Web services can be chaotic without a clear definition of how to use them. This tutorial, Part 5 of the Understanding web services series, explains the concepts behind WS-Policy and related standards, such as WS-SecurityPolicy, which provide a means to specify possible configurations of a web service, and also to enforce defined security and authentication.

Before you start

In this tutorial you learn about Web Services Policy, or WS-Policy. This standard is for developers who want to expose their own services in an environment and use policies that define how these web services are allowed to interact with other clients and services. The term WS-Policy refers to a framework that defines governance, such as the method of transport, and security, such as the level of encryption.

In order to follow along with this tutorial, you should have a basic understanding of SOAP, which you can achieve by reading Part 1 of this series (by extension, you also need a basic understanding of XML). SOAP is programming-language agnostic. However, the samples in this tutorial use Java™ and the Apache Axis2 project, even though the concepts apply to any programming language and environment.

About this series

This tutorial series teaches the basic concepts of web services by following the exploits of the fictional newspaper, Daily Moon, as the staff uses web services to create a workflow system to increase productivity in this competitive environment.

Part 1 starts simply, explaining the basic concepts behind web services and showing you how to use SOAP, the specification that underlies most of what is to come, connecting the classifieds department with the Content Management System.

Part 2 takes things a step further, explaining how to use Web Services Description Language (WSDL) to define the messages that web services produce, enabling the team to more easily create services and the clients that connect to them.
Part 3 finds the team with a number of services in place and a desire to locate them easily. In response, Universal Description, Discovery, and Integration (UDDI) provides a searchable registry of available services as a way to publicize its own services to others.

In Part 4, Rudy, publisher of the Daily Moon, decides that the paper needs to institute better security procedures for web services that access their internal systems.

In this Part 5, WS-Policy, you see the changes the teams need to make in order to access those newly secured services.

Part 6 focuses on interoperability, as services from several different implementations must be accessed from a single system. Part 6 covers the requirements and tests involved in WS-I certification.

Finally, Part 7 shows how to use Business Process Execution Language (WS-BPEL) to create complex applications from individual services.

About this tutorial

In this tutorial, follow along as the Daily Moon newspaper team uses the WS-Policy specification to specify the policies that web services must adhere to in order to use the company's web services.

In the course of this tutorial, you will:

• Learn the syntax of creating policy documents, as well as how to create your own policies, and how to use other, already existing assertions.
• See how you can manipulate Policy documents using mathematical properties of the underlying XML constructs.
• Use Apache Neethi, the incomplete WS-Policy implementation from Axis2 that allows Axis2 users to manipulate and combine policy documents.
• Learn how to use WS-Policy and WSDL together, and learn how to write WSDL that requires ports to adhere to a defined policy.

You will need a few tools to begin.

Prerequisites

Much of this tutorial is conceptual, but in order to follow along with the code that uses Apache Neethi to work with WS-Policy documents, you need the following software installed:

Java 2 Standard Edition version 1.4.2 or higher -- All of these tools are Java-based, as are the services and clients you build in this tutorial.

Apache Neethi -- Apache Neethi is what Axis2 uses to create a run-time representation of policy documents, and to perform normalization, merging, and intersection operations on policy documents.
Overview

Before you get into specifying policies for web services, review the sample scenario up to this point in the series.

The story so far

This series follows the staff of the fictional *Daily Moon* newspaper as it moves many of its everyday operations to a web services-based system. In Part 1, the Classifieds Department learned about SOAP by interacting with the Content Management System, and in Part 2, they created their own service, and defined it using Web Services Description Language (WSDL). Then in part 3, they learned how to interact with a UDDI registry, and also how to find data within one, which ultimately led them to create one for the company to enable other organizations to interact with the *Daily Moon*. Because Rudy insisted that his colleagues Gene and Francis provide a way to prevent unauthorized access to their systems in Part 4, Gene and Francis had to implement WS-Security for their web services.

Now, Rudy just realized that clients accessing their web services are not necessarily using the new web services that he requires. Thus, he needs to define policies for the web services as to how and in what way they can be used. In other words, Rudy wants to enforce that clients accessing the web services of the *Daily Moon* do so in a prescribed manner, ensuring the security that was built into it back in Part 4 of this series. He has got Gene and Francis on the task!

Why use WS-Policy

The Web Services Policy Framework, or WS-Policy, is a specification that allows a web service to have a set of rules that must be met, or *consumed*. Authors of clients that consume web services are then to study the policy information to see whether or not they can adhere to these policies. Thus, a client could not be written to simply access a web service that has a policy that requires all messages be encrypted or signed in a certain way. Nor could a client access a web service having a policy requiring a timestamp and send a message that does not have one. And that is the goal of WS-Policy: to specify policy information that web service consumers must adhere to.

When Gene sits down to implement WS-Policy, he discovers a problem. Axis2 does not yet fully support WS-Policy. The Apache Neethi component, which handles WS-Policy, reads and understands policies, but Axis2 does not provide a way for the service to get at that information...yet.

That is the plan, however. So Gene and Francis decide to proceed as though support is already in place, so that they will be fully versed in the topic and ready to go when support becomes available. Rudy agrees that the time would be well spent, given the importance of WS-Policy to the coming world of SOA he has been reading about.

What a policy looks like

You must be aching to know what a policy looks like! Well, here it is. First you start out with a simple policy document, as shown in Listing 1.
Listing 1. The first policy document you have ever seen

```xml
  <wsp:ExactlyOne>
    <wsp:All>
      <sec:SecurityToken>
        <sec:TokenType>sec:X509v3</sec:TokenType>
      </sec:SecurityToken>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

Essentially, a web service using this policy file has one requirement for the clients transacting with it: a security token of type X509v3 (for more information on security tokens, see Related topics). Next, you will see how having a policy or set of policies reinforces WS-Security.

How WS-Policy reinforces WS-Security

WS-Policy and WS-Security go hand in hand. In fact, there is a WS-SecurityPolicy specification that contains several assertions that can be included in WS-Policy documents, because there are several means and methods for implementing security, and so a policy needs to exist for those types so that they can be included in policies. Creating policies that require certain types of encryption, signing, or requiring a timestamp helps solidify the security of a web service because no incoming message can get a desired response from a secured web service with defined policies unless clients follow the policies the web service gives. Francis loves this idea. What would all that hard work bring forth if there were no way to enforce it? Hooray for WS-Policy!

Excellent! Dive in with Gene and Francis, starting with how to create policy documents.

Constructing WS-Policy documents

You saw in Listing 1 that a policy document is XML just like most other web services documents, and in this section, you can learn how to construct your own by learning the syntax and what assertions are already available to you.

Terminology

Here are a few vocabulary words that all WS-Policy gurus, including Francis and Gene, should know:

Policy
A list of policy alternatives.

Policy alternative
Contains policy assertions. In normal form, a policy contains a list of policy alternatives specified in wsp:All tags, meaning that a client following the policy can choose to follow either of the available policy alternatives. It can choose which one, but it must adhere to all policies with an alternative. You will see more about how this works later in this section.
Policy assertion
Represents a requirement or capability. For example, a policy assertion could require that a certain type of encryption be used in encrypting transmitted data. In normal form, policy assertions are listed within a policy alternative.

Policy assertion type
A class of policy assertions. For example, `<sec:SecurityToken>` in Listing 1 is an example of a policy assertion type.

Policy expression
The XML representation of a policy, as shown in Listing 1, in normal or compact form.

Policy subject
An entity or object to which a policy can be applied.

Policy scope
The set of objects to which a policy can be applied.

Policy attachment
The way policies are associated with one or more policy scopes.

See Related topics for an overview article covering WS-Policy. Now you will see about creating policy documents in normal form.

Normal form
Normal form policy expression is the syntax defined in the WS-Policy specification. The simplest policy document is an empty one:

```xml
<wsp:Policy />
```

This is not very useful, however. Next, expand it while keeping it empty (see Listing 2.).

Listing 2. An empty policy document in normal form

```xml
<wsp:Policy ...>
  <wsp:ExactlyOne>
    <wsp:All>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

The policy document in Listing 2 specifies a policy with two policy alternatives that contain no policy assertions. Now look back at Listing 1, which contains a single policy assertion.

Note that the policy assertions from Listing 1 are not valid, but are shown here as an example. This document contains two policy alternatives: one with a single policy assertion, and the other with two policy assertions. Thus, another newspaper's calling client can pick to adhere to the policy alternative containing the one policy assertion, DummyAssertion1, or the other policy alternative containing two policy assertions, DummyAssertion1 and DummyAssertion2.

Next you see how to refer to defined policy assertions in the WS-SecurityPolicy specification.

WS-SecurityPolicy
The WS-SecurityPolicy specification contains several policy assertions and policy assertion types. There are several assertions specified in the specification, including the following:
SignedParts
   Specifies which parts of a message, such as header, body, and so on, must be signed.

SignedElements
   Specifies which elements within a message must be signed.

EncryptedParts
   Same as SignedParts, except encryption is required.

EncryptedElements
   Same as SignedElements, except encryption is required.

Token assertions
   Several Token assertions are listed in the WS-SecurityPolicy document (see Related topics for a link) that define what type of token to include in messages.

Properties
   Several properties can be set also, such as encryption algorithms, or a timestamp requirement.

Now, Gene and Francis can create a policy document relevant to their web services projects that uses the WS-SecurityPolicy specification.

Building a policy using the WS-SecurityPolicy specification

Now, on to building a policy for the Daily Moon. Francis started with a small policy, as shown in Listing 3.

Listing 3. Building a policy for the newspaper

```xml
<wsp:Policy
   xmlns:sp="http://schemas.xmlsoap.org/ws/2002/12/secext" >
   <wsp:ExactlyOne>
      <wsp:All>
         <sp:SecurityToken>
            <sp:TokenType>sp:X509v3</sp:TokenType>
         </sp:SecurityToken>
      </wsp:All>
   </wsp:ExactlyOne>
</wsp:Policy>
```

This document contains a single policy assertion requiring the inclusion of a certain security token. If this policy were attached to a web service, clients wanting to transact with that web service are forced to send security tokens of the type specified in the policy, X509v3 in this case. Gene wants to add to the policy by specifying signing, encryption, timestamp, and username token policy assertions (see Listing 4).
Listing 4. Adding to the newspaper's policy

```xml
<wsp:Policy
 xmlns:sp="http://schemas.xmlsoap.org/ws/2002/12/secext" >
   ...
   <sp:SecurityToken>
      <sp:TokenType>sp:X509v3</sp:TokenType>
   </sp:SecurityToken>
   <sp:SignedParts />
   <sp:EncryptedParts>
      <sp:Body />
   </sp:EncryptedParts>
   <sp:TransportBinding>
      <sp:IncludeTimeStamp />
   </sp:TransportBinding>
</wsp:All>
<wsp:ExactlyOne>
</wsp:Policy>
```

The line containing `sp:WssUsernameToken11` requires a username token. `<sp:SignedParts />` requires the entire message to be signed. The `sp:EncryptedParts` XML construct contains a `<sp:Body />` tag, meaning that only the body is required to be encrypted. Lastly, the `sp:TransportBinding XML construct contains the `sp:IncludeTimeStamp` tag, requiring that a timestamp be included in the messages.

There you have it: a completed policy file, thanks to Gene and Francis. Next, see how to write a policy compactly by referencing one that already exists rather than duplicating it.

Referencing a policy document within a policy

Start with the policy from Listing 4. Now, transform it into the following, as shown in Listing 5.

Listing 5. Referencing a policy within a policy

```xml
<wsp:Policy
 xmlns:sp="http://schemas.xmlsoap.org/ws/2002/12/secext"
 xmlns:wsu=http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity
  -utility-1.0.xsd >
   <wsp:ExactlyOne>
      <wsp:All>
         <sp:SecurityToken>
            <sp:TokenType>sp:X509v3</sp:TokenType>
         </sp:SecurityToken>
      </wsp:All>
      <wsp:Policy wsu:Id="myPolicy">
         <sp:UsernameToken />
         <sp:SignedParts />
         <sp:EncryptedParts>
            <sp:Body />
         </sp:EncryptedParts>
         <wsp:Policy>
         <sp:TransportBinding>
            <sp:IncludeTimeStamp />
         </sp:TransportBinding>
         </wsp:Policy>
      </wsp:Policy>
      <wsp:PolicyReference URI="#myPolicy" />
   </wsp:ExactlyOne>
</wsp:Policy>
```
Notice that three policy assertions are placed inside `wsp:Policy`, and an ID is associated with it (`myPolicy`). The `wsp:PolicyReference` statement basically copies and pastes the policy construct to the specified location. Gene thinks this is excellent for creating compact policy expressions.

In the next section, you learn about policy operators and how you can manipulate policy documents using the mathematical properties of the policy operators.

**Using policy operators**

The constructs that make up a policy, such as policy assertions and policy alternatives, have certain mathematical properties. Knowing these properties enables you to write complex policies in the most compact way. In this section, you learn these properties and how they apply to policy documents.

**Constructs overview**

The constructs that apply to the rest of this section in expressing complex policy in the most compact way, are `wsp:Policy`, `wsp:All`, and `wsp:ExactlyOne`. To help in writing complex policies, these three constructs can be nested within each other. The following rules apply when transforming a compact expression (with as few expressions as possible) into a normal form expression (no line is wasted; the entire expression is written in the most verbose way):

- Equivalence
- Empty
- Associative
- Commutative
- Distributive

You see how to use each of these rules in the rest of this section.

**Equivalence**

This rule, equivalence, specifies that `wsp:Policy` is equivalent to `wsp:All`. Because `wsp:All` specifies a policy alternative, `wsp:Policy` also specifies a policy alternative.

**Empty**

Both a policy and a policy alternative can be empty. If a policy is empty, such as `<wsp:ExactlyOne />`, then the policy has zero policy alternatives. Similarly, if a policy alternative is empty, such as `<wsp:All />` or `<wsp:Policy />`, then the policy has zero policy assertions.

**Associative**

The operators `wsp:ExactlyOne` and `wsp:All` are also associative. This means that, using `wsp:ExactlyOne` as defined in Listing 6 (where `<!-- assertion a -->` and `<!-- assertion b -->` are any assertion), the policy is equivalent to the policy in Listing 4.
Listing 6. **wsp:ExactlyOne showing association**

```xml
<wsp:ExactlyOne>
  <!-- assertion a -->
  <wsp:ExactlyOne> <!-- assertion b --> </wsp:ExactlyOne>
</wsp:ExactlyOne>
```

The equivalent policy of `wsp:ExactlyOne` is shown in Listing 7 using association.

Listing 7. **Equivalent policy**

```xml
<wsp:ExactlyOne>
  <!-- assertion a -->  <!-- assertion b -->
</wsp:ExactlyOne>
```

And the policy, using `wsp:All`, in Listing 8 is equivalent to the policy in Listing 9.

Listing 8. **wsp:All showings association**

```xml
<wsp:All>
  <!-- assertion a -->  <!-- assertion b -->
</wsp:All>
```

The equivalent policy, using the association of `wsp:All`, is shown in Listing 9.

Listing 9. **Equivalent policy**

```xml
<wsp:All>
  <!-- assertion b -->  <!-- assertion a -->
</wsp:All>
```

Next you see how the two operators are also commutative.

**Commutative**

The operators `wsp:ExactlyOne` and `wsp:All` are also commutative. This means that when two or more policy assertions are nested within either operator, their order does not matter. For example, the statement in Listing 10 using `wsp:All` is equivalent to the statement in Listing 11.

Listing 10. **<wsp:All> showing commutative operators**

```xml
<wsp:All>
  <!-- assertion a -->  <!-- assertion b -->
</wsp:All>
```

The equivalent policy of `wsp:ExactlyOne` is shown in Listing 11 using commutative operators.

Listing 11. **Equivalent statement**

```xml
<wsp:All>
  <!-- assertion b -->  <!-- assertion a -->
</wsp:All>
```
The same applies to the `wsp:ExactlyOne` operator shown in Listing 12 and Listing 13.

**Listing 12. `wsp:ExactlyOne` operator showing commutative operators**

```xml
<wsp:ExactlyOne>
  <!-- assertion a --> <!-- assertion b -->
</wsp:ExactlyOne>
```

The equivalent policy, using the commutative operator of `wsp:ExactlyOne`, is shown in Listing 13.

**Listing 13. Equivalent statement**

```xml
<wsp:ExactlyOne>
  <!-- assertion b --> <!-- assertion a -->
</wsp:ExactlyOne>
```

Next, you see how the operators are also distributive.

**Distributive**

Both `wsp:ExactlyOne` and `wsp:All` are distributive. Following are examples to show this in the policy expression space (see Listing 14).

**Listing 14. `wsp:All` distributing over `wsp:ExactlyOne`, part 1**

```xml
<wsp:All>
  <wsp:ExactlyOne>
    <!-- assertion a --> <!-- assertion b -->
  </wsp:ExactlyOne>
</wsp:All>
```

The equivalent policy, by distributing `wsp:All` over `wsp:ExactlyOne`, is shown in Listing 15.

**Listing 15. `wsp:All` distributing over `wsp:ExactlyOne`, part 2**

```xml
<wsp:ExactlyOne>
  <wsp:All>
    <!-- assertion a -->
  </wsp:All>
  <wsp:All>
    <!-- assertion b -->
  </wsp:All>
</wsp:ExactlyOne>
```

And here is a more complex example. This one is similar to multiplying \((x + a)\) by \((y + z)\) in that you get four expressions: \(xy + xz + ay + az\) (see Listing 16).

**Listing 16. `wsp:All` distributing over `wsp:ExactlyOne`, part 3**

```xml
<wsp:All>
  <wsp:ExactlyOne>
    <!-- assertion a --> <!-- assertion b -->
  </wsp:ExactlyOne>
  <wsp:ExactlyOne>
    <!-- assertion c --> <!-- assertion d -->
  </wsp:ExactlyOne>
</wsp:All>
```
The equivalent policy, by distributing `<wsp:All>` over `<wsp:ExactlyOne>`, is shown in Listing 17.

### Listing 17. `<wsp:All>` distributing over `<wsp:ExactlyOne>`, part 4

```xml
<wsp:ExactlyOne>
  <wsp:All>
    <!-- assertion a --> <!-- assertion c -->
  </wsp:All>
  <wsp:All>
    <!-- assertion a --> <!-- assertion d -->
  </wsp:All>
  <wsp:All>
    <!-- assertion b --> <!-- assertion c -->
  </wsp:All>
  <wsp:All>
    <!-- assertion b --> <!-- assertion d -->
  </wsp:All>
</wsp:ExactlyOne>
```

Note that distributing `<wsp:All>` over an empty `<wsp:ExactlyOne>` results in no alternatives (`<wsp:ExactlyOne />`), because it is nearly the same as multiplying by zero. This also gives the same result if either of the `<wsp:ExactlyOne>` expressions in Listing 15 are empty: the result is a policy with no alternatives.

### Idempotent

The operators are also idempotent, which means that any value having the idempotent property multiplied by that same value remains unchanged. The examples in Listings 18-21 show this property.

#### Listing 18. `<wsp:All>` showing idempotence

```xml
<wsp:All>
  <wsp:All> <!-- assert a --> <!-- assert b --> </wsp:All>
</wsp:All>
```

The equivalent policy, using the idempotence of `<wsp:All>`, is shown in Listing 19.

#### Listing 19. Equivalent listing

```xml
<wsp:All>
  <!-- assertion a --> <!-- assertion b -->
</wsp:All>
```

Listing 20 and Listing 21 show that `<wsp:ExactlyOne>` is idempotent.

#### Listing 20. `<wsp:ExactlyOne>` showing idempotence

```xml
<wsp:ExactlyOne>
  <wsp:ExactlyOne> <!-- assert a --> <!-- assert b --> </wsp:ExactlyOne>
</wsp:ExactlyOne>
```

The equivalent policy, using the idempotence of `<wsp:ExactlyOne>`, is shown in Listing 21.
Listing 21. Equivalent listing

```xml
<wsp:ExactlyOne>
    <!-- assert a --> <!-- assert b -->
</wsp:ExactlyOne>
```

Essentially, a `wsp:All` nested within a `wsp:All`, or a `wsp:ExactlyOne` nested within a `wsp:ExactlyOne`, is redundant, which means you can remove one of the `wsp:All` or `wsp:ExactlyOne` operators from the expression. It is the same as multiplying anything by 1: the value remains unchanged. The mathematical properties of these operators are important to understand when you learn about manipulating and normalizing policy documents using Apache Neethi in the next section.

Using Apache Neethi to work with policy documents

The Apache Neethi project is what Axis2 uses as its run-time representation of policy documents. In this section you install Apache Neethi; load up some policy documents; normalize, merge, and intersect them; and then view the resulting normalized policy documents. Now is a good time to download the sample code from the Downloadable resources section. Unzip it, and look at the readme file.

Reading policy documents

Before you can do anything useful using WS-Policy, you need to be able to read and write policy. Francis thinks this is cool. She is looking forward to intersecting two policy documents. See Listing 22 for the policy reader.

Listing 22. Reading a policy document

```java
public static Policy readPolicy(String file){
    Policy policy = null;
    try{
        PolicyReader reader =
            PolicyFactory.getPolicyReader
                (PolicyFactory.DOM_POLICY_READER);
        FileInputStream fis =
            new FileInputStream(file);
        policy = reader.readPolicy(fis);
    } catch(Exception e) {
        System.out.println("Exception occurred: "+ e);
        e.printStackTrace();
    }
    return policy;
}
```

Place the `readPolicy()` method in a file called PolicyUtility.java. You use it later for reading the policy files in your Apache Neethi experiment. This method simply reads in a specified policy file, places it in an object-named policy, and returns it. Next is the policy writer.

Writing policy documents

Writing a policy document allows you to view the results of your policy manipulations. See Listing 23 for the policy writer code.
Listing 23. Writing a policy document

```java
public static void writePolicy(Policy w, String file){
    try{
        FileOutputStream fos =
            new FileOutputStream(file);
        PolicyWriter writer =
            PolicyFactory.getPolicyWriter
                (PolicyFactory.StAX_POLICY_WRITER);
        writer.writePolicy(w, fos);
    } catch(Exception e) {
        System.out.println("Exception occurred: " + e);
        e.printStackTrace();
    }
}
```

Place this in the same class file as before: PolicyUtility.java. This method writes the contents of a given policy document to a given filename. Next, you use these two utility methods in experimenting with Apache Neethi.

Normalizing a policy document

When a document is normalized in Apache Neethi, it is arranged in normal form (not compactly). In the sample code that comes with this tutorial (see Downloadable resources), you see three files: normalize1.xml, normalize2.xml, and normalize_output_final.xml. normalize1.xml, in Listing 24, contains a policy document that references an outside policy. normalize2.xml, shown in Listing 25, is the document that normalize1.xml is referring to.

Listing 24. normalize1.xml

```xml
<wsp:Policy
xmlns:sp="http://schemas.xmlsoap.org/ws/2002/12/secext"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd" >
    <wsp:ExactlyOne>
        <wsp:All>
            <sp:SecurityToken>
                <sp:TokenType>sp:X509v3</sp:TokenType>
            </sp:SecurityToken>

            <wsp:Policy wsu:Id="#myPolicy">
                <sp:UsernameToken />
                <sp:SignedParts />
                <sp:EncryptedParts>
                    <sp:Body />
                </sp:EncryptedParts>
            </wsp:Policy>

            <sp:TransportBinding>
                <sp:IncludeTimeStamp />
            </sp:TransportBinding>
        </wsp:All>
    </wsp:ExactlyOne>
</wsp:Policy>
```

The policy referenced above, myPolicy, is shown in Listing 25.
Listing 25. normalize2.xml
<wsp:Policy wsu:Id="myPolicy"
xmlns:sp="http://schemas.xmlsoap.org/ws/2002/12/secext"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:UsernameToken />
      <sp:SignedParts />
      <sp:EncryptedParts>
        <sp:Body />
      </sp:EncryptedParts>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>

The code to normalize the policy in Listing 24 is shown in Listing 26.

Listing 26. Normalizing a policy document
import org.apache.ws.policy.Policy;
import org.apache.ws.policy.util.PolicyRegistry;
import org.apache.ws.policy.util.PolicyReader;
import org.apache.ws.policy.util.PolicyWriter;
import org.apache.ws.policy.util.PolicyFactory;
import java.io.FileInputStream;
import java.io.FileOutputStream;
class PolicyUtility{
  public static void main(String argsv[]){
    Policy normalize1 = readPolicy("normalize1.xml");
    Policy normalize2 = readPolicy("normalize2.xml");
    PolicyRegistry pr = new PolicyRegistry();
    pr.register("#myPolicy", normalize2);
    normalize1 = (Policy)normalize1.normalize(pr);
    writePolicy(normalize1, "normalize_output.xml");
  }
}

The result of the normalization is in normalize_output_final.xml (see Listing 27).

Listing 27. The resulting normalize_output_final.xml
<wsp:ExactlyOne>
  <wsp:All>
      <sp:TokenType>sp:X509v3</sp:TokenType>
    </sp:SecurityToken>
    <sp:UsernameToken>
      <sp:SignedParts>
        <sp:Body />
      </sp:SignedParts>
      <sp:TransportBinding>
        <sp:IncludeTimeStamp />
      </sp:TransportBinding>
    </sp:UsernameToken>
  </wsp:All>
</wsp:ExactlyOne>
If you run the program by following the instructions in the readme file, the result is written to normalize_output.xml. Notice how the PolicyReference in normalize1.xml is replaced with the policy alternative in normalize2.xml. Feel free to modify the two files to see what results you get when running other normalizations. Next, merge two policy documents.

**Merging two policy documents**

Merging enables two policy documents to merge all the policy alternatives from two separate files. For example, if you have two files, each with two policy alternatives, you end up with one file with four policy alternatives. If one file has two policy alternatives and the other file has one policy alternative, you end up with two policy alternatives. Take a look at the two files you can merge in Listings 28 and 29.

**Listing 28. merge1.xml**

```xml
<wsp:Policy
 xmlns:sp="http://schemas.xmlsoap.org/ws/2002/12/secext"
>
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:SecurityToken>
        <sp:TokenType>sp:X509v3</sp:TokenType>
      </sp:SecurityToken>
      <sp:UsernameToken />
      <sp:EncryptedParts>
        <sp:Body />
      </sp:EncryptedParts>
      <sp:TransportBinding>
        <sp:IncludeTimeStamp />
      </sp:TransportBinding>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

You can merge the policy above, merge1.xml, with the policy shown in Listing 29, merge2.xml.
Listing 29. merge2.xml

```xml
<wsp:Policy
xmlns:sp="http://schemas.xmlsoap.org/ws/2002/12/secext">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:SecurityToken>
        <sp:TokenType>sp:X509v3</sp:TokenType>
      </sp:SecurityToken>
      <sp:SignedParts />
      <sp:EncryptedParts />
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

Experience is the best way to learn this concept. So Listing 30 continues the PolicyUtility.java class you started by implementing code to merge the two policy documents in Listings 28 and 29.

Listing 30. Merging two policy documents

```java
... normalize1 = (Policy)normalize1.normalize(pr);
writePolicy(normalize1, "normalize_output.xml");

Policy merge1 = readPolicy("merge1.xml");
Policy merge2 = readPolicy("merge2.xml");
merge1 = (Policy)merge1.merge(merge2);
writePolicy(merge1, "merge_output.xml");
...
```

Here, two policy documents are read and merged, and then the output of the merging is written to a file (see Listing 31).

Listing 31. The resulting merge_output_final.xml

```xml
  <wsp:ExactlyOne>
    <wsp:All>
        <sp:TokenType>sp:X509v3</sp:TokenType>
      </sp:SecurityToken>
      <sp:UsernameToken>
      </sp:UsernameToken>
      <sp:EncryptedParts>
        <sp:Body>
          <sp:IncludeTimeStamp>
          </sp:IncludeTimeStamp>
        </sp:Body>
      </sp:EncryptedParts>
      <sp:TransportBinding>
        <sp:IncludeTimeStamp>
        </sp:IncludeTimeStamp>
      </sp:TransportBinding>
      <sp:SecurityToken>
        <sp:TokenType>sp:X509v3</sp:TokenType>
      </sp:SecurityToken>
      <sp:SignedParts>
      </sp:SignedParts>
      <sp:EncryptedParts>
      </sp:EncryptedParts>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```
Go back to Listings 28 and 29. They each contain one policy alternative, so the resulting document contains one policy alternative with all the policy assertions in both policy alternatives copied over to a new one. Check out the results in merge_output_final.xml, shown in Listing 31. Notice that all the policy assertions from the single policy alternative in merge1.xml and all the policy assertions from the single policy alternative merge2.xml are combined in merge_output_final.xml to form a single, merged policy alternative.

**Policies and signatures**
Because policies hold critical information about how web services interact with each other, always make sure that the policy you received is indeed from the entity you requested it from. It is good practice to sign your policies to prevent spoofing.

**Intersecting two policy documents**
Intersection is an even more important concept. Basically, if two web services are going to interact, they want to do so using the policy alternative that both support. The web services need to intersect each other’s policy documents. The result is a merge of the policy alternatives that match. Francis is keen on this idea so that, when calling other web services with her client code, she can intersect the available policies of the *Daily Moon* with the policies of the web service she wishes to interact with (see Listing 32).

**Listing 32. intersect1.xml**

```xml
<wsp:Policy
xmlns:sp="http://schemas.xmlsoap.org/ws/2002/12/secext">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:SecurityToken>
        <sp:TokenType>sp:X509v3</sp:TokenType>
      </sp:SecurityToken>
      <sp:UsernameToken />
      <sp:SignedParts />
      <sp:EncryptedParts>
        <sp:Body />
      </sp:EncryptedParts>
      <sp:TransportBinding>
        <sp:IncludeTimeStamp />
      </sp:TransportBinding>
    </wsp:All>
    <wsp:All>
      <sp:SecurityToken>
        <sp:TokenType>sp:X509v3</sp:TokenType>
      </sp:SecurityToken>
      <sp:UsernameToken />
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

You can intersect the policy above, `intersect1.xml`, with the policy shown in Listing 33, `intersect2.xml`. 
Listing 33. intersect2.xml

```xml
xmlns:sp="http://schemas.xmlsoap.org/ws/2002/12/secext">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:SecurityToken>
        <sp:TokenType>sp:X509v3</sp:TokenType>
      </sp:SecurityToken>
      <sp:SignedParts />
    </wsp:All>
    <wsp:All>
      <sp:SecurityToken>
        <sp:TokenType>sp:X509v3</sp:TokenType>
      </sp:SecurityToken>
      <sp:UsernameToken />
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

Look at the code to make the intersection of intersect1.xml and intersect2.xml happen in Listing 34.

Listing 34. Intersecting two policy documents

```java
merge1 = (Policy)merge1.merge(merge2).normalize();
writePolicy(merge1, "merge_output.xml");

Policy intersect1 = readPolicy("intersect1.xml");
Policy intersect2 = readPolicy("intersect2.xml");
intersect1 = (Policy)intersect1.merge(intersect2);
writePolicy(intersect1, "intersect_output.xml");
```

Here, two policy documents are read and intersected, and then the output of the intersection is written to a file. Look at the two files in Listings 32 and 33. They each contain two policy alternatives, but only one from each file matches. The non-matching policy alternatives are removed, and the matching policy alternatives are merged together (with one match, in this example). Now, check out the actual results in merge_output_final.xml, shown in Listing 35.

Listing 35. The resulting merge_output_final.xml

```xml
  <wsp:ExactlyOne>
    <wsp:All>
        <sp:TokenType>sp:X509v3</sp:TokenType>
      </sp:SecurityToken>
      <sp:UsernameToken />
      <sp:SecurityToken>
        <sp:TokenType>sp:X509v3</sp:TokenType>
      </sp:SecurityToken>
      <sp:UsernameToken />
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```
Notice how the matching policy alternatives in each file (the second policy alternatives in intersect1.xml and intersect2.xml) are essentially merged together in the single policy alternative shown in merge_output_final.xml. That completes your work with Apache Neethi.

Next, explore working with policy documents in WSDL.

**Working with policy documents in WSDL**

The WS-PolicyAttachment specification details how policies are attached to bindings and is essentially the glue that enforces a web service to adhere to a policy. This section covers the goals of the WS-PolicyAttachment specification and how it enables you to attach policies to your web service.

**WS-PolicyAttachment**

The goal of the WS-PolicyAttachment specification is to define a means for developers to reference policies from WSDL definitions. Details in the specification also exist to associate policies with UDDI entities, which are not covered in this tutorial. See Related topics section for a link to the full WS-PolicyAttachment specification. The rest of this section shows how to follow the WS-PolicyAttachment specification for referencing the *Daily Moon* policy document to WSDL. Gene is excited about this task so that he can get policy up and running on the *Daily Moon* web services as quickly as possible.

**Requiring WS-Policy**

There is a construct to enforce your WSDL to use and reference policies. Listing 36 shows an example.

**Listing 36. Requiring policy in your WSDL**

```xml
<wsp:UsingPolicy wsdl:Required="true" />
```

Next, you add policies inline with the WSDL.

**Where to place policies**

Place policies right after the `wsp:UsingPolicy` requirement line. For the *Daily Moon*, you can do it like Gene did, shown in Listing 37.

**Listing 37. Defining policies in WSDL**

```xml
<wsp:UsingPolicy>
    <wsdl:types>
        <wsdl:schema
```
There you go! The policy is inside the WSDL. Now you only have to associate it with a binding, which you can do next.

**Attaching policies to bindings**

Now that a policy has been written, it is useless unless you associate it with a port binding. If a binding was defined, Listing 38 shows you how to attach the *Daily Moon* policy with the binding.

**Listing 38. Associating a policy with a binding: enforcing the policy**

```xml
<wsdl:definitions ...

<wsp:UsingPolicy wsdl:Required="true" />
<wsp:Policy wsu:Id="myPolicy">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:SecurityToken>
        <sp:TokenType>sp:X509v3</sp:TokenType>
      </sp:SecurityToken>
      <sp:UsernameToken />
      <sp:SignedParts />
      <sp:EncryptedParts>
        <sp:Body />
      </sp:EncryptedParts>
      <sp:TransportBinding>
        <sp:IncludeTimeStamp />
      </sp:TransportBinding>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>

<wsdl:binding name="myBinding" type="tns:myPortType">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" />
  <wsp:PolicyReference URI="#myPolicy" wsdl:required="true" />
</wsdl:binding>
</wsdl:definitions>
```

Notice the way Gene wrote the WSDL. In the port binding, look at the `wsp:PolicyReference` tag that refers to the policy, *myPolicy*. This tag specifies that the web service operation (*myOperation*) being bound (using *myBinding*) to the specified portType (*myPortType*) must enforce that all
incoming messages adhere to the policy, as defined in the policy with wsu:Id="#myPolicy". That is all there is to it! If Axis2 fully supported policies, then a client trying to interact with the Daily Moon web service would not be able to perform successful transactions with it unless it adheres to all assertions of one of the alternatives to the policy.

Summary

In order for web services to be truly useful in the enterprise environment, they need to have appropriate configuration and security policies. WS-Policy solves this problem by allowing policy definitions to be bound to web services. These policy definitions define how the transaction is to occur, and they can require a myriad of things to enforce a more secure environment to transact.

By enforcing a certain level of encryption and digital signatures, you can secure the data during its transport and limit access to authorized individuals or organizations, and then verify that information has not been altered in transit. Having your policy require a timestamp (and signing it) allows you to prevent messages from being captured and replayed.

In this tutorial, the staff at the Daily Moon enforced security for the web services they created and secured in earlier parts of this series. Next, in Part 6, they will apply WS-I, or WS-Interoperability, to web services.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Part 5 source code</td>
<td>wspolicycode.zip</td>
<td>5KB</td>
</tr>
</tbody>
</table>
Related topics

- The WS-Policy document (version 1.2), dated March 2006, has more information about WS-Policy.
- To get a good grounding in XML, read the Introduction to XML tutorial (developerWorks, August 2002).
- Download Apache Axis2 version 1.0 or higher.
- Download Java 2 Standard Edition version 1.4.2 or higher.
- The IBM SOA Web site offers an overview of SOA and how IBM can help you to get there.

© Copyright IBM Corporation 2007
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)