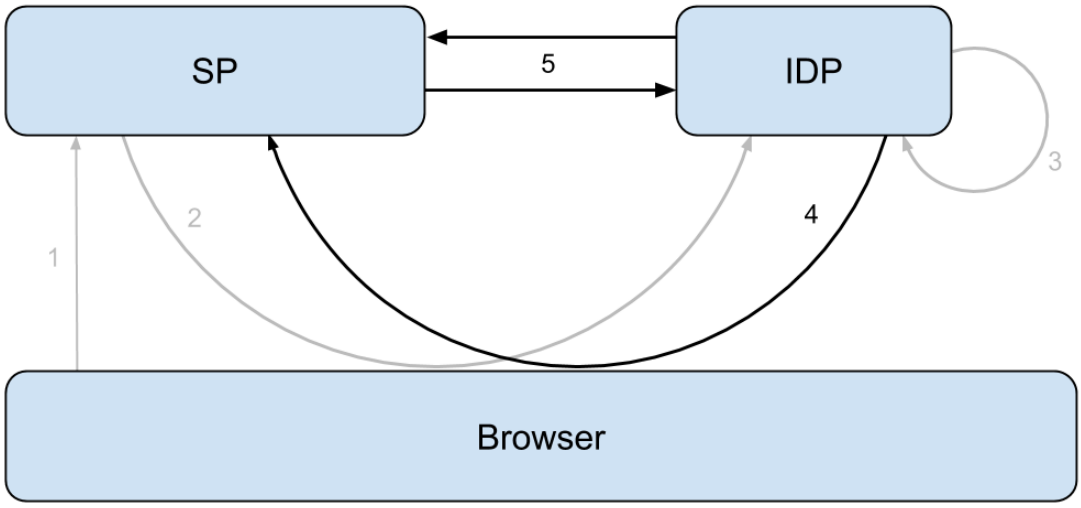
```
<saml2p:AuthnRequest
    AssertionConsumerServiceURL="http://localhost:8080/webprofile-
ref-project/sp/consumer"
    Destination="http://localhost:8080/webprofile-ref-
project/idp/singleSignOnService"
    ID="_2d2962422c817f8ac1ec4ac5a696908c"
    IssueInstant="2014-07-24T17:58:02.804Z"
    ProtocolBinding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-
Artifact"
    Version="2.0">

  <saml2:Issuer>
    TestSP
  </saml2:Issuer>

  <saml2p:NameIDPolicy
    AllowCreate="true"
    Format="urn:oasis:names:tc:SAML:2.0:nameid-format:transient" />
  <saml2p:RequestedAuthnContext Comparison="minimum">
    <saml2:AuthnContextClassRef>
        urn:oasis:names:tc:SAML:2.0:ac:classes:Password
    </saml2:AuthnContextClassRef>
  </saml2p:RequestedAuthnContext>
</saml2p:AuthnRequest>
```

Part II: Chapter 4

Step 4 and 5: The Artifact and Artifact Resolution



This chapter covers partly step 4 and step 5 in the authentication process. Step 3 is not covered in this book as this step is completely handled at the IdP and the scope of this book is restricted to the interactions of the SP.

In step 3, the IdP authenticates the user. The IdP ensures that the user is who he or she claims to be. In other words, the identity of the user is confirmed. The method used for authentication is based on the criteria specified in the requested authentication context (in the AuthnRequest message) that was attached to the user when sent to the IdP. However, the decision on how to authenticate the user is ultimately the IdPs.

A note on Single Sign-On

Here is where the SSO happens. If the IdP has previously authenticated the user, the IdP will not perform step 3. The IdP will just see that the user has a session and consider the user authenticated. Notice that if the user is authenticated or was previously authenticated is totally transparent to the SP. The SP just gets a user with an assured identity.

If the authentication of the user is successful in step 3, the IdP links this authentication information with a unique identifier called an artifact. The artifact is encoded as a URL parameter and the user and artifact is sent back to the SP using HTTP redirect in step 4.

To get the authentication information the SP creates an ArtifactResolve object and packages the Artifact inside. The ArtifactResolve is sent to the IdP using a SOAP web service call and the IdP responds with an ArtifactResolveResponse containing the SAML Assertion, the authentication information.

It is the responsibility of the SP to validate the artifact. Therefore, the content and function of the artifact is explained in detail in the section below.

I. The Artifact

The artifact is a reference to a SAML object. An artifact can use different formats but the SAML specification defines a general format that all must follow. The format is specified and explained below:

Part name	Position
TypeCode	1 - 2 Bytes
EndpointIndex	3 - 4 Bytes
Remaining artifact	5 -… Bytes

- The type code identifies the type of the current artifact; different types of artifacts can specify a stricter format for the remaining artifact.
- The endpoint index is a value identifying the web service endpoint defined in metadata that should be used to exchange the artifact against the actual SAML object it references.
- The artifact is the Base64 encoded concatenation of these parts.
- There are recommendations for how to construct the remaining artifact and the SAML specification defines a type that satisfies these recommendations, type 4.
- Type 4 specifies that the remaining artifact be built up of two parts. The whole format for a type 4 artifact looks like the following:

Part name	Position
TypeCode	1 - 2 Bytes
EndpointIndex	3 - 4 Bytes
SourceID	5 - 24 Bytes
MessageHandle	25 - 44 Bytes

- SourceID should be a SHA-1 hash of the entity id of the sender.
- MessageHandle is the actual reference to the SAML message at the sender and should be a random identifier of at least 16 bytes. If the identifier is less than 20 bytes, it should be padded to the length of 20 bytes.

II. The Artifact Binding

The artifact binding is a way to transmit SAML messages, by reference, through an HTTP user agent, for example a browser. For the transmission of the reference, the artifact, two different bindings can be used, HTTP Post or HTTP Redirect. The artifact is then exchanged for the actual SAML message using a direct channel between the SP and IdP; for example a SOAP channel. The artifact binding is used, if for some reason, there is a problem associated with transmitting the entire SAML message via the browser. Reasons for this could be, for example, technical limitations or security reasons. In this case, HTTP artifact binding with HTTP redirect is used for transmission of the SAML Assertion. This means the artifact is transmitted as a URL encoded, URL parameter.

The Artifact binding is not the only method that can be used to transmit an Assertion, but it is the most complex one and therefore warrants a thorough explanation in this book.

III. Building ArtifactResolve

The code below shows how to build and populate a SAML ArtifactResolve object to later be sent to the IdP.

The first thing to do is to get the artifact from the URL parameter.

```
Artifact artifact = OpenSAMLUtils.buildSAMLObject(Artifact.class);
artifact.setArtifact(httpServletRequest.getParameter("SAMLart"));
```

The ArtifactResolve object is instantiated using the utility method presented in Heading V of Part I: Chapter 2.

```
ArtifactResolve artifactResolve = OpenSAMLUtils
    .buildSAMLObject(ArtifactResolve.class);
```

Next the properties of the request are set.

Issuer: Issuer is a common object that is present in every SAML request and response. This is the identification of the sender and in this case, it only contains the entity ID of the sending SP. The entity ID is an identifying string for the SP. It can be set to any string but it is often practical to use the URL of the SP. For example: http://www.example.com/sp

```
Issuer issuer = OpenSAMLUtils.buildSAMLObject(Issuer.class);
issuer.setValue(SPConstants.SP_ENTITY_ID);
authnRequest.setIssuer(issuer);
```

Time of the Request: The time when the request was created. The receiving end checks this to ensure that the request is not too old.

```
artifactResolve.setIssueInstant(new DateTime());
```

Setting ID of the request: This is used in general to identify the request and it plays a role validating the response for the request. This is generally a secure random number.

```
artifactResolve.setID(OpenSAMLUtils.generateSecureRandomId());
```

The destination URL: This is controlled at the receiver to ensure that the request was actually meant for it.

```
artifactResolve.setDestination(getIPDArtifactResloveDestination());
```

Finally the artifact is added to the request.

```
artifactResolve.setArtifact(artifact);
```

IV. Sending ArtifactResolve Using SOAP

The code below shows how to send the SAML ArtifactResolve object, which was created in the previous section, to the IdP.

As covered under Heading II of Part II: Chapter 3, information about a SAML message is stored in a message context. This is also the case when sending a message using the SOAP client.

First the main message context is created

```
MessageContext<ArtifactResolve> contextout
    = new MessageContext<ArtifactResolve>();

contextout.setMessage(artifactResolve);
```

There are no mandatory contexts needed for sending SOAP messages, but as it is often used the security parameters context is included to provide information needed for signing the message.

```
SignatureSigningParameters signatureSigningParameters
    = new SignatureSigningParameters();
signatureSigningParameters.setSignatureAlgorithm(
    SignatureConstants.ALGO_ID_SIGNATURE_RSA_SHA256);
signatureSigningParameters.setSigningCredential(
    SPCredentials.getCredential());
signatureSigningParameters.setSignatureCanonicalizationAlgorithm(
    SignatureConstants.ALGO_ID_C14N_EXCL_OMIT_COMMENTS);

SecurityParametersContext securityParametersContext
    = contextout.getSubcontext(SecurityParametersContext.class,
        true);
```

```
securityParametersContext.setSignatureSigningParameters(
    signatureSigningParameters);
```

As this operation is a two-way communication, an InOutOperationContext is used to hold the outgoing and incoming message context. The InOutOperationContext is created and the outgoing context is added.

```
InOutOperationContext<ArtifactResponse, ArtifactResolve> context
    = new ProfileRequestContext<ArtifactResponse, ArtifactResolve>();
context.setOutboundMessageContext(contextout);
```

In order to send the SOAP message, a soap client is needed. The client invokes message handlers, encoder and decoder in order to transform and send the message. The handlers, encoder and decoder are contained inside of a pipeline. More about message handlers can be read in Heading V of this chapter.

The way that a client is created is by extending AbstractPipelineHttpSOAPClient and implementing the newPipline method. This method should return the pipeline that should be used by the client when sending the message.

```
AbstractPipelineHttpSOAPClient<SAMLObject, SAMLObject> soapClient
    = new AbstractPipelineHttpSOAPClient() {
        protected HttpClientMessagePipeline newPipeline() throws
            SOAPException {
```

Create the encoder and decoder used to transport the message

```
HttpClientRequestSOAP11Encoder encoder
    = new HttpClientRequestSOAP11Encoder();
HttpClientResponseSOAP11Decoder decoder
    = new HttpClientResponseSOAP11Decoder();
```

Create the pipeline

```
BasicHttpClientMessagePipeline pipeline
    = new BasicHttpClientMessagePipeline(
        encoder,
        decoder
);
```

The SAMLOutboundProtocolMessageSigningHandler is set as an outbound message handler to sign the outgoing message.

```
pipeline.setOutboundPayloadHandler(
    new SAMLOutboundProtocolMessageSigningHandler());
```

A HttpClient is provided to the SOAP client to facilitate encoding and decoding to HTTP.

```
HttpClientBuilder clientBuilder = new HttpClientBuilder();
soapClient.setHttpClient(clientBuilder.buildClient());
```

Finally the SAML message is sent using the clients send method, specifying the destination endpoint and the message context

```
soapClient.send(IDPConstants.ARTIFACT_RESOLUTION_SERVICE, context);
```

The method waits until the message is delivered and the response is received or a timeout is reached. When the response is received the SAML message can be retrieved from the incoming message context.

```
return context.getInboundMessageContext().getMessage();
```

An example of an ArtifactResolve and ArtifactResponse is provided in the Appendix in the end of the book.

V. Processing SAML Messages using Message Handlers

An exciting new feature in the latest version of OpenSAML is a collection of message handlers. The message handlers process the messages, and can provide among other things, message validation, signature verification and signing. This is done usually before an encoder or decoder encodes or decodes the message onto a channel. Below are some of the handlers available.

- MessageLifetimeSecurityHandler – message lifetime validation
- SAMLOutboundProtocolMessageSigningHandler – signing of outbound messages
- ReceivedEndpointSecurityHandler – validation of message destination

When invoking the handlers they use context information needed by that specific handler. Using the above handlers as an example:

- MessageLifetimeSecurityHandler will require SAMLMessageInfoContext, containing information about issue time etc.
- SAMLOutboundProtocolMessageSigningHandler will require SecurityParametersContext, containing singing parameters.
- ReceivedEndpointSecurityHandler only needs the base message context, containing the SAML message. The required information is extracted directly from the message.

Message handlers can be invoked either directly as shown below.

```
SAMLOutboundProtocolMessageSigningHandler handler
    = new SAMLOutboundProtocolMessageSigningHandler();
handler.initialize();
handler.invoke(context);
```

This can also be done indirectly by a client implementation, for instance with PipelineHttpSOAPClient, when sending SOAP messages. Example of this is shown in Heading IV of this chapter.

As is often the case, there is usually more than one handler that should be used on a message. Instead of initializing and invoking each handler they can be invoked one after the other using a BasicMessageHandlerChain.

```
List handlers = new ArrayList<MessageHandler>();
handlers.add(handler1);
handlers.add(handler2);

BasicMessageHandlerChain<ArtifactResponse> handlerChain
    = new BasicMessageHandlerChain<ArtifactResponse>();
handlerChain.setHandlers(handlers);
```

Below is an example of message handler usage for validating the destination endpoint and the lifetime of the message.

The main message context is created and the message is added as shown below.

```
MessageContext context
    = new MessageContext<ArtifactResponse>();
context.setMessage(artifactResponse);
```

The SAMLMessageInfoContext is populated with general information about the message, in this case the issue instant as depicted below.

```
SAMLMessageInfoContext messageInfoContext
    = context.getSubcontext(SAMLMessageInfoContext.class, true);
messageInfoContext.setMessageIssueInstant(
    artifactResponse.getIssueInstant());
```

The MessageLifetimeSecurityHandler used for validation of message lifetime is populated with the application requirements for message lifetime as shown here.

```
MessageLifetimeSecurityHandler lifetimeSecurityHandler
    = new MessageLifetimeSecurityHandler();
lifetimeSecurityHandler.setClockSkew(1000);
lifetimeSecurityHandler.setMessageLifetime(2000);
lifetimeSecurityHandler.setRequiredRule(true);
```

The ReceivedEndpointSecurityHandler is populated with the servlet request. The handler extracts the required destination from the servlet request object as can be seen below.

```
ReceivedEndpointSecurityHandler receivedEndpointSecurityHandler
    = new ReceivedEndpointSecurityHandler();
receivedEndpointSecurityHandler.setHttpServletRequest(request);
```

The handlers are added to a handler chain, as shown below.

```
List handlers = new ArrayList<MessageHandler>();
handlers.add(lifetimeSecurityHandler);
handlers.add(receivedEndpointSecurityHandler);

BasicMessageHandlerChain<ArtifactResponse> handlerChain
    = new BasicMessageHandlerChain<ArtifactResponse>();
handlerChain.setHandlers(handlers);
```

Finally, the handler chain is initialized and invoked

```
handlerChain.initialize();
handlerChain.doInvoke(context);
```

Part II: Chapter 5

The Assertion in Short

The SAML Assertion is the essence of the authentication and contains information about the authentication and the user. A complete description of the SAML assertion and its contents is out of scope for this book but below is a very short summary.

The assertion is used to contain security information that the receiver can use to make an access control decision. The security information is contained in objects called assertion statements. There are three types of statements:

- Authentication statements contain information about the authentication of a user, for example, time and method used to ensure identity.
- Attribute statement contains attributes that are associated with the user. For example, username, telephone number and address. Attributes are key value pairs that are defined by the IdP.
- Authorization decision statement confirms if the user is authorized to access a specified resource. The authorization decision statement only provides basic authorization capabilities. For more advanced uses, it is recommended to use XACML, which can be integrated with SAML. Further reading on the XACML standard is referenced from the chapter on Further Reading at the end.

I. Use of Assertion

In this case the Assertion is encrypted and signed when received. Chapter 2 and 3 of Part III shows how to decrypt the Assertion and verify the signature. The code below will assume this has already been done.

There is a lot of information that can be extracted and many uses that can be found for an Assertion. Below is an example on how to extract time and method of authentication and how to extract the user specific attributes.

In this code, it is assumed that the Assertion only contains one AuthnStatement.

The time of user authentication is stated in the AuthnInstant property of the AuthnStatement.

```
logger.info("Authentication instant: "
    + assertion.getAuthnStatements().get(0).getAuthnInstant());
```

The authentication method through which the user was authenticated is stated in the AuthnContextClassRef property of the AuthnContext in the AuthnStatement.

```
logger.info("Authentication method: "
    + assertion.getAuthnStatements().get(0).getAuthnContext()
        .getAuthnContextClassRef().getAuthnContextClassRef());
```

The AttributeStatement can contain many attributes, the code above shows how to loop through and log every attribute with name and value.

```
for (Attribute attribute : assertion.getAttributeStatements()
    .get(0).getAttributes()) {
  logger.info("Attribute name: " + attribute.getName());
  for (XMLObject attributeValue : attribute.getAttributeValues()) {
    logger.info("Attribute value: "
        + ((XSString) attributeValue).getValue());
  }
}
```

An example of an encrypted and decrypted Assertion is provided in the Appendix in the end of the book.

PART III

Chapter 1: Credentials in OpenSAML

When using cryptographic functions in OpenSAML, credentials are used to hold the key or keys. A credential holds cryptographic key information and can hold a symmetric or asymmetric key. In the case of asymmetric keys, the credential can contain either the public key or a public/private key pair.

I. Generating credentials

The Credential can be created manually using the KeySupport utility. For experimental purposes, the KeySupport utility can also be used to generate asymmetric key pairs and symmetric keys. The code below shows how to generate an asymmetric key pair and create a credential object from it.

```
KeyPair keyPair = KeySupport.generateKeyPair("RSA", 1024, null);
return CredentialSupport.getSimpleCredential(keyPair.getPublic(),
    keyPair.getPrivate());
```

II. Reading credentials

The preferred way is, however, to use one of the many CredentialResolvers to create the credential from existing keys. In this book, the keys are stored in a Java keystore.

Below is an example that shows how to create a credential using a KeyStoreCredentialResolver

The KeyStoreCredentialResolver constructor takes a key store and a map with the key alias and the key password.

```
Map<String, String> passwordMap = new HashMap<String, String>();
passwordMap.put(KEY_ENTRY_ID, KEY_STORE_ENTRY_PASSWORD);
KeyStoreCredentialResolver resolver = new KeyStoreCredentialResolver(
    keystore, passwordMap);
```

The credential is resolved by querying the credential resolver with a criterion populated with the key alias.

```
Criterion criterion = new EntityIDCriterion(KEY_ENTRY_ID);
CriteriaSet criteriaSet = new CriteriaSet(criterion);
resolver.resolveSingle(criteriaSet);
```

This is only one of the CredentialResolvers available, example of other resolvers are:

- MetadataCredentialResolver, used to resolve credentials from SAML Metadata,

- FilesystemCredentialResolver, used to resolve credentials from keys stored in files on the file system.

These will not be demonstrated any further, as it out of the scope of this book.

Part III: Chapter 2

Cryptographic Signatures in OpenSAML

A signature is a cryptographic proof of integrity. It is used to prove that a previous signee's piece of data has not been changed since the signing. OpenSAML provides functions to sign and verify signatures on a SAML message.

I. Signing a Message

Every SAML object that implements the SignableXMLObject interface can be signed. A signature is produced in four steps.

1. The Signature object is created and populated with properties for the signature.

```
Signature signature = OpenSAMLUtils
    .buildSAMLObject(Signature.class);
signature.setSigningCredential(credential);
signature
    .setSignatureAlgorithm(SignatureConstants
        .ALGO_ID_SIGNATURE_RSA_SHA1);
signature
    .setCanonicalizationAlgorithm(SignatureConstants
        .ALGO_ID_C14N_EXCL_OMIT_COMMENTS);
```

The credential that is used to produce the signature is set on the signature object. Chapter 1 of Part III explains how to obtain credentials. The algorithm that is used when the signature is computed is set on the signature object.

2. Next, the signature is associated with an object of type SignableXMLObject.

```
signableXMLObject.setSignature(signature);
```

3. Next, the SignableXMLObject is marshalled into a XML object.

```
XMLObjectProviderRegistrySupport
    .getMarshallerFactory()
    .getMarshaller(object)
    .marshall(object);
```

4. The last step to produce the signature is to compute the signature using the Signer utility class.

```
Signer.signObject(signature);
```

In some cases, signing is done by OpenSAML components that abstracts the signing and transport of SAML messages. This is the case with the HTTPRedirectDeflateEncoder in Heading II of Part II: Chapter 3.

II. Verifying a Signature

The methods shown here only perform a cryptographic verification of the signature to match the key against the signature. It does not help to determine if the key itself can be trusted. Trust in a key means; you know that the key belongs to the other party. If you do not already trust the origin of the key, it can be established using trust engines. One way is to use a certificate authority. This book, however, does not cover trust engines.

If the object is not signed when running the validation method, they will throw a NullPointerException. To avoid this, it is a good idea to check and handle this case before performing the validation.

```
if (!assertion.isSigned()) {
    throw new RuntimeException("The SAML Assertion was not signed");
}
```

The first step in the verification of the signature is to validate that the signature conforms to the SAML Signature specification.

```
SAMLSignatureProfileValidator profileValidator
    = new SAMLSignatureProfileValidator();
profileValidator.validate(assertion.getSignature());
```

The next step is to do the actual cryptographic validation where the key is matched against the signature.

```
SignatureValidator sigValidator = new SignatureValidator(credential);
sigValidator.validate(assertion.getSignature());
```

The SignatureValidator takes a cryptographic key as a Credential object. Chapter 1 explains how credentials can be created.

If any of the validation methods fails in validating the signature, a ValidationException will be thrown.

Part III: Chapter 3

Encryption in OpenSAML

Encryption is used to hide data using a symmetric or asymmetric key. When using a symmetric key, the data is encrypted and decrypted using the same key. When using an asymmetric key, a key consists of two keys, one public and one private. The public key can be widely distributed while the private key always stays with the owner. When encryption uses asymmetric keys, the public key is used for encryption and the private is used for decryption.

Asymmetric encryption has its advantages in that the public key can be sent in the open without it having any impact if it is seen. Data can then be encrypted and sent back to the owner of the private key that can decrypt it. In contrast, symmetric keys are always secret and the transport of these needs to be secured.

The disadvantage with asymmetric encryption is that it is much slower than symmetric. Because of this, these two are often used in combination. Asymmetric encryption is used to encrypt the symmetric key that is in turn used to encrypt the actual data. This way the advantages of both mechanisms can be used. This usage is shown below.

I. Encryption

Encryption of SAML objects is done in five steps.

1. First, the parameters for the data encryption are set.

The parameters are contained in the DataEncryptionParameters object. These contain information about how the data will be encrypted using a symmetric key. In this case, the encryption is done using the block AES-128 cipher.

```
DataEncryptionParameters encryptionParameters
    = new DataEncryptionParameters();
encryptionParameters
    .setAlgorithm(EncryptionConstants.ALGO_ID_BLOCKCIPHER_AES128);
```

2. Next, the KeyEncryptionParameters are specified.

These specify how the symmetric key, used to encrypt the data, is encrypted using asymmetric encryption. In this case, the RSA OAEP key transport algorithm is used.

```
KeyEncryptionParameters keyEncryptionParameters
    = new KeyEncryptionParameters();
keyEncryptionParameters.setEncryptionCredential(SPCredentials
    .getCredential());
```

```
keyEncryptionParameters
    .setAlgorithm(EncryptionConstants.ALGO_ID_KEYTRANSPORT_RSAOAEP);
```

3. Next, an Encrypter object is created using the Encryption- and KeyEncryption-parameters

```
Encrypter encrypter = new Encrypter(encryptionParameters,
    keyEncryptionParameters);
```

4. Next, the placement of the encrypted symmetric key is set. Here, the key is set to be placed in the KeyInfo element with the encrypted data

```
encrypter.setKeyPlacement(Encrypter.KeyPlacement.INLINE);
```

5. Lastly, the Encrypter is used to encrypt an object.

The Encrypter provides methods for encrypting Assertion, Attribute, NameID, BaseID, and NewID. Here the encrypter is used to encrypt an Assertion.

```
EncryptedAssertion encryptedAssertion = encrypter.encrypt(assertion);
```

II. Decryption

When decrypting, the encrypted symmetric key produced in the encryption process, must be found and used. As the encryption above uses inline key placement, an InlineEncryptedKeyResolver is used to find the encrypted symmetric key. A StaticKeyInfoCredentialResolver is used to read the private asymmetric key that is used to decrypt the symmetric key. The symmetric key, which is then used to decrypt the data in the last step.

```
StaticKeyInfoCredentialResolver keyInfoCredentialResolver
    = new StaticKeyInfoCredentialResolver(
        SPCredentials.getCredential());

Decrypter decrypter = new Decrypter(null,
    keyInfoCredentialResolver, new InlineEncryptedKeyResolver());
```

The decrypter is created using these objects.

```
decrypter.decrypt(encryptedAssertion);
```

The decrypter is used to decrypt an EncryptedAssertion.

Appendix - XML Samples

Service Provider Metadata XML

```
<md:entitydescriptor
    xmlns:md="urn:oasis:names:tc:SAML:2.0:metadata"
    entityid="TestSP">
  <md:spssodescriptor
      authnrequestssigned="true"
      wantassertionssigned="true"
   protocolsupportenumeration="urn:oasis:names:tc:SAML:2.0:protocol">
    <md:keydescriptor use="signing">
      <ds:keyinfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <ds:x509data>
          <ds:x509certificate>
            ======= Public key for verifying signatures =======
          </ds:x509certificate>
        </ds:x509data>
      </ds:keyinfo>
    </md:keydescriptor>
    <md:keydescriptor use="encryption">
      <ds:keyinfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <ds:x509data>
          <ds:x509certificate>
            ======= Public key for encrypting data =======
          </ds:x509certificate>
        </ds:x509data>
      </ds:keyinfo>
    </md:keydescriptor>
    <md:nameidformat>
      urn:oasis:names:tc:SAML:2.0:nameid-format:transient
    </md:nameidformat>
    <md:assertionconsumerservice
        binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Artifact"
        location="http://localhost:8080/webprofile-ref-
project/sp/consumer"
        index="0">
    </md:assertionconsumerservice>
  </md:spssodescriptor>
</md:entitydescriptor>
```

Authentication Request URL

```
http://localhost:8080/webprofile-ref-project/idp/singleSignOnService
    ?SAMLRequest=<The deflated AuthnRequest>
    &SigAlg=http%3A%2F%2Fwww.w3.org%2F2000%2F09%2Fxmldsig%23rsa-sha1
    &Signature=<The signature>
```

Authentication Request XML

```
<saml2p:AuthnRequest
    xmlns:saml2p="urn:oasis:names:tc:SAML:2.0:protocol"
    AssertionConsumerServiceURL="http://localhost:8080/webprofile-
ref-project/sp/consumer"
    Destination="http://localhost:8080/webprofile-ref-
project/idp/singleSignOnService"
    ID="_2d2962422c817f8ac1ec4ac5a696908c"
    IssueInstant="2014-07-24T17:58:02.804Z"
    ProtocolBinding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-
Artifact"
```

```
    Version="2.0">
  <saml2:Issuer xmlns:saml2="urn:oasis:names:tc:SAML:2.0:assertion">
    TestSP
  </saml2:Issuer>

  <saml2p:NameIDPolicy
    AllowCreate="true"
    Format="urn:oasis:names:tc:SAML:2.0:nameid-format:transient" />
  <saml2p:RequestedAuthnContext Comparison="minimum">
    <saml2:AuthnContextClassRef
xmlns:saml2="urn:oasis:names:tc:SAML:2.0:assertion">
        urn:oasis:names:tc:SAML:2.0:ac:classes:Password
    </saml2:AuthnContextClassRef>
  </saml2p:RequestedAuthnContext>
</saml2p:AuthnRequest>
```

Artifact Resolve Request XML

```
<saml2p:ArtifactResolve
    xmlns:saml2p="urn:oasis:names:tc:SAML:2.0:protocol"
    Destination="http://localhost:8080/webprofile-ref-
project/idp/artifactResolutionService"
    ID="_f640be46f18b5203053b90bf07aa24e5"
    IssueInstant="2014-07-24T17:58:08.698Z"
    Version="2.0">
  <saml2:Issuer xmlns:saml2="urn:oasis:names:tc:SAML:2.0:assertion">
    TestSP
  </saml2:Issuer>
  <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    <ds:SignedInfo>
      <ds:CanonicalizationMethod
          Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
      <ds:SignatureMethod
Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
      <ds:Reference URI="#_f640be46f18b5203053b90bf07aa24e5">
        <ds:Transforms>
          <ds:Transform
             Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-
signature" />
          <ds:Transform
              Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
        </ds:Transforms>
        <ds:DigestMethod
Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
        <ds:DigestValue>2qa1Qx6MFCT33ETLA/QoSYL9SGM=</ds:DigestValue>
      </ds:Reference>
    </ds:SignedInfo>
    <ds:SignatureValue>THE SIGNATURE VALUE</ds:SignatureValue>
  </ds:Signature>
  <saml2p:Artifact>
AAQAAMFbLinlXaCM+FIxiDwGOLAy2T71gbpO7ZhNzAgEANlB90ECfpNEVLg=
  </saml2p:Artifact>
</saml2p:ArtifactResolve>
```

Artifact Resolve Response XML With Encrypted Assertion

```
<saml2p:ArtifactResponse
    xmlns:saml2p="urn:oasis:names:tc:SAML:2.0:protocol"
    Destination="http://localhost:8080/webprofile-ref-
project/sp/consumer"
    ID="_7588e788db1a5a76587427513141bef2"
    InResponseTo="Made up ID"
```

```
    IssueInstant="2014-07-24T18:14:11.943Z"
    Version="2.0">
  <saml2:Issuer xmlns:saml2="urn:oasis:names:tc:SAML:2.0:assertion">
    TestIDP
  </saml2:Issuer>
  <saml2p:Status>
    <saml2p:StatusCode
Value="urn:oasis:names:tc:SAML:2.0:status:Success" />
  </saml2p:Status>
  <saml2p:Response
      Destination="http://localhost:8080/webprofile-ref-
project/sp/consumer"
      ID="_d6528ed9c43e8cae757433c09a786e00"
      IssueInstant="2014-07-24T18:14:11.945Z"
      Version="2.0">
    <saml2:Issuer
xmlns:saml2="urn:oasis:names:tc:SAML:2.0:assertion">
      TestIDP
    </saml2:Issuer>
    <saml2p:Status>
      <saml2p:StatusCode
Value="urn:oasis:names:tc:SAML:2.0:status:Success" />
    </saml2p:Status>
    <saml2:EncryptedAssertion
xmlns:saml2="urn:oasis:names:tc:SAML:2.0:assertion">
      <xenc:EncryptedData
          xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
          Id="_818ba123674bc4c7b5e0e8f81fdbd31f"
          Type="http://www.w3.org/2001/04/xmlenc#Element">
        <xenc:EncryptionMethod
           Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc" />
        <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
          <xenc:EncryptedKey Id="_51b10fa77457f837c84202d5e1ec94c1">
            <xenc:EncryptionMethod
                Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-oaep-
mgf1p">
              <ds:DigestMethod
Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
            </xenc:EncryptionMethod>
            <xenc:CipherData>
              <xenc:CipherValue>THE CIPHER VALUE</xenc:CipherValue>
            </xenc:CipherData>
          </xenc:EncryptedKey>
        </ds:KeyInfo>
        <xenc:CipherData>
          <xenc:CipherValue>THE CIPHER VALUE</xenc:CipherValue>
        </xenc:CipherData>
      </xenc:EncryptedData>
    </saml2:EncryptedAssertion>
  </saml2p:Response>
</saml2p:ArtifactResponse>
```

Decrypted Assertion XML

```
<saml2:Assertion
    xmlns:saml2="urn:oasis:names:tc:SAML:2.0:assertion"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    ID="_a2f9bc546e21ef57dfb5fac7453d53d4"
    IssueInstant="2014-07-24T18:14:11.945Z"
    Version="2.0">
  <saml2:Issuer>TestIDP</saml2:Issuer>
  <ds:Signature
```

```
xmlnsSignaturexmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    <ds:SignedInfo>
      <ds:CanonicalizationMethod
          Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
      <ds:SignatureMethod
Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
      <ds:Reference URI="#_a2f9bc546e21ef57dfb5fac7453d53d4">
        <ds:Transforms>
          <ds:Transform
              Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-
signature" />
          <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-
c14n#">
            <ec:InclusiveNamespaces
                xmlns:ec="http://www.w3.org/2001/10/xml-exc-c14n#"
                PrefixList="xs" />
          </ds:Transform>
        </ds:Transforms>
        <ds:DigestMethod
Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
        <ds:DigestValue>ANuG4qqG3VCm4NYmVviXhp0CDQs=</ds:DigestValue>
      </ds:Reference>
    </ds:SignedInfo>
    <ds:SignatureValue>THE SIGNATURE VALUE</ds:SignatureValue>
  </ds:Signature>
  <saml2:Subject>
    <saml2:NameID
        Format="urn:oasis:names:tc:SAML:2.0:nameid-format:transient"
        NameQualifier="Name qualifier"
        SPNameQualifier="SP name qualifier">
      Some NameID value
    </saml2:NameID>
    <saml2:SubjectConfirmation
Method="urn:oasis:names:tc:SAML:2.0:cm:bearer">
      <saml2:SubjectConfirmationData
          InResponseTo="Made up ID"
          NotBefore="2014-07-22T18:14:11.947Z"
          NotOnOrAfter="2014-07-26T18:14:11.948Z"
          Recipient="http://localhost:8080/webprofile-ref-
project/sp/consumer" />
    </saml2:SubjectConfirmation>
  </saml2:Subject>
  <saml2:Conditions
      NotBefore="2014-07-22T18:14:11.948Z"
      NotOnOrAfter="2014-07-26T18:14:11.948Z">
    <saml2:AudienceRestriction>
      <saml2:Audience>
        http://localhost:8080/webprofile-ref-project/sp/consumer
      </saml2:Audience>
    </saml2:AudienceRestriction>
  </saml2:Conditions>
  <saml2:AttributeStatement>
    <saml2:Attribute Name="username">
      <saml2:AttributeValue
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:type="xs:string">
        bob
      </saml2:AttributeValue>
    </saml2:Attribute>
    <saml2:Attribute Name="telephone">
      <saml2:AttributeValue
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:type="xs:string">
```

```
        999999999
      </saml2:AttributeValue>
    </saml2:Attribute>
  </saml2:AttributeStatement>
  <saml2:AuthnStatement AuthnInstant="2014-07-24T18:14:11.952Z">
    <saml2:AuthnContext>
      <saml2:AuthnContextClassRef>
        urn:oasis:names:tc:SAML:2.0:ac:classes:Smartcard
      </saml2:AuthnContextClassRef>
    </saml2:AuthnContext>
  </saml2:AuthnStatement>
</saml2:Assertion>
```

Further Reading

SAML Technical Overview

http://www.oasis-open.org/committees/download.php/27819/sstc-saml-tech-overview-2.0-cd-02.pdf
The technical overview is a useful document that gives an introduction to the SAML specification.

SAML Specifications

http://saml.xml.org/saml-specifications

The page contains the SAML specifications; these are the specifications referenced in the book.

- Core specification – Specifies assertions and protocols
- Bindings specification – Specifies transport bindings
- Profile specification – Specifies how to use SAML to solve use cases
- Metadata specification – Specifies configuration metadata
- Authentication Context – Specifies possible authentication methods

XACML Specification

https://www.oasis-open.org/committees/xacml/
XACML specifies access control represented in XML.

SAML Products

This is a list of products that can be used to solve common SAML use cases:

- Shibboleth

http://www.internet2.edu/products-services/trust-identity-middleware/shibboleth/
SAML product from Internet2 that uses OpenSAML. Open source.

- OpenAM

http://forgerock.com/products/open-identity-stack/openam/
Formerly OpenSSO, maintained by ForgeRock. Open source.

- Spring Security SAML extension

http://projects.spring.io/spring-security-saml/
Extension for Spring Security that gives Spring the ability to authenticate using SAML.

References

Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0 – Errata Composite. (2014). 2nd ed. [PDF] Available at: http://www.oasis-open.org/committees/download.php/35711/sstc-saml-core-errata-2.0-wd-06-diff.pdf [Accessed 23 Jul. 2014].

Bindings for the OASIS Security Assertion Markup Language (SAML) V2.0 – Errata Composite. (2014). 2nd ed. [PDF] Available at: http://www.oasis-open.org/committees/download.php/35387/sstc-saml-bindings-errata-2.0-wd-05-diff.pdf [Accessed 23 Jul. 2014].

Profiles for the OASIS Security Assertion Markup Language (SAML) V2.0 – Errata Composite. (2014). 2nd ed. [PDF] Available at: https://www.oasis-open.org/committees/download.php/35389/sstc-saml-profiles-errata-2.0-wd-06-diff.pdf [Accessed 23 Jul. 2014].

Security Assertion Markup Language (SAML) V2.0 Technical Overview. (2008). 2nd ed. [PDF] Available at: http://www.oasis-open.org/committees/download.php/27819/sstc-saml-tech-overview-2.0-cd-02.pdf [Accessed 23 Jul. 2014].

Wiki.shibboleth.net, (2014). *OSTwoUserManual - OpenSAML 2.x - Confluence*. [online] Available at: https://wiki.shibboleth.net/confluence/display/OpenSAML/OSTwoUserManual [Accessed 23 Jul. 2014].