

Build secure Web services

Using Rational Application Developer

Skill Level: Introductory

[Indran Naick \(indrann@us.ibm.com\)](mailto:indrann@us.ibm.com)
e-business architect
EMC

[Jeff Miller \(jeffmil@us.ibm.com\)](mailto:jeffmil@us.ibm.com)
e-business architect
EMC

02 Dec 2005

Security is an essential part of any Web service. Rational Application Developer allows you to take advantage of security standards and without too much effort create all of the necessary configuration to add security to your services. This tutorial shows you how to authenticate using a user ID and password, ensure integrity using digital signatures, and ensure confidentiality using encryption.

Section 1. Before you begin

Should I take this tutorial?

Take this tutorial if you are a Web developer or architect and want to understand how to build secure Web services using Rational® Application Developer. This tutorial assumes that you have a basic knowledge of Java™ technology and Web services. It takes you through a fairly complete example of adding signatures, encryption, and a token to a Web service.

Rational Application Developer is easy to use, so you'll find this tutorial easy to follow even if you're a beginner to Web services and Java technology. In addition to

showing you how to use the tools within Rational Application Developer, this tutorial gives you an introduction to Web services security and shows you what happens behind the scenes in a Web services architecture. If you are a complete newbie to Java technology, some of these concepts might be easier to follow if you have a basic understanding of Web services.

About the tutorial

This tutorial describes the functionality available within Rational Application Developer to secure Web services. Rational Application Developer provides features with which you can apply authentication, integrity, and confidentiality to Web services.

There are various mechanisms for implementing security in a distributed system. Many of these secure the transport protocol and use a variety of other security mechanisms to achieve their objectives. The security that we will focus on in this tutorial is SOAP message security. This means that the security information is contained in and travels with each SOAP message, making it transport-independent. This security is based on:

- **XML digital signatures:** provides integrity
- **XML encryption:** provides confidentiality
- **Security tokens:** provides authentication

It is important to distinguish between security mechanisms that are *transport dependent* and those that are *transport independent*. Developers often strive to ensure that their services are not bound to any particular transport. If your security model is based on the transport, you are indirectly tightly coupling your service, should you need it to be secure, to a fixed protocol.

In addition, it is preferable to have security abstracted out of the service -- that is, to have it be a deployment-time option. This allows you to modify the security as and when required without changing the service. Changing your code every time you change your security policies can be very difficult, expensive, and prone to error. Having security abstracted out also allows you the option of deploying your services with or without security. Security adds processing overhead to any operation, and it should be used only when it is warranted.

Prerequisites

To complete the steps in this tutorial, you need to install Rational Application Developer V6.0 or higher. You can download a trial copy of [Rational Application](#)

[Developer for WebSphere® Software V6.0](#) from developerWorks. The installation process is straightforward and hassle free, and you will need to complete a short registration form. WebSphere Application Server V6.0 test environment within Rational Application Developer for WebSphere Software V6.0 was used in this tutorial to test the examples. The screen captures were generated using Rational Application Developer for WebSphere Software V6.0. If you are using Rational Application Developer for WebSphere Software V6.0.1 it should still work. Some of the screens might be slightly different to the ones shown here, however the relevant fields are shown here.

You also need to download the sample code [AtomicClock.java](#).

Section 2. Security basics

When defining a security model, you have to show how data can flow through an application and over a network to meet the requirements defined by the business without exposing the data to any risk. The ISO security standard defines seven security services:

- **Identification:** Can you can tell me who you are?
- **Authentication:** Can you prove that you are who you say you are?
- **Authorization:** Now that I have determined your identity, what is the set of things that you're allowed to do?
- **Integrity:** Was the information that you sent to me changed or corrupted as it was transmitted?
- **Confidentiality:** Can anyone read the information that is passed back and forth between us?
- **Auditing:** Are all transactions recorded so that we can review them later?
- **Non-repudiation:** Can I prove that it was you who sent the message?

When enabling security for any application, one or more of the above services, based on the requirements of your application, is implemented. The real security challenge is to understand and assess the risk involved in securing any application and apply the appropriate measures. For Web services, understand what security technology exists today and at the same time track emerging standards and understand how they will be used to offset the risk in new Web services.

In Web services, the SOAP envelope is defined in XML. Web services can therefore

use many existing XML security technologies and standards, such as XML encryption and XML digital signatures. In addition to such existing security techniques, many new specifications have begun to emerge. The WS-Security specification (see [Resources](#)) is the cornerstone in the effort to pull all of these requirements together. The abstract of the WS-Security specification document says that "WS-Security describes enhancements to SOAP messaging to provide quality of protection through message integrity, message confidentiality, and single message authentication. These mechanisms can be used to accommodate a wide variety of security models and encryption technologies..."

The first version of the WS-Security specification was proposed by IBM, Microsoft, and VeriSign in April 2002. After the formalization of the April 2002 specifications, the specification was transferred to the OASIS consortium (<http://www.oasis-open.org>). The latest specification and profiles of WS-Security were proposed in March 2004 as an OASIS standard. The latest core specification, Web Services Security: SOAP Message Security 1.0 (WS-Security 2004) was standardized in March 2004. The two profiles, Web Services Security UsernameToken Profile 1.0 and Web Services Security X.509 Certificate Token Profile 1.0, were standardized at the same time.

It is important to note that WS-Security makes use of many existing technologies rather than creating new ones. XML encryption and XML digital signatures are two good examples of existing technologies that are used within this new specification.

This tutorial covers the security implementation in individual exercises. The use of security tokens, XML encryption, and XML digital signatures as currently supported by Rational Application Developer are demonstrated. Other supported features are usually variations of the examples shown here. There are a number of security extensions and unsupported features also available. A complete description of these features can be found in the redbook, [WebSphere Version 6 Web Services Handbook Development and Deployment](#).

Rational Application Developer, WebSphere 6.0 and WS-Security

The support of the April 2002 draft specification of WS-Security is provided in WebSphere Application Server Versions 5.0.2 and 5.1. WebSphere Application Server Version 6.0 supports the WS-Security 2004 specification and two token profiles (UsernameToken 1.0, X.509 Certificate Token 1.0). WebSphere Application Server V6.0 provides the runtime for the previously mentioned specifications. Rational Application Developer V6.0 provides functions to secure Web services based on these specifications.

WebSphere Application Server V6.0 has two types of the WS-Security runtime: one is for Version 5.x, the other is for Version 6.0 (Figure 21-61). Because there are two

runtimes, the server can receive a request from both a J2EE 1.3 client with Version 5.x and a J2EE 1.4 client with Version 6.0 WS-Security. However, a J2EE 1.3 client cannot connect to the Version 6.0 runtime and cannot use the new features of Version 6.0 WS-Security. If you want to use Version 6.0 functions in your secure Version 5.x application, you have to migrate from Version 5.x to Version 6.0. For more information on interoperability and migration see [WebSphere Version 6 Web Services Handbook Development and Deployment](#).

The WS-Security configuration can be done either via the Wizards in Rational Application Developer or defined directly in the IBM extension of the Web services deployment descriptors and bindings. These descriptors and bindings are based on a Web service port, that is, each port can have its own security definitions. These definitions are defined outside of the application business logic and therefore allows for the separation of roles. After the application is developed the security expert can apply the appropriate policies. This means that the original service never changes: all security processing occurs outside of the code, allowing you greater deployment flexibility.

There are two sets of security handler configurations on the client side and two on the server side. Both sides have a request generator and response consumer. As their name indicates they define the constraints on the responses and requests that are applied.

Rational Application Developer provides the ability to add security tokens, XML digital signatures, and XML encryption to Web service messages either via the Web services wizard (Now when using the wizard, you can do either one or the other, or both) or the via directly configuring the deployment descriptors and bindings.

You can add a token only via the Web services deployment descriptor. Using the latter method, you can choose to add any of the options in any combination. In this tutorial, however, you will see how to add signatures and encryption via the Web services wizard, and security tokens via the deployment descriptor. The IBM Web services Redbook [WebSphere Version 6 Web Services Handbook Development and Deployment](#) describes the steps required to do all of this using the deployment descriptor.

Steps in this tutorial

In this tutorial, security implementation is covered in the following sequence:

1. Generate a Web service without any security and look at the SOAP envelopes that the service generates.
2. Implement XML encryption and view the encrypted parts of the envelope.

3. Implement XML digital signatures and view the associated elements within the SOAP envelope.
 4. Add a security token, using a user ID and password, and view how these elements are represented within the SOAP envelope.
-

Section 3. Set up the environment

In this tutorial, you are going to build a simple service and add security to it. This service will expose the atomic clock inside your computer. Well, you don't really have an atomic clock, but in this tutorial you will create a JavaBean component called `AtomicClock` that returns the current date and time, and you'll expose it as a Web service. You will then add security to encrypt and sign the information. This is not necessarily the type of data that would typically need security, but you could think of the date and time as being a credit card number or a social security number. This example is being used for illustrative purposes only.

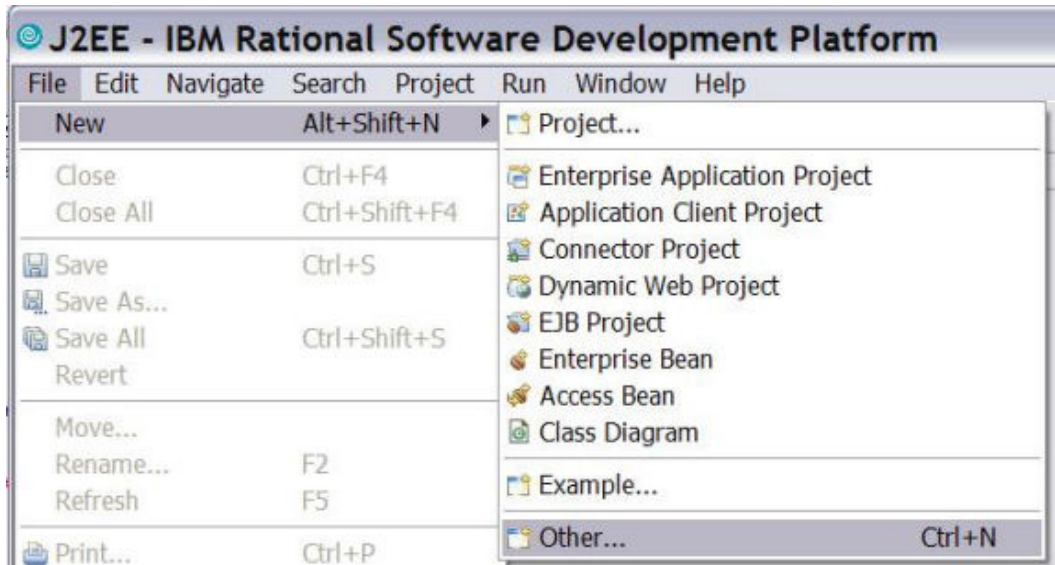
Setting up the servers

Start by defining two servers. The first server will run a Web services as well as the test clients. The second will monitor the traffic or SOAP messages that are passed between the Web services client and server.

To create an application server:

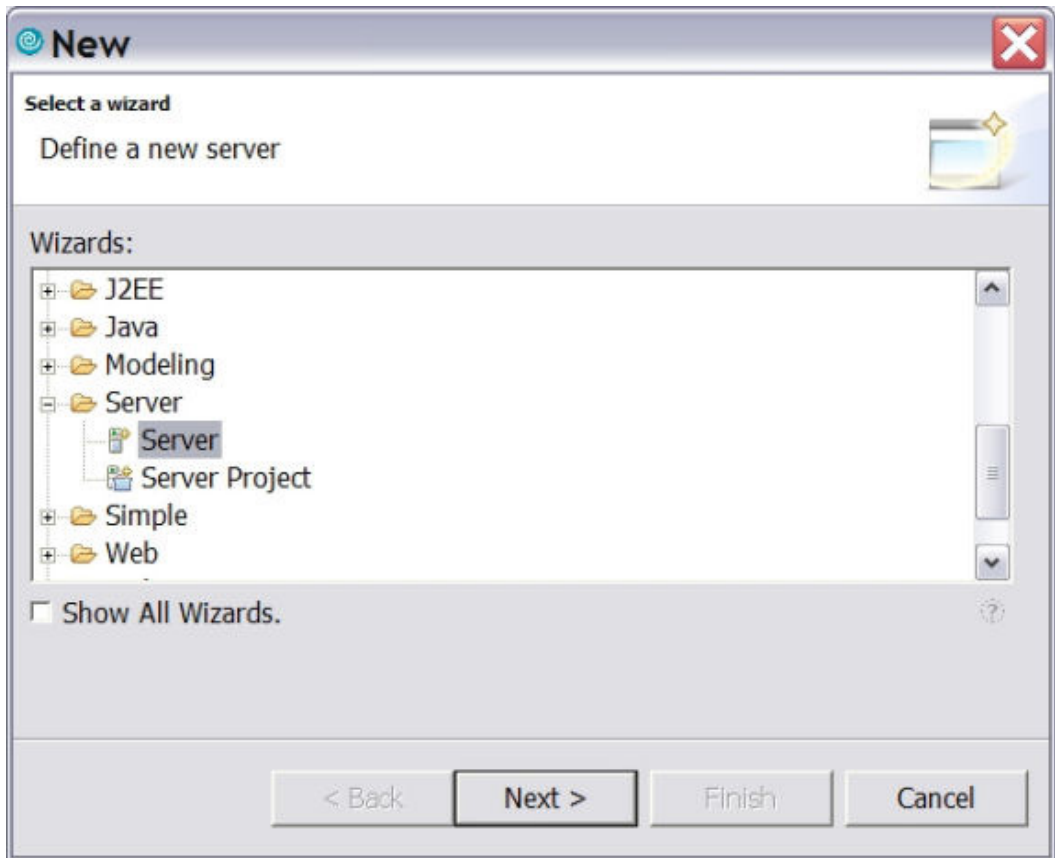
1. Start Rational Application Developer. Click **New** and select **Other** from the drop-down list.

Figure 1. Using the New button to select a project



2. Scroll down and expand Server and select **Server** from the drop-down list.

Figure 2. Selecting a Server



3. Leave the name localhost. The WebSphere V6.0 Server should be

selected by default. Click **Next**.

Figure 3. Defining the server type

4. Leave the defaults Click **Finish**.
Figure 4. Defining the server settings

The application server is created.

To setup the monitoring server:

1. Click on the servers tab and right-click on the server you just created and select **Start**.
2. After the server is started, which might take a few minutes, right-click on the server and select **Monitoring > Properties** from the context menu.
Figure 5. Monitoring Properties
3. Select the **Add** Button. Select **9080** from the list. Note the monitoring port (9081) and and click **OK**. Select the monitoring server you just defined and Click **Start**.
Figure 6. Monitoring Ports
4. Click **OK** to exit
5. To ensure that the monitor shows up when there is activity select **Window > Preferences > Internet > TCP/IP monitor** and select **Show TCP/IP Monitor view when there is activity**.
Figure 7. TCP/IP monitor settings

Both servers are now defined. These are used for all the services you define in this tutorial. You might need to restart the monitoring server each time you restart Rational Application Developer

Creating a new Rational Application Developer project

Create a dynamic Web project to hold your application. The name of this project does not really matter for Web services, since it just serves as a container for all that you'll do.

1. Select **File > New > Dynamic Web Project**.
2. Enter `AClockWeb` in the Project name field and click **Finish**.
Figure 8. Creating a new web project

Creating the JavaBean component

You can either import the code for the JavaBean component or manually add a bean and enter the code. In this case, use the code provided and import it into the workspace.

1. Create a package to hold the JavaBean component. Right-click the **AClockWeb** project in your workspace and select **New > Other > Java** and expand the Java tab and select **Package** and click **Next**.
Figure 9. Adding a java package
2. Enter the name `acpkg` and click **Finish**.
3. Right-click on the newly created package and click **Import**.
Figure 10. Importing the source
4. Select **File System** and then browse to the directory where you downloaded `AtomicClock.java` and select the file. Click **Finish**.

You now have a Java application in your workspace. In the next section, you will use it to generate a standard, insecure Web service; in subsequent sections, you'll add security features to that service.

Generating a Web service

To generate a standard Web service:

1. Right-click the `AtomicClock.java` file and select **New > Other... >** , scroll down and expand Web Service, select **Web Services**.
Figure 11. Using the Web Services Wizard
2. Click **Next**. If you get a confirmation window, click **OK** In the Web service type window, make sure the type is JavaBean Web Service and that the boxes are checked as in the figure below. You're going to generate a proxy and a test client at the same time that you generate the Web service itself. Remember to check the **Overwrite files without warning** check box as well.
Figure 12. Defining the options
3. Click **Next**.
4. Accept the defaults on the Object Selection Page. Since you started from the Bean, the fields are correct as is and include `acpkg.AtomicClock`.

Click **Next**.

5. Accept the defaults on the Service Deployment Configuration window. The test client is generated in a new Web project named `AClockWebClient`. You'll use the new WebSphere V6 Web services runtime engine. Click **Next**.
6. Accept the defaults in the Service Endpoint Interface Selection, make sure `acpkg.AtomicClock` is the specified bean and click **Next**.
7. In the Web Service JavaBean Identity window, uncheck the `main(java.lang.String[])` method, as shown in the figure below.
Figure 13. Selecting the methods to be exposed
8. Click **Finish**. At this point, several things happen:
 - The rest of the Web service support code is generated.
 - The client proxy and test client application are generated.
 - The Web project and Web client project are published to the WebSphere Test Environment application server, which itself is created if it doesn't already exist.
 - The application server is started, and the browser opens with the generated *test client*, which is a set of JSPs that use the proxy to talk to the Web service.

Be patient, as all these processes can take a few minutes to complete. The logical architecture is shown in the next figure.

Figure 14. Logical architecture of artifacts generated

9. After some time, the test client comes up and you can test your `AtomicClock` Web service. Your screen should look something like this:
Figure 15. Testing the generated code
10. Click **getTheDate()** under Methods. Click **Invoke** under Inputs. You should see the date displayed under Result.
11. Click **getTheTime()** on the left and then **Invoke** on the right. Click **Invoke** again and watch the seconds change in the time display.
12. Click **getTheFormattedDate(int)** on the left, enter a number between 0 and 3 (these correspond to the `java.text.DateFormat` static values), and click **Invoke**. Try it with a different value to see the different formats.
13. To see the URL that the proxy is accessing, click **getEndpoint()** followed by **Invoke**. The result should be

`http://localhost:9080/AClockWeb/services/AtomicClock`. This is the URL of the Web service engine running on the server.

14. To look at the messages being sent in the request and response, open the Server perspective if it is not already open. In the Server Configuration view, right-click and start the new TCP Monitor server. This server listens on a specified port, captures the HTTP request, displays it, and passes the request on to a specified destination port. By default, the destination port is 9080 (WebSphere). The TCP monitor also displays the response that passes back through it to the client.
15. Back in the Web browser Test Client, click **getEndPoint()**, then **Invoke** and copy the result. Click `http://localhost:9080/AClockWeb/services/AtomicClock` and paste in the copied URL. Change the port to 9081. The default port on which the TCP Monitor listens is `http://localhost:9081/AClockWeb/services/AtomicClock`.
16. Click **getTheFormattedDate()** again, enter a value for the format between 0 and 3, and click **Invoke**. Try this a few times with different values.

To see the HTTP request and responses that carry the SOAP messages:

1. Towards the bottom of the window, click the TCP/IP Monitor tab.
2. Double-click the title bar of the view to maximize it.
3. Near the top, under `localhost:9081`, click each of the entries to see the request and response in the bottom panes on the left and right, respectively. For easier viewing, you might want to select the XML View from the drop down box or copy the request and response messages to WordPad or another text editor. Select the message text and then copy via the right-click contextual menu.

The request should look like this:

```
<soapenv:Envelope xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Header/>
  <soapenv:Body>
  <p928:getTheDate xmlns:p928="http://acpkg"/>
  </soapenv:Body>
</soapenv:Envelope>
```

And the response should look like this:

```
<soapenv:Envelope xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Header/>
  <soapenv:Body>
  <p928:getTheDateResponse xmlns:p928="http://acpkg">
  <getTheDateReturn>The date is Sun Nov 13 2005</getTheDateReturn>
  </p928:getTheDateResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Section 4. Integrity: Adding an XML digital signature

A message is said to have *integrity* if you can be certain that it has not been tampered with. With Web services, integrity is ensured using XML digital signatures. Rational Application Developer allows you to sign different parts of the SOAP message: the body, the security token (discussed in [Authentication: Adding a security token](#)), or the timestamp. However, if you're using the Web services wizard, you can only sign the body. If you want to sign any other parts, use the Web services deployment descriptor.

Adding a digital signature to the messages

To add a digital signature to the body of your service's SOAP messages using the Web services wizard:

1. Right-click the AtomicClock.java file and select **New > Other... >**, scroll down and expand Web Service, select **Web Services**.
Figure 16. Using the Web Services Wizard
2. Click **Next**. If you get a confirmation window, click **OK**. In the Web service type window, make sure the type is JavaBean Web Service and that the boxes are checked as in the figure below. You're going to generate a proxy and a test client at the same time that you generate the Web service itself. Remember to check the **Overwrite files without warning** check box as well.
Figure 17. Defining the options
3. Click **Next**.

4. Accept the defaults on the Object Selection Page. Since we started from the Bean the fields are correctly filled to `acpkg.AtomicClock`. Click **Next**.
5. Accept the defaults on the Service Deployment Configuration window. The test client is generated in a new Web project named `AClockWebClient`. You'll use the new WebSphere V6 Web services runtime engine. Click **Next**.
6. Accept the defaults in the Service Endpoint Interface Selection, make sure `acpkg.AtomicClock` is the specified bean and click **Next**.
7. In the Web Service JavaBean Identity window, uncheck the `main(java.lang.String[])` method, as shown in the figure below.
Figure 18. Selecting the security option
8. Select **XML Digital Signature** as the security configuration and click **Next**.
9. A warning window opens stating that your choices may not comply with WS-I. This is expected because WS-Security is not yet part of the WS-I Basic Profile 1.0. Click **Ignore** to proceed.
Figure 19. WS-I warning
10. Leave the defaults on the Web Service Test Window and click **Next**.
11. Leave the defaults in the Web Service Proxy Page window. Notice that XML Digital Signature is now the default for the client proxy. Click **Next**. Click **Ignore** again on the WS-I compliance warning for the client.
Figure 20. Defining security on the Client
12. Leave the defaults in the Web Service Client Test window and click **Next**.
13. Leave the defaults in the Web Service Publication window and click **Finish**. The wizard takes time to generate the Web service. After it has completed, it launches your browser with the generated test client for the Web service.
14. Select **getEndpoint()** and then **Invoke**. Copy the returned URL from the Result frame.
Figure 21. Testing the generated code
15. Using `setEndpoint()`, set the endpoint to:

```
http://localhost:9081/AClockWeb/services/AtomicClock
```



```

1UECBMIS2FuYwdhd2ExDzANBgNVBACTBllhbWF0bzEMMAoGAlUEChMDSUJNMQwwCgYDVQQLEwNUUkwXGTAXBgNVB
AMTEFNPQVAgMi4xIFRlc3QgQ0ExIjAgBgkqhkiG9w0BCQEW21hcnV5YW1hQGpwLmlibS5jb22CAgEBMA0GCSqGS
Ib3DQEBBQUAA4GBAHkthdGDgCvdIL9/vXUo74xpFOQd/rrlowBmMdb1TWdOyzwbOHC71kUlnKrki7SofwSLSDUP5
71liiMXUx3tRdmAVCoMMFuDXh9V7212luXccx0s1S5KN0D3xW97LLNegQC0/b+aFD8XKw2U5ZtwbwFTRgs097dmz
09RosDKkLlM</wsse:BinarySecurityToken>
<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#">
<ec:InclusiveNamespaces xmlns:ec="http://www.w3.org/2001/10/xml-exc-c14n#"
PrefixList="wsse ds xsi soapenc xsd soapenv "/>
</ds:CanonicalizationMethod>
<ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
<ds:Reference URI="#wssecurity_signature_id_1910483460897397841">
<ds:Transforms>
<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="xsi soapenc p928 xsd wsu soapenv "/>
</ds:Transform>
</ds:Transforms>
<ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
<ds:DigestValue>+jrIAi4aJc8oOutGUkDuW665zZA</ds:DigestValue>
</ds:Reference>
</ds:SignedInfo>
<ds:SignatureValue>tktHbLNd4/4Jj/e/sF1CmQfcxgfgM15u9yUUTH7K15iWP/MoXvw/o0yHhGtS3X7s
Fgfhx5jKtA5uyje42KQ4kcAdox71zBZS0k3Zi1wAycacKsCWy2p5SNiRDzO91k981hJYIS0uU+vxR5MsBircti60A
Pxo3L0MpsImaNAftE</ds:SignatureValue>
<ds:KeyInfo>
<wsse:SecurityTokenReference>
<wsse:Reference URI="#x509bst_8784860953065228620"
ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token
-profile-1.0#X509"/>
</wsse:SecurityTokenReference>
</ds:KeyInfo>
</ds:Signature>
</wsse:Security>
</soapenv:Header>
<soapenv:Body xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss
-wssecurity-utility-1.0.xsd" wsu:Id="wssecurity_signature_id_1910483460897397841">
<p928:getTheFormattedDate xmlns:p928="http://acpkg">
<format>3</format>
</p928:getTheFormattedDate>
</soapenv:Body>
</soapenv:Envelope>

```

First, notice how long the (abbreviated) `<Security>` header is compared to the actual `<Body>` of the message. The `<BinarySecurityToken>` is a digital certificate of type X509.v3 and is Base64-encoded. Its value is shortened in the listing above but is really 1,116 bytes as sent in the request. It has a `wsu:Id` attribute by which it can be referred to.

The next section is the `<Signature>`. It contains three main parts: `<SignedInfo>`, `<SignatureValue>`, and `<KeyInfo>`. The first subelement is `<SignedInfo>`, which specifies information about how and what is signed. `<CanonicalizationMethod>` indicates how the content to be signed is put into a standard canonical form. `<SignatureMethod>` is the algorithm used to do the signing. `<Reference>` contains a URI attribute that points to the actual content that is signed. Notice that the URI matches the `wsu:Id` attribute of the `<Body>` element further down. Inside `<Reference>`, the `<Transforms>` element designates the order of processing that the content goes through before being signed. Then the

<DigestMethod> specifies the hashing algorithm used on the <Body> element, and the <DigestValue> is the <Body> hash itself.

Next, the <SignatureValue> is actually the value of the signature of the entire <SignedInfo> element. The <KeyInfo> element contains information about the key required to validate the signature. Notice that inside <SecurityTokenReference>, the <Reference> element's URI attribute contains the `wsu:Id` of the <BinarySecurityToken> earlier in the header. Since this token is a digital certificate, it contains the public key that should be used for validation. Sometimes the certificate is not sent right inside the header, but is referred to with a URL that points to an external location of the certificate, thus allowing a shorter header.

Finally, you come to <soapenv:Body> itself, which is unchanged from a normal, unsigned SOAP message, except for the addition of the `wsu:Id` attribute giving it an identifier value so that it can be referenced from the security header. Clever, no?

The format of the response security header is the same.

Section 5. Confidentiality: Adding XML encryption

This step is very much a repeat of the steps you completed for the digital signature. However, this time you choose XML encryption instead of XML digital signature in the Web services wizard. Steps are repeated here for the sake of completeness. The end result of this process is a <wsse:Security> header containing encryption information for the encrypted SOAP <Body>. You'll see that this security header content is different from the content we discussed earlier. There's no need to stop the ACServer for this section; you can leave it running. You'll just restart the application after it is recreated.

Encrypting the messages

Confidentiality can be applied to either the body content or the security token. Again, use the Web services wizard, which applies confidentiality to the body.

1. Right-click the AtomicClock.java file and select **New > Other... >** , scroll down and expand Web Service, select **Web Services**.
Figure 23. Using the Web Services Wizard
2. Click **Next**. If you get a confirmation window, click **OK** In the Web service

type window, make sure the type is JavaBean Web Service and that the boxes are checked as in the figure below. You're going to generate a proxy and a test client at the same time that you generate the Web service itself. Remember to check the **Overwrite files without warning** check box as well.

Figure 24. Defining the options

3. Click **Next**.
4. Accept the defaults on the Object Selection Page. Since we started from the Bean the fields are correctly filled to `acpkg.AtomicClock`. Click **Next**.
5. Accept the defaults on the Service Deployment Configuration window. The test client is generated in a new Web project named `AClockWebClient`. You'll use the new WebSphere V6 Web services runtime engine. Click **Next**.
6. Accept the defaults in the Service Endpoint Interface Selection, make sure `acpkg.AtomicClock` is the specified bean and click **Next**.
7. In the Web Service Identity window, uncheck the `main(java.lang.String[])` method as shown in the figure below.

Figure 25. Selecting XML Encryption

8. Select **XML Encryption** as the security configuration and click **Next**.
9. A warning window opens stating that your choices might not comply with WS-I. This is to be expected because WS-Security is not yet part of the WS-I Basic Profile 1.0. Click **Ignore** to proceed.

Figure 26. WS-I Warning

10. Leave the defaults on the Web Service Test Window and click **Next**.
11. Leave the defaults in the Web Service Proxy Page window. Notice that XML Encryption is the default for the client proxy. Click **Next** followed by **Ignore** on the WS-I compliance warning for the client.

Figure 27. Client Proxy Security Options

12. Leave the defaults in the Web Service Client Test window and click **Next**.
13. Leave the defaults in the Web Service Publication window and click **Finish**. The wizard takes time to generate the Web service. After it has completed, it launches your browser with the generated test client for the Web service.
14. Click **getEndpoint()** and **Invoke** in the Result frame. Copy the returned

URL.

Figure 28. Getting the endpoint

15. Using `setEndpoint()`, set the endpoint to `http://localhost:9081/AClockWeb/services/AtomicClock`. This should be the same value you just copied, except with the port changed from 9080 to 9081. Now requests from the JSP-based client are passed through the TCP/IP Monitor server listening on port 9081, and then forwarded on to ACServer listening on port 9080. You should be able to see the SOAP requests and responses as they pass through.
16. Click **getTheFormattedDate()** and enter 3 as a parameter.
17. If you switch to the TCP/IP monitoring server, you can see the request and response messages. We'll look at these messages in more detail next.

Examining the request and response messages

The best way to look at the request and response content is to copy it to WordPad or another text editor. Right-click in either pane and choose **Select All** followed by another right-click and **Copy**. Paste the content into an editor for a better look at it.

Review the `<wsse:Security>` header contents. Here's an abbreviated version of the encrypted SOAP request:

```
<soapenv:Envelope xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Header>
    <wsse:Security xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd" soapenv:mustUnderstand="1">
      <EncryptedKey xmlns="http://www.w3.org/2001/04/xmlenc#">
        <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
        <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
          <wsse:SecurityTokenReference>
            <wsse:KeyIdentifier ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3SubjectKeyIdentifier"
            >/62wXObED7z6clyX7QkvN1thQdY=</wsse:KeyIdentifier>
          </wsse:SecurityTokenReference>
        </ds:KeyInfo>
        <CipherData>
          <CipherValue>pRfT0LBvSRLACOIUvSMLhIEdEhLVHM6wF4ikXXwC2hWEu1AvsFwYNS+lkqXu2NabCcLKW
Ns4ljn4Wlm+JoNTjJm12NwB8FjA/J5533JaWXAq0n60RqnPLFQzE1Bje6bSKinug749lw1NEfDxyj1W3W/j66k+x
DyxSMkcDPoFiyQ=</CipherValue>
        </CipherData>
        <ReferenceList>
          <DataReference URI="#wssecurity_encryption_id_5313834292926976316"/>
        </ReferenceList>
      </EncryptedKey>
    </wsse:Security>
  </soapenv:Header>
  <soapenv:Body>
```

```

<EncryptedData xmlns="http://www.w3.org/2001/04/xmlenc#"
  Id="wssecurity_encryption_id_5313834292926976316"
  Type="http://www.w3.org/2001/04/xmlenc#Content">
  <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#tripleDES-cbc"/>
  <CipherData>
  <CipherValue>YVnjlOQYU3nzzAkebMR01aC00+jhGCW6r72zLlZBWyKzPxRY1vh4zBCFCBbbQnxOP3EAM1
  sxu7n6Q5nQ2/skqqARdhJ5AzBvcqtfGxvGJ80St87GHWj2+oSh1GivGYeDqGpcmmuTqGw8Bki/sRKUg==
  </CipherValue>
  </CipherData>
</EncryptedData>
</soapenv:Body>
</soapenv:Envelope>

```

With XML encryption, the actual key included in the message is a symmetric key, encrypted with the public key of the recipient. A symmetric key is used because symmetric encryption is faster than asymmetric encryption. Thus, the recipient only needs to use its private key to decrypt the symmetric key. Then it can use the symmetric key to decrypt the `<Body>` business data content, which could be much larger than in the example.

The security header is shorter than one that would contain a signature because there is no contained encoded digital certificate. Inside the `<Security>` header, the `<EncryptedKey>` element contains all the information about the symmetric key used to encrypt the business data and about how the key itself is encrypted, along with the encrypted value of the key. The `<KeyInfo>` element contains the `<SecurityTokenReference>`, which in turn contains an encrypted `<KeyIdentifier>` element. `<CipherValue>`, inside `<CipherData>`, contains the symmetric key itself, encrypted. The recipient of this SOAP request message, the service, must be able to decrypt this encrypted key in order to use it to decrypt the `<Body>` content.

The `<ReferenceList>` element contains a list of references to the content that has been encrypted. In this case, there is only one `<DataReference>`, but there can be more than one encrypted block of data using XML encryption. Notice that the URI attribute of the `<DataReference>` points to the `Id` attribute of the `<EncryptedData>` element within the `<Body>`. The actual `<Body>` content, the business data, has been replaced entirely by this `<EncryptedData>` element. The encrypted `<Body>` data is inside the `<CipherValue>` content within `<CipherData>`.

Section 6. Authentication: Adding a security token

The *authentication* process adds a security token to the SOAP message. This security token is inserted into the message to authenticate the caller. There are a

number of possible tokens that can be added to the message: signature, ID assertion, LTPA, custom, or basic authentication. Add basic authentication, which adds a user ID and password into the message. Note that this user ID and password is not encrypted, so it passes in the clear. You would have to add encryption to the token to protect the password. So, in a situation like this, it is advisable to send the message using transport-level security, like SSL.

In this section, you'll see how to add a basic authentication header to a SOAP message. This header does not require that either XML encryption or XML digital signatures be selected during the generation of the Web service. In other words, you can regenerate your Web service to the simple service that you began with, or you can just add basic authentication to the existing service that has encryption added. If you would like to have a simple service, follow the instructions in [Generating a Web service](#) and then come back here. These steps work with either service. Ensure that your Web service is working prior to enabling security.

There are three steps to adding authentication after you've created the Web service:

1. Enable Security on the Server
2. Update the client configuration by editing the `web.xml` file directly
3. Update the server configuration by editing the `webservices.xml` file directly

Enabling security on the server and in your service

To enable security on your server and enable your service to process a security token:

1. On the server panel, double-click on the server to bring up its configuration panel.
Figure 29. Selecting the server for configuration
2. On the Security tab, make sure Enable Security is checked. Enter a user ID and password that is defined on the operating system with the appropriate authority. Save and close the configuration.
Figure 30. Defining the security options
3. Save the server configuration and exit.
4. right-click on the server again and select **Run Administrative console**, select yes to any prompts you receive about certificates.
Figure 31. The Admin console

5. Use any ID, this username is not checked, and click **log in**.
6. Expand **Security** and click **Global Security**.
Figure 32. Global Security
7. Click **Local OS** on the right side and enter a valid user name and password, then click **OK**.
Figure 33. Entering the user details
8. Click **enable global security** and deselect **Enforce Java 2 security** as shown in the figure below. Click **OK**.
Figure 34. Enabling Global Security
9. Save the configuration. Click **Save** on the top of your panel and **Save** again.
10. Restart the server to activate the changes. To check if security is enabled try opening the administrative console. If security is enabled you will be prompted for a userid and a password. This is an easy way to verify that security is enabled.
11. First, configure the Application Server Services. In the Web perspective, expand the `AClockWeb > WebContent > WEB-INF` folder and open the `webservices.xml` file. Use this file to define the security policy for this Web service. When the file opens, you'll see the **Web Services Editor** window.
Figure 35. Web Services Editor
12. Click the Web Services Security Extensions panel. Expand the **Request Consumer Service Configuration Details**.
13. This heading controls how incoming messages (requests) should be processed as they come in from clients. Expand the **Required Security Token** heading. Click **Add** (note that if the Add option is greyed out, select the AtomicClock service under the Port Component Binding tab). Then give the Token a name and select **Username** as the token type. Click **OK**.
Figure 36. Required Security Token
14. Now expand **Caller Part** under the Request Consumer Configuration Details and click **Add**. Add a name (`basicAuth`) for the caller part and select **Username** as the Token type.
Figure 37. Entering a Caller Configuration
15. Click **OK** and then click the Binding Configurations tab. On the right side of this tab, you'll see the Request Consumer Binding Configuration Details

section. Expand this tab.

16. Scroll down to the Token Consumer section and click **Add**.
Figure 38. Defining the Token Consumer
17. Give the Token consumer a name. The Token consumer class is `com.ibm.wsspi.wssecurity.token.UsernameTokenConsumer`. Select the Security token, in your case `UNToken`. Select the Use `jaas.config` checkbox and enter `system.wssecurity.UsernameToken` as the `jaas.config` name.
18. Click **OK** to store these settings.
19. Save and close the configuration file.

Creating and testing an authentication-ready service client

Your service is now deployed with basic authentication. When a SOAP request comes in to the service, the header must contain a username and password. That username must exist on the system hosting the service. The next step is to create a client application that understands how to build a SOAP envelope correctly and use the service.

1. In the Web perspective, expand the **AClockWebClient > WebContent > WEB-INF** folder of your project. You'll see a file named `web.xml`.
Figure 39. Setting the client configuration
2. Open this file. The screen you'll get looks similar to the Web Services Editor you used when you defined the security policy for the Web service on the server. As you'd expect, the steps you go through to define the security policy for the client application are similar to those for the service. Select the WS Extensions tab. On the right side of the panel you'll see the Request Generator Configuration settings. Expand this tab.
3. Click **Add** to add a security Token. Enter a name and select Username as the token type and click **OK**.
Figure 40. Defining the token
4. Switch to the WS Binding Tab in the editor. Expand Request Generator Binding Configuration. Expand the tab. On the right side of this panel, scroll to the Token Generator section.
5. Click **Add** to add a token generator. In the window that appears, add a Token generator name. Leave the Token generator class to the default setting. Set the Security token value to **basicAuth**.

6. For the callback handler, select **NonPromptCallbackHandler** from the drop-down list. You can enter your username and password in this window. Enter a username and password defined for your system. Click **OK**. Save and close the configuration file.

Figure 41. Setting the userid and password

To test the Web service with your newly-defined authentication settings:

1. Restart the server.
2. In the server perspective, click the server name and on the context menu select **Restart**.
3. In the Web perspective, expand **AClockWebClient > Web Content > sample**, right-click **TestClient.jsp**, then choose **Run on Server**.
4. When you run the service, you should see the same results you've seen in earlier demos. Under the covers, of course, the WebSphere infrastructure creates and validates the basic authentication security token for you. To see the actual SOAP envelopes, run them on your server, but redirect the messages to the TCP/IP monitoring server.
5. You should see the following token added to the request message. (The username you'll see will, of course, be whatever you have defined).

```
<wsse:UsernameToken>
<wsse:Username>demouser</wsse:Username>
<wsse:Password Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss
-username-token-profile-1.0#PasswordText">some01spwd</wsse:Password>
</wsse:UsernameToken>
```

6. The SOAP request is in the lower left corner of the tracing window. You can see the `<wsse:UsernameToken>` in the header of the message, with the `<wsse:Username>` and `<wsse:Password>` elements inside it. The response that comes back from the service is a normal SOAP message.

As you can see, setting up a Web service for basic authentication doesn't take much configuration.

Section 7. Summary

Security is an essential part of any Web service. One option to secure Web services is to create a custom security model and build it into your application, but doing so requires an extensive amount of implementation and maintenance work. As an alternative, Rational Application Developer allows you to take advantage of security standards and without much effort create all of the necessary configuration to add security to your services. This tutorial has shown you how to authenticate using a user ID and password, ensure integrity using digital signatures, and ensure confidentiality using encryption.

Downloads

Description	Name	Size	Download method
The workspace used in this tutorial.	AtomicClock.java	2KB	HTTP

[Information about download methods](#)

Resources

Learn

- For more on adding security to Web services by editing the deployment descriptor, check out the IBM Redbook [WebSphere Version 6 Web Services Handbook Development and Deployment](#).
- Take a look at the [WS-Security](#) specification.
- Check out the joint IBM-Microsoft Whitepaper [Security in a Web services world: A proposed architecture and roadmap](#).

Get products and technologies

- Download [AtomicClock.java](#), the sample code file that accompanies this tutorial.
- Download an evaluation version of [Rational Application Developer 6.0](#).

Discuss

- [Participate in the discussion forum for this content](#).

About the authors

Indran Naick

Indran Naick is an e-business architect for IBM Developer Relations Technical Consulting in Austin, Texas, which provides education, enablement, and consulting to IBM business partners. Indran has over 14 years of industry experience. He joined IBM in South Africa in 1990. Prior to being transferred to Austin, he served as a software solutions architect, consulting to a number of financial and government institutions. He has authored a number of publications and is a graduate of the University of the Witwatersrand in South Africa.

Jeff Miller

Jeff Miller is a software consultant with IBM ISV and Developer Relations Worldwide Developer Skills program. He has over 24 years of software development experience as an electrical engineer, software developer and architect. His focus at IBM is Java EE application architecture, design, development, Web services, SOA and security. Jeff consults, codes, teaches, writes technical articles and speaks to universities and groups. He is an IBM-certified On Demand Solution

Designer and Solution Technologist, an IBM Certified Solution Designer -- Service Oriented Architecture, and is IBM-certified on Rational Application Developer and WebSphere Application Server. Jeff is a CompTIA Security+ Certified Professional. He received his Masters degree in Computer Science from Rensselaer Polytechnic Institute.