IBM

developerWorks

Building a secure SOAP client for J2ME, Part 1: Exploring Web Services APIs (WSA) for J2ME

Integrating security components into WSA

Skill Level: Intermediate

Bilal Siddiqui
Freelance consultant

16 Jun 2006

In this three-part tutorial series you'll learn how to build a secure Web services client based on Java™ 2, Micro edition (J2ME). This first part introduces application scenarios which allow wireless access to your Web services. Part 1 also discusses the architecture of a secure Web service application and explains the way different technology components collaborate with each other in a J2ME application to provide security features. We also explore Web Services APIs (WSA) for J2ME in detail by digging deep inside a couple of WSA applications. Later parts of this tutorial will expand upon these concepts to incorporate security into WSA applications.

Section 1. Before you start

About this tutorial series

This series demonstrates how to incorporate security in Java™ 2, Micro edition (J2ME)-based wireless access to Web services. We use the following components and technologies together in a J2ME MIDlet:

1. Web Services APIs (WSA) for J2ME
2. Cryptography

© Copyright IBM Corporation 2006. All rights reserved.

3. XML Digital Signature
4. Java Card

First you'll see several application scenarios where you will need to incorporate security in wireless access for your Web services.

WSA uses the idea of stub classes, so other technology components such as cryptography, XML signatures and Java Card technology have to fit into WSA stub classes. Therefore, we will explore how WSA stub classes work and demonstrate how other technology components cooperate with WSA.

This tutorial series also demonstrates various testing and debugging arrangements that you can use to integrate different technology components. The series concludes by putting together all the concepts into a "stub enhancer tool". This tool will enhance functionality of WSA stub classes by incorporating security features.

About this tutorial

This first part of the series introduces the concept of integrating various Internet technologies to build secure client-side Web service applications in J2ME.

We provide sample application scenarios and a comprehensive architectural discussion on how different technology components work together to build a secure Web service client.

We also present a graphical image of different modules in the security architecture, and identify the role of each module.

The architectural discussion follows an analysis of WSA stub classes. This is incremental; we first explore stub classes for a simple Web service and move to more comprehensive Web services.

This tutorial concludes by introducing the interface of a secure Web service. In forthcoming parts of this series we implement security features.

Prerequisites

As this tutorial is all about integrating various technology components, it's important for you to have a basic understanding of the components. Specifically, it is assumed readers have the following background:

1. You should be a Java programmer and also have a basic understanding

© Copyright IBM Corporation 2006. All rights reserved.

of J2ME MIDlets.

2. WSA uses Web Services Definition Language (WSDL) and Simple Object Access Protocol (SOAP). Therefore, you need to know how WSDL interfaces are mapped to SOAP method invocation calls.

3. You also need to know the basics of W3C's XML Schema, especially the use of `xsd:element` and `xsd:complexType`.

Moreover, some background on XML signatures will be useful.

IBM developerWorks has many excellent articles and tutorials about these topics. The Resources section lists some for ready reference.

Should I take this tutorial?

This tutorial will guide you in planning to securely enable wireless access to your Web services.

This tutorial also contains value for you if you wish to implement security in non-wireless access to your Web services.

Tutorial topics

Part 1 is organized in the following seven sections:

1. Tutorial introduction

2. Sample application scenarios that require wireless access to Web services. This section also introduces WSA architecture and stub classes

3. Explanation of why you need to secure wireless access to your Web services. This section also includes a demonstration of how different technology components work together to provide security

4. Detailed analysis of a simple client-side WSA application

5. More analysis of a simple client-side WSA application

6. Discussion of a more comprehensive WSA application. This discussion provides all details about the working of WSA stub and other classes, which you need to know in order to start incorporating security into WSA

7. Wrap-up

© Copyright IBM Corporation 2006. All rights reserved.

Code samples and installation requirements

We used J2ME Wireless Toolkit version 2.2, to generate and try the code for Part 1.

Later sections of this tutorial will also need the following software tools, which are all free downloads (see Resources).

1. Sun Java Wireless Toolkit version 2.3 Beta. We use version 2.2 in Part 1, primarily because version 2.3 is in beta at the time of writing Part 1. But we will try the code presented in later parts of the tutorial on both versions 2.2 and 2.3.
2. XML Security Suite for Java (XSS4J) from IBM alphaWorks.
3. Java Card Development Kit from the Sun Web site.

Section 2. Wireless access to Web services

Wireless clients consuming Web services

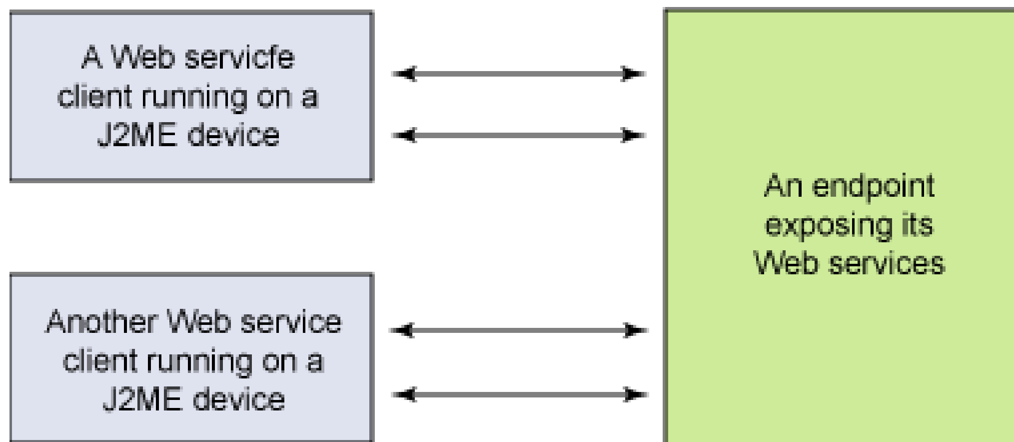
The Java Specification Request 172 (JSR 172) defines a standardized API that J2ME clients can use to invoke SOAP and XML-based Web services. This API is in the form of an optional package for J2ME, and is referred to as Web Services APIs (WSA) for J2ME. WSA is actually a subset of the Java API for XML-based Remote Procedure Call (JAX-RPC) defined by JSR 101.

JAX-RPC uses the popular concept of Web service "endpoints" and "clients". Endpoints expose Web services and JAX-RPC clients invoke -- access, consume or make use of -- the services exposed by endpoints.

WSA, as a subset of JAX-RPC, only includes the set of interfaces that are used to define Web service clients. This makes sense because J2ME devices are not likely to expose their own Web service endpoints. J2ME devices are only expected to consume Web services exposed by service endpoints. This scenario is depicted in Figure 1.

Figure 1: Web service clients consuming services exposed by a Web service endpoint

© Copyright IBM Corporation 2006. All rights reserved.



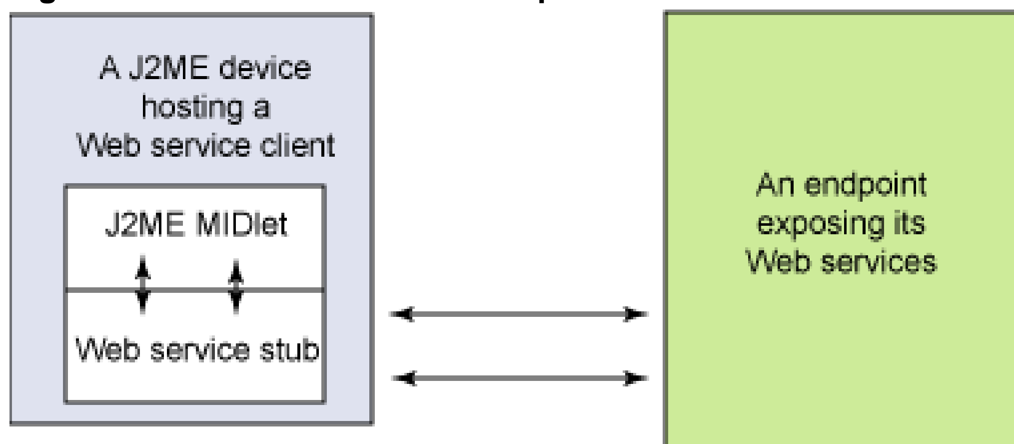
In the Sample Web service application scenarios section, we provide example application scenarios where you will need to expose your Web services to be consumed by J2ME clients.

Using WSDL to generate stub classes

Both JAX-RPC and WSA use Web Services Definition Language (WSDL) to define a Web service interface and generate its client side stub. A Web service stub is a set of classes that acts as a local agent, or proxy, of the actual Web service endpoint.

A MIDlet application will use stub classes to invoke the remote Web service. The MIDlet to stub interaction will be local and happen within the J2ME device. Stub classes will handle all communication with the remote Web service endpoint. This is shown in Figure 2.

Figure 2: MIDlet-stub and stub-endpoint interactions



The J2ME Wireless Toolkit version 2.2 from Sun (see Resources) comes with a stub generator tool that takes a WSDL file and generates a set of stub classes for the Web service defined by the WSDL file.

© Copyright IBM Corporation 2006. All rights reserved.

We refer to this set of classes as *WSA stub classes*.

The Resources section has a link to the developerWorks article "Web Services APIs for J2ME, Part 1: Remote service invocation API." You can refer to this article to learn the architecture of JSR 172 as well as the generation and use of WSA stub classes to consume Web services.

Another developerWorks article, "Designing mobile Web services" (see Resources) discusses server side issues in designing Web services that are meant to be consumed wirelessly. You can refer to the article to learn why you need to plan your Web services specifically for consumption by wireless clients.

How wireless access to Web services works

Following are a few important points about how wireless access to Web services works.

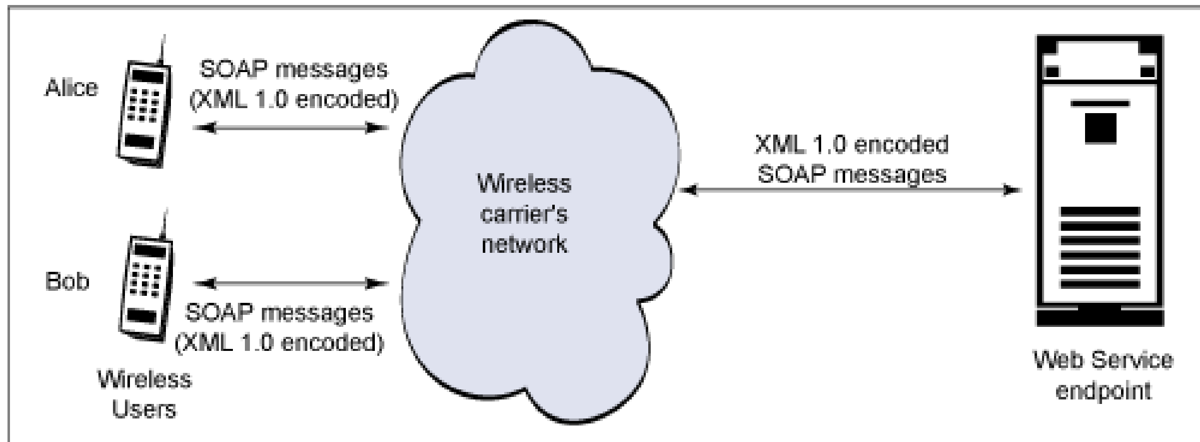
All JSR172-compliant devices are required to support Simple Object Access Protocol (SOAP) version 1.1. However, note that JSR 172 does not require a J2ME device to support SOAP encoding (or representation) according to XML 1.0. This means JSR 172 devices may or may not understand XML1.0 representation of SOAP messages.

If a JSR 172-compliant wireless device does not implement XML1.0 encoding, it may use some other encoding as defined in the XML Information Set (Infoset, see Resources). In this case it is the responsibility of the wireless carrier's network to transform messages from other encoding to XML and back, so that messages reaching the Web service endpoint are interoperable according to XML format.

This means the service endpoint will only see SOAP/XML coming from WSA devices. This scenario is depicted in Figure 3.

Figure 3: A wireless carrier's network ensures XML encoding of SOAP messages

© Copyright IBM Corporation 2006. All rights reserved.



This also means that even in the case where a WSA implementation in a J2ME device does not support XML 1.0, it must produce a SOAP representation that can be transformed into interoperable XML 1.0 representation.

Let's look at some sample application scenarios in which you would like your Web service to be consumed by J2ME clients. These scenarios will also help you understand why you will need security while accessing a Web service.

Sample Web service application scenarios

There are many application scenarios where you will need to expose your Web services to wireless clients. Let's look at a set of such Web services that allow J2ME device users to order small items using their cell phones.

Suppose you have implemented a set of Web services that allows users to do the following:

1. Book a hotel room in a city a user is going to visit
2. Check for important email messages, such as a message from a particular sender
3. Order food with any of a number of local restaurants
4. Order a cab service
5. Make a payment installment
6. Call for an emergency vehicle repair or ambulance service to the user's current location

Your clients may need such services any time any day, so we'll call them "Everyday

© Copyright IBM Corporation 2006. All rights reserved.

Web services".

In order to provide Everday Web services, you will need to make arrangements with many service providers in different locations. For example, you will need partner cab services operating in different physical locations. Similarly, you will need partner hotels and restaurants in different cities.

You will design Web service endpoints for each of the service included in your Everday Web services. Your partner companies such as cab services, restaurants, and hotels, will implement and host the endpoints you design.

As you can easily see from the nature of these services, users will get a real value from such Web services only if they are able to access the service wirelessly.

Therefore, you will also provide a J2ME application to your customers the users of Everday Web services. The J2ME application will run on the cell phones of your customers, allowing them mobile access to the services of your partners. I will call this J2ME application "Everyday MIDlet".

We need to have a variety of Web service application scenarios to demonstrate the design, development, implementation, and testing of secure mobile access to Web services. Your "Everyday Web services" example provides the required variety of Web services.

Therefore we will use Everyday Web services as sample application scenarios throughout this tutorial.

We start by showing SOAP messages that your Everyday MIDlet will send to Web service endpoints.

SOAP messages for the Everyday Web service

> **A note about WSDL and SOAP terminology**
> In WSDL terminology a "SOAP method invocation call" is an "operation". We use the terms "method invocation call" and "operation" interchangeably in this tutorial.

Suppose one of your customers, Alice, is on her way to a nearby city and wants to book a room in a hotel. She will use the Everyday MIDlet to send SOAP messages to Everyday Web service endpoint in order to find a suitable room.

For example, she may want to first check for hotels in a specific area with rooms available in a specific range of rent. Once she knows the hotels with rooms available, she may want to choose and reserve a room.

© Copyright IBM Corporation 2006. All rights reserved.

Look at Listing 1, which shows the SOAP message that Alice will send to check for room availability in a specific area in a specific rent range:

Listing 1: Alice's SOAP message that checks for room availability

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:tns="http://www.everdaywebservices.com/hotelservice">
<soap:Body>
<tns:checkForRoomAvailability>
   <roomRequired>
      <city>Lahore</city>
      <area>Gulberg</area>
      <maxRent>2500</maxRent>
   </roomRequired>
</tns:checkForRoomAvailability>
</soap:Body>
</soap:Envelope>
```

Note that the SOAP message of Listing 1 contains a method invocation call to a remote method named `checkForRoomAvailability`.

The `checkForRoomAvailability` method invocation call contains a `roomRequired` element containing three child elements, namely:

1. `city` (the city in which Alice is looking for a hotel room),
2. `area` (name of the area in which Alice would like to stay), and
3. `maxRent` (the maximum price that Alice is willing to pay).

Listing 2 shows response from the Everyday Web service.

Listing 2 Response to Alice's checkForRoomAvailability call

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:tns="http://www.everdaywebservices.com/hotelservice">
<soap:Body>
<tns:checkForRoomAvailabilityResponse>
   <tns:roomsAvailable>
      <room>
        <hotelName>Hotel1</hotelName>
        <location>Gulberg 2</location>
        <roomType>BusinessClass</roomType>
        <roomRent>2500</roomRent>
      </room>
      <room>
        <hotelName>Hotel2</hotelName>
        <location>Gulberg 5</location>
        <roomType>Economy</roomType>
```


© Copyright IBM Corporation 2006. All rights reserved.

```
        <roomRent>2000</roomRent>
      </room>
      <room>
        <hotelName>Hotel3</hotelName>
        <location>Gulberg 3</location>
        <roomType>Economy</roomType>
        <roomRent>1800</roomRent>
      </room>
   </tns:roomsAvailable>
</tns:checkForRoomAvailabilityResponse>
</soap:Body>
</soap:Envelope>
```

Note `roomsAvailable` element in Listing 2 contains a number of `room` elements. Each `room` element contains `hotelName`, `location`, `roomType`, and `roomRent` elements.

I will call this hotel-related Everday Web service as "Everday hotel service" or simply "hotel service".

Some more SOAP messages for the Everyday hotel service

Look at Listing 3, which shows a SOAP invocation call to a method named `reserve`.

Listing 3: The reserve SOAP method invocation call

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:tns="http://www.everdaywebservices.com/hotelservice">
<soap:Body>
<tns:reserve>
   <room>
    <hotelName>Hotel3</hotelName>
    <location>Gulberg 3</location>
    <roomType>Economy</roomType>
    <roomRent>1800</roomRent>
  </room>
</tns:reserve>
</soap:Body>
</soap:Envelope>
```

The `reserve` method takes a `room` element and returns the booking status, as shown in Listing 4.

Listing 4: Booking status returned in response to the reserve method call

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```


© Copyright IBM Corporation 2006. All rights reserved.

```
     xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:tns="http://www.everdaywebservices.com/hotelservice">
<soap:Body>
<tns:reserveResponse>
  <bookingStatus>Confirmed</bookingStatus>
</tns:reserveResponse>
</soap:Body>
</soap:Envelope>
```

WSDL definition for the Everyday hotel services

Listing 5 shows the WSDL definition of the Everyday hotel service comprising of two operations (`checkForRoomAvailability` and `reserve`).

Listing 5: WSDL for the hotel service

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions
     xmlns="http://schemas.xmlsoap.org/wsdl/"
     xmlns:tns="http://www.everydaywebservices.com/hotelservice"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
     name="hotelService"
     targetNamespace="http://www.everydaywebservices.com/hotelservice">

    <types>
     <schema xmlns="http://www.w3.org/2001/XMLSchema"
         targetNamespace="http://www.everydaywebservices.com/hotelservice">
       <xsd:element name="roomRequired">
         <xsd:complexType>
            <xsd:sequence>
             <xsd:element name="city" type="xsd:string"/>
             <xsd:element name="area" type="xsd:string"/>
             <xsd:element name="maxRent" type="xsd:int"/>
            </xsd:sequence>
         </xsd:complexType>
       </xsd:element>

       <xsd:element name="room">
         <xsd:complexType>
             <xsd:sequence>
               <xsd:element name="hotelName" type="xsd:string"/>
               <xsd:element name="location" type="xsd:string"/>
               <xsd:element name="type" type="xsd:int"/>
               <xsd:element name="rent" type="xsd:int"/>
             </xsd:sequence>
         </xsd:complexType>
       </xsd:element>

       <xsd:element name="roomsAvailable">
         <xsd:complexType>
          <xsd:sequence>
             <xsd:element name="room" ref="tns:room" minOccurs="0"
             maxOccurs="unbounded"/>
          </xsd:sequence>
         </xsd:complexType>
       </xsd:element>
       <xsd:element name="bookingStatus" type="xsd:string"/>
     </schema>
    </types>
```


© Copyright IBM Corporation 2006. All rights reserved.

```
<message name="roomRequired">
  <part name="roomRequired" element="tns:roomRequired"/>
</message>

<message name="roomsAvailable">
  <part name="roomsAvailable" element="tns:roomsAvailable"/>
</message>

<message name="room">
  <part name="room" element="tns:room"/>
</message>

<message name="bookingStatus">
  <part name="bookingStatus" element="tns:bookingStatus"/>
</message>

<portType name="hotelService">
   <operation name="checkForRoomAvailability">
    <documentation>
      Check room availability by city, location and rent.
    </documentation>
    <input message="tns:roomRequired"/>
    <output message="tns:roomsAvailable"/>
  </operation>
  <operation name="reserve">
    <documentation>
      Reserve a room.
    </documentation>
    <input message="tns:room"/>
    <output message="tns:bookingStatus"/>
  </operation>
</portType>

<binding name="hotelServiceBinding" type="tns:hotelService">
 <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
 <operation name="checkForRoomAvailability">
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
    <soap:operation
 soapAction="http://www.everdaywebservices.com/hotelservice/checkForRoomAvailability"/>
  </operation>
 <operation name="reserve">
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
    <soap:operation
       soapAction="http://www.everdaywebservices.com/hotelservice/reserve"/>
  </operation>
  </binding>
   <service name="HotelService">
   <port name="hotelServicePort"
        binding="tns:hotelServiceBinding">
    <soap:address location="http://www.everdaywebservices.com/hotelService"/>
  </port>
</service>
</definitions>
```

Without going into the details of WSDL format used in Listing 5, we wish to point out that Listing 5 contains four important WSDL components:

© Copyright IBM Corporation 2006. All rights reserved.

1. Data types defined for the hotel service (`types` element in Listing 5),
2. Input and output messages (`message` elements in Listing 5),
3. Port type definitions (`portType` element in Listing 5), and
4. Port type bindings with the transport mechanism (`binding` element in Listing 5) and
5. Details of the endpoint (`service` element in Listing 5)

WSDL components of the hotel Web service

Here is a brief description of the five WSDL components of the hotel service.

The types element in Listing 5 defines the data types to be used in the hotel service. Note the complex type definition inside the `xsd:element` named `room`, which provides a simple description of a hotel room. It contains four child elements namely `hotelName`, `location`, `type`, and `rent`.

The `hotelName` element specifies name of hotel. The location element specifies the physical location of hotel in the form of a textual string. The `type` and `rent` elements specify type and rent of the room, respectively.

The `message` elements in Listing 5 define the messages (the `checkForRoomAvailability` SOAP message that you saw earlier in the SOAP message of Listing 1, that travel across the Internet to invoke methods of the remote service.

Note that each `message` element in Listing 5 has a `part` child element. A `part` element defines a part of the message. For the sake of simplicity, we use single-part messages in this tutorial. So you will see one `part` child element in all `message` elements.

Each `part` element refers to a data type definition in the `types` element using its `element` attribute. For example, look at the `part` element in Listing 5, whose `element` attribute has value `roomRequired`. This value matches with the `name` attribute of an xsd:element within the `types` element.

The `portType` element in Listing 5 defines the port type of the hotel service. A port type definition specifies an abstract interface of your Web service, consisting of operations included in the Web service. For example, the hotel service WSDL shown in Listing 5 consists of two operations (`checkForRoomAvailablity` and `reserve`). You can find two operation children of the `portType` element in Listing 5, one for the `checkForRoomAvailability` operation and the other for the

© Copyright IBM Corporation 2006. All rights reserved.

`reserve` operation.

The `binding` element in Listing 5 defines message encoding and transport protocol, SOAP, for the Web service. I will be exclusively using SOAP binding for all Everyday Web services discussed in this tutorial.

Look in particular at the `soapAction` attribute of `soap:operation element` (which is a grandchild of the `binding` element in Listing 5). `soapAction` is a SOAP-specific attribute, which specifies how a SOAP server will invoke the SOAP method.

The service element in Listing 5 defines the service endpoint hosting the remote Web service.

Note that the same Web service is normally hosted by different endpoints providing the same service, that is, you may have different partner companies providing hotel services in different cities.

This means different service endpoints will host your port type and each hosted service will provide the details of its own service.

For example, the `service` element in Listing 5 has a grand child named `soap:address`. The `soap:address` element has an attribute named `location`, which specifies the network address of the endpoint. Each service endpoint will specify its own `location` attribute value.

Later in Exploring stub classes section, we will relate WSDL components to WSA stub classes, showing how stub classes represent a client-side implementation of a Web service.

You have seen the SOAP format and WSDL definition for the hotel service. The hotel service is not yet secure. The next section will discuss security in the hotel service.

Section 3. Securing wireless access to Web services

Why need security in the hotel service?

You have seen two operations, `checkForRoomAvailability` and `reserve`, in the hotel service of Listing 5.

© Copyright IBM Corporation 2006. All rights reserved.

You may want to allow the general public to call the `checkForRoomAvailability` method of your hotel service. This means anyone among the public will be able to check for room availability in any area at any time, without paying anything for your hotel service.

Therefore, the `checkForRoomAvailability` operation does not require any security, such as authentication and authorization.

On the other hand, the reserve operation does require authenticating a requesting client before reserving a room on the client's behalf. That's because you need to charge for the hotel reservation and you cannot charge without ensuring that your customer really asked for the service.

This means you need an authentication mechanism to identify the client who is requesting the reservation service. This is to ensure that no one else, such as a hacker, has access to the service when your server thinks the service is being provided to one of your customers.

We use digital signatures as an authentication mechanism in this tutorial.

In the next section we go over the cryptographic options available in using digital signatures in order to secure wireless access to your Web services. After that, we present an enhanced version of the "reserve" SOAP message that you saw in Listing 5. We explain the enhancements that allow incorporation of authentication data in the SOAP message.

Next, we explain how different wireless and security-related technologies cooperate with each other to build a secure Web service client for J2ME. We also list some limitations in J2ME that you have to consider before starting to build a secure J2ME client for your Web services.

Public and secret key cryptography

You can use either public or secret key cryptography to incorporate authentication logic in your SOAP request messages.

Secret key cryptography uses the concept of a shared secret between communicating parties. Kerberos, a popular security protocol, defines a mechanism to securely exchange the shared secret.

IBM developerWorks has some excellent resources that describe the use of Kerberos secret keys in J2ME applications. Check out the Resources section to learn more about using Kerberos in your J2ME MIDlets.

Public key cryptography does not require sharing a secret. Instead, it allows you to generate a pair of keys called private and public keys.

© Copyright IBM Corporation 2006. All rights reserved.

You will keep your private key secure with you and should not give it to anyone. On the other hand, your public key becomes available for every one -- such as your friends, family, and colleagues.

You will use your private key to produce a cryptographic signature, which your friends will verify using your public key. If a signature over a message can be cryptographically verified using your public key, it means the message was really sent by you.

Whether you are using secret keys or public keys, you can use the security architecture that is presented in this tutorial.

Using Java Card technology

We use a combination of several technologies with WSA in this series of tutorials. One of the technologies is Java Card. The Java Card technology provides a means of securely storing confidential information, private and secret keys.

Java Card tehcnology type of smart card, which are a highly enhanced version of magnetic strip cards with several thousand times more capacity of storing information. Smart cards can also contain computing and processing logic -- the logic to compute a digital signature. The Resources section offers links that discuss smart cards and Java Cards.

Java Card tehcnology a type of smart card, which are a highly enhanced version of magnetic strip cards with several thousand times more capacity of storing information. Smart cards can also contain computing and processing logic -- the logic to compute a digital signature. The Resources section offers links that discuss smart cards and Java Cards.

We use the Java Card technology for two purposes in this tutorial:

1. to securely store a cryptographic key
2. to implement signature calculation algorithm

Here's how we intend to use the Java Card technology.

Suppose Alice is a Java Card user, who wants to produce a cryptographic signature using the Java Card. She will need to provide a username and password while accessing her Java Card.

The username-password pair will work with only Alice's Java Card. Therefore, if a hacker steals her Java Card, the hacker will also need to know the username-password pair for that specific Java Card.

© Copyright IBM Corporation 2006. All rights reserved.

This type of security is sometimes called dual factor security, which is demonstrated in a developerWorks article on how to integrate Java Card technology into J2ME MIDlets using SATSA (see Resources.)

In order to communicate with a Java Card, a J2ME device will need an API called Security and Trust Services API (SATSA). In the third tutorial of this series, we demonstrate how WSA stub classes use SATSA to talk to a Java Card application.

Using XML signatures

Another important technology in this tutorial is XML signatures. We use XML signatures to wrap a cryptographic signature within an XML message. For this purpose, we will use the XML Signature specification by W3C (see Resources.)

Listing 6 shows an enhanced version of the reserve method invocation request that you saw in Listing 3.

Listing 6: The signed version of the reserve SOAP request

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:tns="http://www.everydaywebservices.com/hotelservice">
<soap:Body>
<tns:reserve Id="R1">
 <room>
    <hotelName>Hotel3</hotelName>
    <location>Gulberg 3</location>
    <roomType>Economy</roomType>
    <roomRent>1800</roomRent>
 </room>
</tns:reserve>
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
 <SignedInfo>
   <CanonicalizationMethod
     Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315">
    </CanonicalizationMethod>
    <SignatureMethod
     Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1">
 </SignatureMethod>
 <Reference URI="#R1">
    <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"></DigestMethod>
    <DigestValue> gmwIRIS+Od1t7+kBmCEsflI4I3U=</DigestValue>
   </Reference>
 </SignedInfo>
 <SignatureValue>
    VfWjfJqN7AV78v+Ye8B3qACRmhzC+FCXZTp78DJpn1bP5hUSimMHwgh
    Wf/F0M9D1PoLZUN6mG131MiDDfSCVQo/4zEo9aw+seyfAGh0rvxQ/3x+LCz9c5pEfmy+Um1/1I+EBOrT
    NxFPMyykBq3JhRoRt3VZIwGzRxkjq+cWOR4c=
 </SignatureValue>
 <KeyInfo>
   <KeyName>Alice</KeyName>
  </KeyInfo>
</Signature>
</soap:Body>
</soap:Envelope>
```


© Copyright IBM Corporation 2006. All rights reserved.

If you compare Listing 3 with its enhanced version of Listing 6, you will find that the reserve method in Listing 6 contains a `Signature` element. The `Signature` element in Listing 6 has three child elements:

1. `SignedInfo` child of `Signature` element specifies the algorithms that were used to produce the cryptographic signature. It also contains a reference to the data that was signed to produce the XML signature.
2. `SignatureValue` child specifies the actual cryptographic signature value.
3. `KeyInfo` element has a `KeyName` child, which specifies the key that was used to produce the signature value.

For the purposes of this tutorial, we focus on demonstrating how to secure wireless access to Web services instead of going into details of XML signatures. IBM developerWorks already has many resources that describe XML signature format in detail. The Resources section of this tutorial offers further details on XML signatures.

Especially you may refer to "Data wrapped inside an XML signature" section of the tutorial Secure XML messaging with JMS, Part 1 (developerWorks, November, 2005) which addresses the details of each of the three children of Signature element.

An important limitation of WSA

As mentioned in Wireless clients consuming Web services section, JSR 172 has defined WSA as a subset of JAX-RPC. While defining WSA functionality, JSR 172 keeps in mind limited computation resources of a J2ME device. Therefore JSR 172 has excluded some of the JAX-RPC functionality.

One important limitation of WSA that you have to face while securing wireless access to your Web services is the inability to author XML attributes.

In Instantiating elements of the simple email service section, we demonstrate that WSA can not author any XML attributes. This means you can not include XML attributes in your SOAP request message.

On the other hand, the Signature element of Listing 6 according to the XML Signature specification includes several XML attributes, such as the `URI` attribute of the `Reference` element in Listing 6. This means it is impossible to author a `Signature` element exactly according to XML Signature specification using the current version of WSA.

© Copyright IBM Corporation 2006. All rights reserved.

Overcoming WSA limitation

Listing 7 shows an edited version of the earlier SOAP message of Listing 6. The only difference between Listing 6 and Listing 7 is that we have changed attribute nodes of Listing 6 with content or element nodes.

Listing 7: A SOAP request with a Signature element that WSA can author

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:tns="http://www.everydaywebservices.com/hotelservice">
<soap:Body>
<tns:reserve Id="R1">
   <room>
    <hotelName>HotelGulberg</hotelName>
    <location>GulbergIII</location>
    <roomType>Economy</roomType>
    <roomRent>1800</roomRent>
 </room>
</tns:reserve>
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
<SignedInfo>
  <CanonicalizationMethod>
     http://www.w3.org/TR/2001/REC-xml-c14n-20010315
  </CanonicalizationMethod>
  <SignatureMethod>
    http://www.w3.org/2000/09/xmldsig#rsa-sha1
 </SignatureMethod>
 <Reference>
     <DigestMethod>
        http://www.w3.org/2000/09/xmldsig#sha1
     </DigestMethod>
     <DigstValue xmlns="">gmwIRIS+Od1t7+kBmCEsflI4I3U=</DigstValue>
     <URI>#R1</URI>
  </Reference>
</SignedInfo>
<SignatureValue>
    VfWjfJqN7AV78v+Ye8B3qACRmhzC+FCXZTp78DJpn1bP5hUSimMHwgh
    Wf/F0M9D1PoLZUN6mG131MiDDfSCVQo/4zEo9aw+seyfAGh0rvxQ/3x+LCz9c5pEfmy+Um1/1I+EBOrT
    NxFPMyykBq3JhRoRt3VZIwGzRxkjq+cWOR4c=
</SignatureValue>
<KeyInfo>
  <KeyName>userName</KeyName>
</KeyInfo>
</Signature>
</soap:Body>
</soap:Envelope>
```

Note that it is still possible to process the `Signature` element of Listing 7 with a standard XML signature processing tool on the server side. This will require a simple XML transformation.

In the second and third tutorials of this series we demonstrate how to use XML Security Suite for Java (XSS4J), an open source XML signature processing and authoring tool by IBM alphaWorks, to process and verify the signatures authored by a WSA client.

© Copyright IBM Corporation 2006. All rights reserved.

Steps to WSA security

Having discussed the different technology components (WSA, cryptography, XML signature and Java Card) that we use in this series of tutorials, we will now put together all the components.

Figure 4 shows the following steps:

1. Suppose Alice, a hotel service user, wants to access the hotel service to reserve a room. Alice's J2ME cell phone has a secure Everyday MIDlet running. So she will invoke the MIDlet.

2. The MIDlet hands over the request to a set of enhanced stub classes that you build in this series of tutorials. As shown in Figure 4, enhanced stub classes use four technologies -- XML signature, Cryptography, SATSA/Java Card, and WSA -- to author a secure reserve SOAP request that wraps user authentication data.

3. Enhanced stub classes use cryptography and SATSA to fetch all cryptographic support required by the secure SOAP request.

4. SATSA, in turn uses a Java Card application to compute cryptographic signature value over Alice's SOAP request.

5. Next, enhanced stub classes use the XML signature support to author a complete XML signature and wrap the signature in the SOAP request.

6. Enhanced stub classes uses WSA framework to send the request over to the remote Web service endpoint implementation.

7. The remote Web service implementation will need to transform (as discussed in Overcoming WSA limitation section) the incoming reserve SOAP request before processing. A transformation module hosted in the remote Web service will do the job. We build this transformation module in the third tutorial of this series.

8. The transformation module will hand over the request to the actual Web service implementation.

9. The Web service will perform reservation.

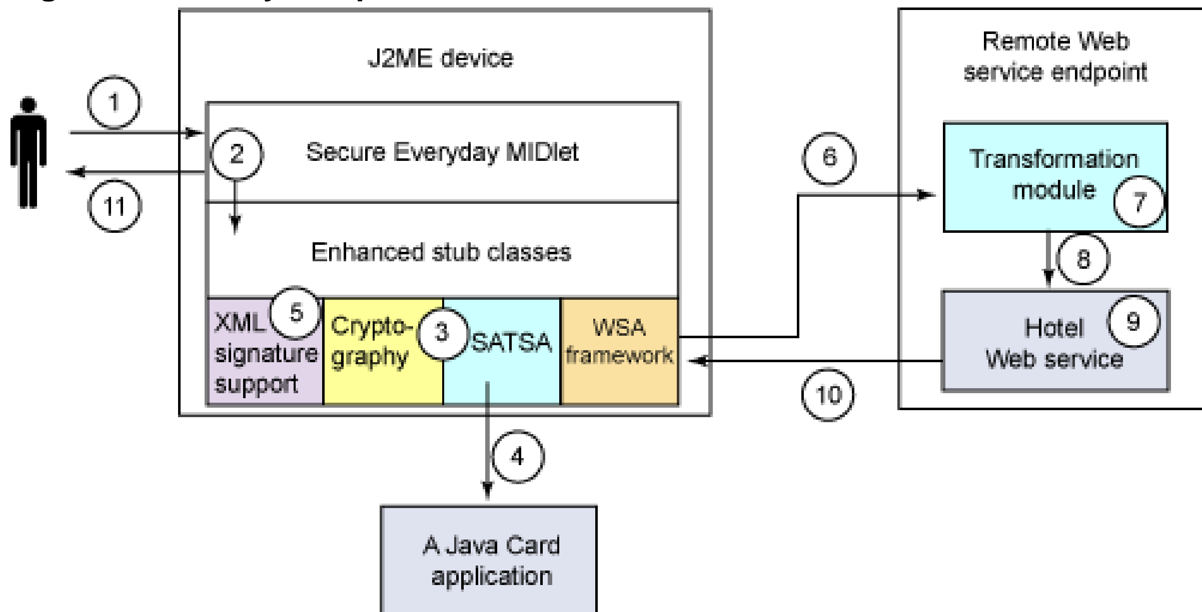
10. The web service will send reservation status in the form of a SOAP response back to the WSA framework.

11. The WSA will process the SOAP response, extract the reservation status

© Copyright IBM Corporation 2006. All rights reserved.

and display the same to the Alice.

Figure 4: Security components for wireless access to Web services



Objectives of enhancing stub classes

In future installments of this series, we cover the following two points:

1. Exploring what's contained in WSA stub classes generated by the stub generator tool
2. How to extend and enhance the stub classes to incorporate security

However, before we start to build the set of enhanced stub classes, We would like to make a couple of important points:

1. You should be able to transform the authentication code in a secure XML message (the Signature element of Listing 7) directly into standard W3C XML signature format (the Signature element of Listing 6). This means, standard signature processing tools should be able to process your XML signatures.
2. You should not make any assumptions about the J2ME device, such as that it contains any proprietary API, except that the J2ME device contains implementation of WSA and SATSA. This means your enhanced set of stub classes should work on any J2ME device with WSA and SATSA.

© Copyright IBM Corporation 2006. All rights reserved.

With these points in mind, we'll start exploring stub classes, so we can enhance them, in the next section.

Section 4. Exploring stub classes

A simple WSDL file

We used hotel service as a sample application in the previous section to explain the usage model and architecture of client-side Web service applications in J2ME. Now we will explore the implementation details of developing secure Web service client applications in J2ME.

Recall from the Using WSDL to generate stub classes section that WSA uses the idea of stub classes generated from a WSDL file to invoke a Web service.

In this section, we use a stub generator tool that comes with J2ME Wireless Toolkit Version 2.2 to generate stub classes for a simple WSDL file. We then explore the stub classes, so that later in the second tutorial of this series, you can see how to enhance the stub classes to incorporate security.

First we will explore stub classes for a simple WSDL file, which does not have complicated data types and multiple operations. This will help explain how stub classes work. Therefore, in this section we will consider WSDL for a simple email service (Listing 8) to generate and discuss stub classes.

We'll call this service as *simple mobile email service* or just *simple email service*.

In the third tutorial of this series, we revisit the hotel service and provide its secure client-side J2ME implementation.

Listing 8: WSDL for the simple email service

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns="http://www.everydaywebservices.com/emailservice"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  name="emailService"
  targetNamespace="http://www.everydaywebservices.com/emailservice">

  <types>
   <schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.everydaywebservices.com/emailservice">
```


© Copyright IBM Corporation 2006. All rights reserved.

```
    <xsd:element name="subjectsList">
        <xsd:complexType>
         <xsd:sequence>
           <xsd:element name="subjectLine" type="xsd:string" minOccurs="0"
               maxOccurs="unbounded"/>
         </xsd:sequence>
       </xsd:complexType>
     </xsd:element>
     <xsd:element name="senderEmailAddress" type="xsd:string"/>
   </schema>
  </types>

  <message name="senderEmailAddress">
    <part name="senderEmailAddress" element="tns:senderEmailAddress"/>
  </message>
  <message name="subjectsList">
    <part name="subjectsList" element="tns:subjectsList"/>
  </message>

  <portType name="simpleMobileEmailService">
    <operation name="getSubjects">
      <documentation>
        Get subject lines of emails from a particular sender.
      </documentation>
      <input message="tns:senderEmailAddress"/>
      <output message="tns:subjectsList"/>
    </operation>
  </portType>

  <binding name="simpleMobileEmailServiceBinding"
    type="tns:simpleMobileEmailService">
   <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
     style="document"/>
   <operation name="getSubjects">
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
      <soap:operation
        soapAction="http://www.everydaywebservices.com/emailservice/getsubjects"/>
    </operation>
  </binding>
  <service name="SimpleMobileEmailService">
    <port name="simpleMobileEmailServicePort"
        binding="tns:simpleMobileEmailServiceBinding">
      <soap:address location="http://www.everydaywebservices.com/emailservice"/>
    </port>
   </service>
</definitions>
```

The simple email service contains just one operation named `getSubjects`, which takes an email address as an input parameter and fetches the subject lines of emails received from the email address.

Users of your Everyday services will use this simple email service when they are expecting an incoming email from a particular sender.

SOAP messages for the simple email service

© Copyright IBM Corporation 2006. All rights reserved.

Listing 9 shows the SOAP message corresponding to the `getSubjects` SOAP request.

Listing 9: The getSubjects SOAP request.

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:tns="http://www.everydaywebservices.com/emailservice">
<soap:Body>
<tns:getSubjects>
   <tns:senderEmailAddress>
     bob@mycompany.com
   </tns:senderEmailAddress>
</tns:getSubjects>
</soap:Body>
</soap:Envelope>
```

Listing 10 shows the SOAP response to the `getSubjects` SOAP request. Note that the response consists of a `subjectsList` element that contains a number of `subjectLine` child elements. Each `subjectLine` child contains the subject line of an email message.

Listing 10: The SOAP response to the getSubjects request

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:tns="http://www.everydaywebservices.com/emailservice">
<soap:Body>
<tns:subjectsList>
   <tns:subjectLine>BirthdayParty_Invitation</tns:subjectLine>
   <tns:subjectLine>GetTogather_update_dated20060404</tns:subjectLine>
</tns:getSubjects>
</soap:Body>
</soap:Envelope>
```

Stub classes for the simple email service

Note that Listing 8 contains all the five WSDL components -- data types, input and output message, port types, bindings and endpoint details -- that you saw in the WSDL definition for the Everyday hotel service section.

Let's explore how these components of the WSDL file are reflected in stub classes generated by the stub generator tool. At this stage, you can use the stub generator tool to generate the stub classes for the WSDL file of Listing 8.

You will see that the stub generator will generate the following three files.

© Copyright IBM Corporation 2006. All rights reserved.

1. SimpleMobileEmailService_PortType
2. SimpleMobileEmailService_PortType_Stub
3. SubjectsList

The files for the simple email service are in the in the source code download of this tutorial.

The SubjectsList file is not really required at this stage. You will learn more about this file in the Matching WSDL port type with stub interface class section. For the moment we are discussing the first two stub files.

Reflecting port type in stub files

We start our exploratioan of stub classes by looking at `SimpleMobileEmailService_PortType.java` file, which contains an interface named `SimpleMobileEmailService_PortType`. Listing 11 shows the `SimpleMobileEmailService_PortType` interface.

Listing 11: The SimpleMobileEmailService_PortType interface

```
public interface SimpleMobileEmailService_PortType extends java.rmi.Remote {
        public java.lang.String[] getSubjects(java.lang.String senderEmailAddress)
       throws java.rmi.RemoteException;
}
```

The `SimpleMobileEmailService_PortType` interface extends the `java.rmi.Remote` interface, which belongs to Remote Method Invocation (RMI) framework in J2ME.

The RMI framework is designed to facilitate method invocation of remote objects.

Whenever your application requires using RMI to invoke methods of a remote object, you need a local object that exposes the same methods that the remote object exposes.

Your application will invoke methods of the local object and the local object will handle communication with the remote object. You can say the local object acts as a local proxy or stub of the actual remote object.

The `java.rmi.Remote` interface does not contain any methods. It just shows that a class implementing this interface is actually a local proxy of a remote object.

WSA uses the RMI framework to invoke remote web services. As you can guess,

© Copyright IBM Corporation 2006. All rights reserved.

WSA stub classes should implement logic to enable SOAP communication with the remote web service.

The `SimpleMobileEmailService_PortType` interface contains just one method named `getSubjects()`. That's because the WSDL file of Listing 8, used to generate `SimpleMobileEmailService_PortType` interface, contained only one operation.

Note that the name of the `SimpleMobileEmailService_PortType` interface matches with the value of the `name` attribute of the `portType` element in the WSDL file of Listing 8, except that the first letter is capitalized and "_PortType" is appended to `name` attribute value.

Matching WSDL port type with stub interface class

You can see that three things in the WSDL file of Listing 8 match with the `SimpleMobileEmailService_PortType` interface of Listing 11:

1. **Name of the simple email service operation:**
   In Listing 8, there is just one `operation` element and its `name` attribute value is `getSubjects`. This value matches with the name of the `getSubjects()` method in the `SimpleMobileEmailService_PortType`interface in Listing 11.
   This means that the name of the WSDL operation is mapped to a method name in the stub interface class.

2. **Parameter passed to the Web service operation:**
   The `operation` element in the WSDL file of Listing 6 has an input child element that defines what input parameters go with the `getSubjects` operation call. The `input` element has a message attribute whose value (`tns:senderEmailAddress`) matches with `name` attribute value of the first message element in the Listing 8.
   This means the name of the input parameter to the `getSubjects` operation call is `senderEmailAddress`.
   Now look at the `senderEmailAddress` parameter that goes along with the `getSubjects()` method call in the `SimpleMobileEmailService_PortType` interface of Listing 11. The parameter name matches with the input parameter defined in the WSDL file.
   Also note the data type of the `senderEmailAddress` parameter in the WSDL file of Listing 8 is `xsd:string`. Whereas the data type of `senderEmailAddress` parameter in Listing 11 is `String`. This means the stub generator automatically maps the XML `xsd:string` data type to a `String` object in Java.

© Copyright IBM Corporation 2006. All rights reserved.

3. **Data type returned by the Web service operation:**
Now look at the output child of the `getSubjects` operation element in Listing 8. It's `message` attribute has a value `subjectsList`, which is an `xsd:complexType`. WSA maps the `subjectLine` complex type to a stub class named `SubjectsList` (third file listed in the Stub classes for the simple email service section). The `SubjectsList` class wraps the sequence of subject lines in an array of `String` object.
Normally the stub generator maps complex types of input and output WSDL parameters to custom objects. In this case, the output complex type is mapped to `SubjectsList` class.
However, the `SubjectsList` class is very simple. It only wraps an array of `String` objects. Later in the Using the Operation object to invoke the remote service section, you will see that the stub class does not use the `SubjectsList` class at all. Instead it will use a string array directly.

In the Extending and securing the email service section we enhance the simple email service and give examples of custom objects that are actually used by stub classes to map input and output parameters.

How the stub generator implements the port type interface

Here's how the stub generator implements the client side of the simple email service, whose port type you just saw in the previous section.

The `SimpleMobileEmailService_PortType` interface is implemented in a class named `SimpleMobileEmailService_PortType_Stub`, which has been copied into Listing 12.

Listing 12: The SimpleMobileEmailService_PortType_Stub class

```
public class SimpleMobileEmailService_PortType_Stub implements
 simple.SimpleMobileEmailService_PortType,
 javax.xml.rpc.Stub
{
    private String[] _propertyNames;
    private Object[] _propertyValues;

    public SimpleMobileEmailService_PortType_Stub() {
  _propertyNames = new String[] {ENDPOINT_ADDRESS_PROPERTY};
  _propertyValues = new Object[] {"http://www.everydaywebservices.com/emailservice"};
    }

    /****************************
    // Handling RPC properties
    /****************************
    public void _setProperty(String name, Object value) {
  int size = _propertyNames.length;
  for (int i = 0; i < size; ++i) {
      if (_propertyNames[i].equals(name)) {
    _propertyValues[i] = value;
```


© Copyright IBM Corporation 2006. All rights reserved.

```
    return;
      }
  }
  String[] newPropNames = new String[size + 1];
  System.arraycopy(_propertyNames, 0, newPropNames, 0, size);
  _propertyNames = newPropNames;
  Object[] newPropValues = new Object[size + 1];
  System.arraycopy(_propertyValues, 0, newPropValues, 0, size);
  _propertyValues = newPropValues;

  _propertyNames[size] = name;
  _propertyValues[size] = value;
    }

    public Object _getProperty(String name) {
  for (int i = 0; i < _propertyNames.length; ++i) {
      if (_propertyNames[i].equals(name)) {
    return _propertyValues[i];
      }
  }
  if (ENDPOINT_ADDRESS_PROPERTY.equals(name) ||
      USERNAME_PROPERTY.equals(name) ||
      PASSWORD_PROPERTY.equals(name)) {
      return null;
  }
  if (SESSION_MAINTAIN_PROPERTY.equals(name)) {
      return new java.lang.Boolean(false);
  }
  throw new JAXRPCException("Stub does not recognize property: "+name);
    }

    protected void _prepOperation(Operation op) {
  for (int i = 0; i < _propertyNames.length; ++i) {
      op.setProperty(_propertyNames[i], _propertyValues[i].toString());
  }
    }

    /********************************************
    // Web service client-side implementation
    /********************************************
    public java.lang.String[] getSubjects(java.lang.String senderEmailAddress)
  throws java.rmi.RemoteException {
  Operation op = Operation.newInstance(_qname_getSubjects,
         _type_senderEmailAddress,
         _type_subjectsList);
  _prepOperation(op);
  op.setProperty(Operation.SOAPACTION_URI_PROPERTY,
     "http://www.everdydaywebservices.com/extendedemailservice/getSubjects");
  Object resultObj;
  try {
      resultObj = op.invoke(senderEmailAddress);
  } catch (JAXRPCException e) {
      Throwable cause = e.getLinkedCause();
      if (cause instanceof java.rmi.RemoteException) {
    throw (java.rmi.RemoteException) cause;
      }
      throw e;
  }
  java.lang.String[] result;
  Object subjectLineObj = ((Object[])resultObj)[0];
  result = (java.lang.String[]) subjectLineObj;
  return result;
    }

    /*************************
    // Data members
    /*************************
    protected static final QName _qname_getSubjects =
  new QName("http://www.everdaywebservices.com/emailservice/", "getSubjects");
```


© Copyright IBM Corporation 2006. All rights reserved.

```
    protected static final QName _qname_senderEmailAddress =
  new QName("http://www.everdaywebservices.com/emailservice/", "senderEmailAddress");
    protected static final QName _qname_subjectLine =
  new QName("http://www.everdaywebservices.com/emailservice/", "subjectLine");
    protected static final QName _qname_subjectsList =
  new QName("http://www.everdaywebservices.com/emailservice/", "subjectsList");
    protected static final Element _type_senderEmailAddress;
    protected static final Element _type_subjectsList;
    static {
  _type_senderEmailAddress = new Element(_qname_senderEmailAddress, Type.STRING);
  Element _type_subjectLine;
  _type_subjectLine = new Element(_qname_subjectLine, Type.STRING, 0, -1, false);
  ComplexType _complexType_subjectsList;
  _complexType_subjectsList = new ComplexType();
  _complexType_subjectsList.elements = new Element[1];
  _complexType_subjectsList.elements[0] = _type_subjectLine;
  _type_subjectsList = new Element(_qname_subjectsList, _complexType_subjectsList);
    }
}
```

Note that the stub generator tool just appends "_Stub" after the name of the port type interface to form the name of the client side implementing class.

The `SimpleMobileEmailService_PortType_Stub` class contains three important code segments. You need to understand the role of each segment. Listing 12 contains annotations to identify the three segments, as listed below:

1. `SimpleMobileEmailService_PortType_Stub` contains various protected, static fields. We have marked the fields as "Data members" in Listing 12. We discuss the fields in the Data members of the stub class section.

2. `SimpleMobileEmailService_PortType_Stub` also implements another interface named `javax.xml.rpc.Stub`, which contains two methods `_setProperty()` and `_getProperty()`. We have marked the `_setProperty()` and `_getProperty()` method implementations in Listing 12 as "Handling RPC properties". I will explain the `javax.xml.rpc.Stub` interface and its methods later in the Specifying configuration properties section.

3. `SimpleMobileEmailService_PortType_Stub` implements the `SimpleMobileEmailService_PortType` interface that you saw earlier in Listing 11. The `SimpleMobileEmailService_PortType` interface contains just one method named `getSubjects()`. We have marked `getSubjects()` implementation in Listing 12 as "Web service client-side implementation", which is explained in the Implementing getSubjects() section.

Now let's explore the three code segments of the `SimpleMobileEmailService_PortType_Stub` class.

© Copyright IBM Corporation 2006. All rights reserved.

Data members of the stub class

We have copied the data members segment from Listing 12 into Listing 13 for easy understanding.

Listing 13: Data members of the stub class

```
protected static final QName _qname_getSubjects =
    new QName("http://www.everdaywebservices.com/emailservice/", "getSubjects");
protected static final QName _qname_senderEmailAddress =
    new QName("http://www.everdaywebservices.com/emailservice/", "senderEmailAddress");
protected static final QName _qname_subjectLine =
    new QName("http://www.everdaywebservices.com/emailservice/", "subjectLine");
protected static final QName _qname_subjectsList =
    new QName("http://www.everdaywebservices.com/emailservice/", "subjectsList");
protected static final Element _type_senderEmailAddress;
protected static final Element _type_subjectsList;

static {
    _type_senderEmailAddress = new Element(_qname_senderEmailAddress, Type.STRING);
    Element _type_subjectLine;
    _type_subjectLine = new Element(_qname_subjectLine, Type.STRING, 0, -1, false);
    ComplexType _complexType_subjectsList;
    _complexType_subjectsList = new ComplexType();
    _complexType_subjectsList.elements = new Element[1];
    _complexType_subjectsList.elements[0] = _type_subjectLine;
    _type_subjectsList = new Element(_qname_subjectsList, _complexType_subjectsList);
}
```

Note from Listing 13 that the first four data members of the `SimpleMobileEmailService_PortType_Stub` class (`_qname_getSubjects`, `_qname_ senderEmailAddress`, `_qname_subjectLine`, and `_qname_subjectsList`) are instances of a `QName` class.

The `QName` class belongs to `javax.xml.namespace` package of WSA and represents an XML qualified name.

The first `QName` object in Listing 13 is named `_qname_getSubjects`, which represents qualified name of `getSubjects` operation. Note from Listing 13 how the `SimpleMobileEmailService_PortType_Stub` class instantiates the `_qname_getSubjects` object using a two argument `QName` constructor.

The first argument to the `QName` constructor specifies the namespace URI for the `QName` object. You can match the namespace URI passed to the `QName`constructor in Listing 13 ("http://www.everdaywebservices.com/emailservice") with the target namespace in the WSDL file of Listing 8. The two namespaces are the same. The stub generator copies the namespace URI from the WSDL file into the stub implementation.

The second argument specifies the local name of the element (`getSubjects`) that the `QName` object represents.

© Copyright IBM Corporation 2006. All rights reserved.

You might be wondering why the QName constructor takes only two arguments, a namespace URI and a local name, while qualified names in XML consist of three parts, a namespace URI, a local name, and a namespace prefix.

Actually QName has three constructors. The simplest is a one-argument constructor that only takes a local name as a string. The second is a two-argument constructor that you just used to create the _qname_getSubjects object. The third is a three-argument constructor that takes three strings representing all the three parts of an XML qualified name.

Note that the stub generator reads the three parts of the qualified name from the WSDL file. We didn't specify the prefix value for the qualified name while defining data types section in the WSDL file for the simple email service (Listing 8), therefore the stub generator used the two argument QName constructor to instantiate the _qname_getSubjects object.

Now look at the rest of the three QName objects. The _qname_senderEmailAddress object represents the qualified name of the sender's email address.

Similarly, names of the other two QName objects are also self-explanatory. The _qname_subjectLine object represents qualified name of subjectLine element of Listing 8, which wraps a subject line. The _qname_subjectsList object represents qualified name of subjectsList element of Listing 8, which wraps a number of subjectLine elements.

Instantiating elements of the simple email service

Now you will see how the SimpleMobileEmailService_PortType_Stub class uses the four QName objects to create XML elements that will carry the Web service invocation request to the remote email service as shown in the SOAP request of Listing 9.

Note that after instantiating the four QName objects in Listing 13, the SimpleMobileEmailService_PortType_Stub class instantiates a number of Element and ComplexType objects. The Element and ComplexType classes are very important in the WSA framework and need explanation.

The Element class represents an XML element that an xsd:element defines in a Web service WSDL file. For example, the SimpleMobileEmailService_PortType_Stub class uses an Element object to represent the senderEmailAddress element defined in the data types section of the WSDL file shown in Listing 8.

The Element constructor takes five parameters to specify the data required by an

© Copyright IBM Corporation 2006. All rights reserved.

xsd:element declaration. The five parameters are:

- The QName object that represents the qualified name of an element.
- A Type object that represents the type of content to be stored in the Element object. WSA defines different values as static fields of the Type class, where each value represents a type of data. For example, if you are using an element to contain a string, you will use the Type.STRING static field of the Type class.
  The Type class contains eight different static fields, each for primitive data types defined by XML 1.0 specification. The eight primitive types are boolean, byte, double, float, int, long, short and string.
  An interesting point to note is that each static filed of Type class, the Type.STRING field, is itself a Type object. Therefore, if you have a class that extends the Type class, you can pass an instance of that class -- instead of a Type static field -- to the Element constructor. This is useful when your element represents complex type content. You will see an example of doing this in the Using the ComplexType class section.
- The third and fourth parameters are of integer type and specify the minimum and maximum occurrences of the element, respectively.
- The last parameter is of boolean type and specifies whether the element is nillable. If nillable parameter is set to true, this means the element will be valid even if it has no content. If an element doesn't contain any content and it is not nillable, an exception will be raised while processing the element on the client side.

The Element class represents an xsd:element declaration in a WSDL fie. An xsd:element declaration defines an XML element which may contain attributes. The current version of the WSA implementation does not contain any method to add or process XML attributes. That's why you can't use WSA to author attributes. You may recall this from the An important limitation of WSA section where we discussed WSA's inability to author XML attributes.

While a ComplexType object represents an xsd:complexType declaration in the WSDL file. The WSDL file of Listing 8 contains only one complex type definition named subjectsList, which serves to represent the list of subject lines of the emails from a particular sender. Now we'll explain how the SimpleMobileEmailService_PortType_Stub class uses the ComplexType class to represent complex type definition in the WSDL file.

Using the ComplexType class

Just as there is only one complex type definition in the WSDL file for the simple email service, there is only one ComplexType object (named

© Copyright IBM Corporation 2006. All rights reserved.

_complexType_subjectsList) among the data members of the stub class shown in Listing 13.

A ComplexType object contains a number of Element objects in an array. The array resides inside the ComplexType object as its publicly available field named elements.

The ComplexType constructor takes no parameters. After instantiating a ComplextType object, you will instantiate the elements field of the ComplexType object and set your elements that you wish to wrap inside the complex type definition one by one. You can see how the SimpleMobileEmailService_PortType_Stub class accomplishes this in Listing 13.

The last line of code in Listing 13 is quite interesting:

```
_type_subjectsList = new Element(_qname_subjectsList,
_complexType_subjectsList);
```

Here you are wrapping the subjectsList complex type inside an Element object. You need to do this because later in the Implementing getSubjects() section you will need the array of subject lines (which at the moment is a ComplexType object with no content) in the form of an Element object.

In order to form an Element object from the _complexType_subjectsList object, you will use a two-parameter Element constructor. This two-parameter Element constructor takes the first two of the five parameters that you saw earlier in the Instantiating elements of the simple email service section.

The first parameter is the _qname_subjectsList object, which represents the subjectsList complex type.

The second parameter is a Type object that specifies what type of element content your Element object will contain. This time the element content is supposed to be a complex type. So, you will pass on the _complexType_subjectsList object as second parameter to the Element constructor.

Note that as the ComplexType class extends from the Type class, so you can pass a ComplexType object instead of a Type object.

You have seen all the data members of the SimpleMobileEmailService_PortType_Stub class. The data members specify the elements and complex types to be used while invoking the getSubjects operation of the simple email service. The next section explains how the SimpleMobileEmailService_PortType_Stub class uses these data members to act as a Web service client.

© Copyright IBM Corporation 2006. All rights reserved.

Section 5. Configuring and using WSA

Specifying configuration properties

Refer to point 2 in the How the stub generator implements the port type interface section, where we mentioned that the `SimpleMobileEmailService_PortType_Stub` class implements an interface named `javax.xml.rpc.Stub`.

Now we explain the purpose of the `javax.xml.rpc.Stub` interface, which includes just two methods named `_setProperty()` and `_getProperty()`.

The `javax.xml.rpc.Stub` interface is part of WSA and all stub implementing classes are supposed to implement this interface.

The `javax.xml.rpc.Stub` interface exposes a mechanism to maintain a set of configuration properties. These configuration properties specify the information that a stub class needs to know in order to invoke the remote service.

For example, the `javax.xml.rpc.Stub` interface defines a property named `ENDPOINT_ADDRESS_PROPERTY`, which specifies the network address of the endpoint where the remote service is listening.

Similarly, the `javax.xml.rpc.Stub` interface also has a property named `SESSION_MAINTAIN_PROPERTY`, which specifies whether the client is requesting the server to maintain a session for this request.

Another two properties named `USERNAME_PROPERTY` and `PASSWORD_PROPERTY` allow you to specify authentication data.

Next we explain how the `SimpleMobileEmailService_PortType_Stub` class implements the `_setProperty()` and `_getProperty()` methods of the `javax.xml.rpc.Stub` interface to maintain configuration properties.

Implementing _setProperty()

The `_setProperty()` method sets the value of a configuration property. A J2ME MIDlet will call `_setProperty()` method to set the configuration properties into a stub class.

© Copyright IBM Corporation 2006. All rights reserved.

A property is in the form of a name-value pair. The name identifies the property and the value is the data contained in the property. Therefore, the `_setProperty()` method takes two parameters, name and value.

We have copied the `_setProperty()` method implementation from Listing 12 into Listing 14 and annotated the code with step numbers and comments for easy understanding.

Listing 14: _setProperty() implementation

```
public void _setProperty(String name, Object value) {
    int size = _propertyNames.length;
    for (int i = 0; i < size; ++i) {
        //Step1: Check if the property already exists.
        //       If it does, replace its value with the new value.
        if (_propertyNames[i].equals(name)) {
            _propertyValues[i] = value;
            return;
        }
    }
    //Step2: If the property doesn't exist, expand the arrays.
    String[] newPropNames = new String[size + 1];
    System.arraycopy(_propertyNames, 0, newPropNames, 0, size);
    _propertyNames = newPropNames;
    Object[] newPropValues = new Object[size + 1];
    System.arraycopy(_propertyValues, 0, newPropValues, 0, size);
    _propertyValues = newPropValues;

    //Step3: Write the new property at the last position in the arrays.
    _propertyNames[size] = name;
    _propertyValues[size] = value;
}
```

From the steps and comments marked in Listing 14, you can see that the `_setProperty()` method keeps the name-value pairs of configuration properties stored in arrays named `_propertyNames` and `_propertyValues`.

The `_setProperty()` method first checks whether the property that you wish to set already exists in the arrays. If it does, the `_setProperty()` method simply overwrites the old value with the new value.

If the property doesn't exist, the `_setProperty()` method expands the arrays and copies the new property (name-value pair) at the last position of the expanded arrays.

Implementing _getProperty()

Now look at the `_getProperty()` method in Listing 15, which returns the value of a configuration property.

Listing 15: _getProperty() implementation

© Copyright IBM Corporation 2006. All rights reserved.

```
    public Object _getProperty(String name) {
        for (int i = 0; i < _propertyNames.length; ++i) {
            //Step1: Check if the property exists.
            //       If exists, return its value.
            if (_propertyNames[i].equals(name)) {
                return _propertyValues[i];
            }
        }

        //Step2: Return default values.
        if (ENDPOINT_ADDRESS_PROPERTY.equals(name) ||
            USERNAME_PROPERTY.equals(name) ||
            PASSWORD_PROPERTY.equals(name)) {
            return null;
        }
        if (SESSION_MAINTAIN_PROPERTY.equals(name)) {
            return new java.lang.Boolean(false);
        }
        throw new JAXRPCException("Stub does not recognize property: "+name);
    }
```

You can see that the `getProperty()` method is intelligent. If the property exists, it returns the value. If the property doesn't exist, it returns a default value depending on which configuration property an application wants to read.

For example, if you are looking for `ENDPOINT_ADDRESS_PROPERTY`, the default value is a null object, meaning the network address of the service endpoint is not known.

On the other hand, if you asked for `SESSION_MAINTAIN_PROPERTY`, the default value is a `java.lang.Boolean` object with `false` as its value, meaning the client doesn't want the endpoint to maintain a session for this request.

SimpleMobileEmailService_PortType_Stub constructor

Look at `SimpleMobileEmailService_PortType_Stub` constructor shown in Listing 16.

Listing 16: SimpleMobileEmailService_PortType_Stub constructor

```
public SimpleMobileEmailService_PortType_Stub() {
    _propertyNames = new String[] {ENDPOINT_ADDRESS_PROPERTY};
    _propertyValues = new Object[] {"http://www.everdaywebservices.com/emailservice"};
}
```

The `SimpleMobileEmailService_PortType_Stub` constructor sets the value of a property named `ENDPOINT_ADDRESS_PROPERTY`, which specifies the network address where the service is hosted.

Note that the stub generator tool reads address information from the `location`

© Copyright IBM Corporation 2006. All rights reserved.

attribute of the `soap:address` element of WSDL file and sets into `ENDPOINT_ADDRESS_PROPERTY`.

Setting the `ENDPOINT_ADDRESS_PROPERTY` property in the stub constructor is only for the convenience of a J2ME programmer. The remote endpoint's network address is often specified in a WSDL file and therefore setting the value of this property in a stub constructor can save a line of MIDlet code.

However a J2ME programmer can still overwrite the property by calling the `_setProperty()` method in the MIDlet.

Implementing getSubjects()

Now we explain the `getSubjects()` method implementation mentioned in point 3 of the How the stub generator implements the port type interface section.

Look at the `getSubjects()` method implementation in Listing 17.

Listing 17: getSubjects() implementation

```
public java.lang.String[] getSubjects(java.lang.String senderEmailAddress)
      throws java.rmi.RemoteException {

    /****Step1****/
    Operation op = Operation.newInstance(_qname_getSubjects,
                         _type_senderEmailAddress, _type_subjectsList);
    /****Step2****/
    _prepOperation(op);
    op.setProperty(Operation.SOAPACTION_URI_PROPERTY,
                        "http://www.everydaywebservices.com/emailservice");
    Object resultObj;
    try {
        /**** Step 3****/
        resultObj = op.invoke(senderEmailAddress);
    } catch (JAXRPCException e) {
        Throwable cause = e.getLinkedCause();
        if (cause instanceof java.rmi.RemoteException) {
            throw (java.rmi.RemoteException) cause;
        }
        throw e;
    }
    /**** Step 4****/
    java.lang.String[] result;
    Object subjectLineObj = ((Object[])resultObj)[0];
    result = (java.lang.String[]) subjectLineObj;
    return result;
}
```

The `getSubjects()` method takes the sender's email address as its input and returns the list of subject lines of all the email messages from the particular sender.

In order to fetch the subject lines, the `getSubjects()` method will use a class named Operation, which is part of `javax.microedition.xml.rpc` package of

© Copyright IBM Corporation 2006. All rights reserved.

WSA. The `Operation` class contains methods to invoke a remote operation.

Look at step 1 in Listing 17, in which the `getSubjects()` method instantiates a new `Operation` object by calling its static method named `newInstance()`. The `newInstance()` method takes three parameters:

1. The first parameter to the `Operation` constructor is a `QName` object named `_qname_getSubjects`. The `_qname_getSubjects` object specifies the WSDL operation that you wish to invoke using the `Operation` object.
   Recall from the Data members of the stub class section that the `SimpleMobileEmailService_PortType_Stub` class instantiated a `QName` object for the `getSubjects` element of the WSDL file shown in Listing 8. In step 1 of Listing 17, it passes the same `QName` object to the `Operation` constructor. That's because you are about to use the `Operation` object to invoke the `getSubjects` operation.

2. The first parameter to the `Operation` constructor has specified the operation that you wish to invoke. You also need to specify the input parameter that goes with the operation invocation call. The second parameter is an `Element` object named `_type_senderEmailAddress` that represents the input parameter. Recall that you created this `_type_senderEmailAddress` object earlier in Listing 13.
   Note another important point. As the `_type_senderEmailAddress` object represents `senderEmailAddress` element of the WSDL file (Listing 8), so the `_type_senderEmailAddress` object only defines the XML structure of the input parameter. It doesn't define the actual content that goes in the input parameter. You will provide the content while invoking the operation later in the Using the Operation object to invoke the remote service section.

3. The third parameter is also an `Element` object `named` `_type_subjectsList`. The third parameter, `subjectsList`, represents the output or return element of the operation call.
   Recall from Data members of the stub class section that the `_type_subjectsList` object wraps the `subjectsList` complex type structure defined in WSDL of Listing 8.
   Why does the `Operation` object need to know the `subjectsList` complex type structure? Because the subjectsList complex type structure represents the output or return format of `getSubjects()` method. Later when you use the `Operation` object to invoke a WSDL operation, the `Operation` object will need to process the response from the remote Web service to extract the output return data. At that time, the `Operation` object will use the subjectsList complex type structure to know the data format of the response message and fill it with content

© Copyright IBM Corporation 2006. All rights reserved.

returned by the remote Web service endpoint.

Preparing the Operation object

After instantiating the `Operation` object, you will prepare it for invocation of the remote service. Look at Listing 17 to find that in step 2, the `getSubjects()` method calls a method named `_prepOperation()`, passing the `Operation` object along with the method call.

`_prepOperation()` is a protected helper method in the `SimpleMobileEmailService_PortType_Stub` class, which sets all the configuration properties stored in the `SimpleMobileEmailService_PortType_Stub` class inside the `Operation` object.

In order to set the configuration properties in the `Operation` object, `_prepOperation()` calls `setProperty()` method of the `Operation` class a number of times, once for each configuration property.

The `setProperty()` method takes a name-value pair representing a configuration property and sets the property in the `Operation` object.

Next, the `getSubjects()` method sets another property named `SOAPACTION_URI_PROPERTY` in the `Operation` object. The `SOAPACTION_URI_PROPERTY` is not defined in the `javax.xml.rpc.Stub` interface, in which other properties (e.g. `ENDPOINT_ADDRESS_PROPERTY`) are defined.

The `SOAPACTION_URI_PROPERTY` is defined in the `Operation` class.

You might be wondering why this property is segregated from the other four properties that were defined in the `javax.xml.rpc.Stub` interface.

`SOAPACTION_URI_PROPERTY` is a SOAP-specific property, which specifies URL of the target operation.

The other four properties defined in the `javax.xml.rpc.Stub` interface are not SOAP-specific. For example, the `ENDPOINT_ADDRESS_PROPERTY` simply defines the network address of the endpoint implementation.

The `Operation` class is supposed to author and process SOAP messages, so it wraps SOAP-specific properties.

Note that the stub generator copies the values of the `ENDPOINT_ADDRESS_PROPERTY` from the `soapAction` attribute of `soap:operation` element in the WSDL file of Listing 8 into the

© Copyright IBM Corporation 2006. All rights reserved.

SimpleMobileEmailService_PortType_Stub class. This means whatever value of the soapAction attribute you set in the WSDL file will be set into the corresponding stub class.

Using the Operation object to invoke the remote service

Your Operation object is now all set to be used to invoke the remote service. So in step 3, you can call an invoke() method of the Operation object, passing the sender's email address as input data. The invoke() method will internally do the following:

1. Author the SOAP request for the getSubjects() method call
2. Send the request to the remote endpoint using the networking support in J2ME
3. Fetch the response SOAP message
4. Parse the response to extract the list of subject lines
5. Load the list of subject lines into an array of String objects, where each String object represents a subject line
6. Return the string array to the calling application as an Object instance

You can see that the Operation object performs most of the hard work involved in invoking the remote Web service.

When the invoke() method returns an Object instance, the getSubjects() method will cast the instance into a string array and return the string array to the calling application, as in step 4 in Listing 17.

Now you have explored all three code segment mentioned in the How the stub generator implements the port type interface section. This completes our discussion on the SimpleMobileEmailService_PortType_Stub class.

Using the simple email service

Next you'll see how the J2ME application will use stub classes of simple email service to invoke getSubjects operation.

You need to perform three steps to invoke the getSubjects operation, as shown in Listing 18:

© Copyright IBM Corporation 2006. All rights reserved.

1. Instantiate the simple email service stub class
2. Set configuration properties for the Web service
3. Call `getSubjects()` method of the stub class, passing sender's email address as parameter along with the method call. The `getSubjects()` method returns a string array containing subject lines of emails received from the particular email address

Listing 18: Steps to invoking the getSubjects operation

```
/**** Step1 ****/
SimpleMobileServiePortType_Stub service = new SimpleMobileServiePortType _Stub();
/**** Step2 ****/
service._setProperty(SignatureServicePortType_Stub.SESSION_MAINTAIN_PROPERTY,
                     new Boolean(true));
/**** Step3 ****/
String[] subjectsList = service.getSubjects("bob@mycompany.com");
```

The source code download for this tutorial contains a MIDlet named `SimpleEmailServiceMIDlet`, which demonstrates all the concepts learned in this and the previous sections.

In order to try `SimpleEmailServiceMIDlet`, you will also need to have simple email service hosted on a SOAP server. A simple classed named RequestReceiver is included in the source code download that can act as an emulator of the simple email service. You can run the emulator by using the following command line statement:

```
%JAVA_HOME%\java -classpath . RequestReceiver
```

The emulator listens for and saves incoming requests from J2ME clients, so that you can see what your SimpleEmailServiceMIDlet has authored.

Section 6. Extending and securing the email service

WSDL of the extended email service

Having explored how stub classes work for the simple email service, it's time to extend the simple service to see how multiple operations and complex types in a WSDL file are reflected in stub classes.

© Copyright IBM Corporation 2006. All rights reserved.

We will call the extended version of the simple email service as "extended mobile email service" or just "extended email service".

Look at Listing 19, which shows the WSDL file of the extended email service named ExtendedMobileEmailService.

Listing 19: WSDL representation of extended email service

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://www.everydaywebservices.com/extendedemailservice"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    name="emailService"
    targetNamespace="http://www.everydaywebservices.com/extendedemailservice">

   <types>
    <schema xmlns="http://www.w3.org/2001/XMLSchema"
         targetNamespace="http://www.everydaywebservices.com/extendedemailservice">

      <xsd:element name="subjectsList">
         <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="subjectLine" type="xsd:string" minOccurs="0"
                 maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:complexType>
      </xsd:element>

      <xsd:element name="senderEmailAddress" type="xsd:string"/>

      <xsd:element name="messageIdentifier">
         <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="subject" type="xsd:string" nillable="true"/>
              <xsd:element name="from" type="xsd:string" nillable="true"/>
            </xsd:sequence>
         </xsd:complexType>
      </xsd:element>

      <xsd:element name="message">
         <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="subject" type="xsd:string" nillable="true"/>
              <xsd:element name="from" type="xsd:string" nillable="true"/>
              <xsd:element name="messageText" type="xsd:string" nillable="true"/>
            </xsd:sequence>
         </xsd:complexType>
      </xsd:element>
    </schema>
   </types>

  <message name="senderEmailAddress">
    <part name="senderEmailAddress" element="tns:senderEmailAddress"/>
  </message>
  <message name="subjectsList">
    <part name="subjectsList" element="tns:subjectsList"/>
  </message>

  <message name="messageIdentifier">
    <part name="messageIdentifier" element="tns:messageIdentifier"/>
  </message>
```


© Copyright IBM Corporation 2006. All rights reserved.

```
  <message name="message">
    <part name="message" element="tns:message"/>
  </message>

  <portType name="extendedMobileEmailService">
    <operation name="getSubjects">
      <documentation>
        Get subject lines of emails from a particular sender.
      </documentation>
      <input message="tns:senderEmailAddress"/>
      <output message="tns:subjectsList"/>
    </operation>
    <operation name="getMessage">
      <documentation>Get an email message from your inbox.</documentation>
      <input message="tns:messageIdentifier"/>
      <output message="tns:message"/>
    </operation>

  </portType>

  <binding name="extendedMobileEmailServiceBinding" type="tns:extendedMobileEmailService">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>

    <operation name="getSubjects">
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/></output>
      <soap:operation
      soapAction="http://www.everdydaywebservices.com/extendedemailservice/getSubjects"/>
    </operation>
    <operation name="getMessage">
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
      <soap:operation
       soapAction="http://www.everdydaywebservices.com/extendedemailservice/getMessage"/>
    </operation>
  </binding>

  <service name="ExtendedMobileEmailServiceBinding">
    <port name="ExtendedMobileEmailServicePort"
      binding="tns:extendedMobileEmailServiceBinding">
      <soap:address
         location=" http://www.everdydaywebservices.com/extendedemailservice"/>
    </port>
   </service>
</definitions>
```

If you compare Listing 19 with the earlier WSDL file in Listing 8 for the simple email service you will find the following differences:

1. Simple email service only contained one operation named `getSubjects`. The extended service contains another operation named `getMessage` in addition to the `getSubjects` operation. The `getMessage` operation takes a subject line and a sender's email address. It returns incoming email message with matching subject line from the particular sender, if any.

© Copyright IBM Corporation 2006. All rights reserved.

2. The input parameter is a complex type named `messageIdentifier` with two string type elements, namely `subject`, the subject line of the message you are looking for, and `from`, the sender's email address.
3. The email message returned by the `getMessage` operation is a complex type named `message` containing three of text strings, namely `subject`, subject line of the message, `from`, the sender's email address, and `messageText`, a textual message.

Stub classes for extended email service

The stub generator will generate the following files for the WSDL of Listing 19:

1. ExtendedMobileEmailService_PortType
2. ExtendedMobileEmailService_PortType_Stub
3. SubjectsList
4. MessageIdentifier
5. Message

The stub files of the extended email service can be found in the source code download of this tutorial.

Now we will explain the differences between these stub files and the stub files for the simple email service you saw earlier in the Stub classes for the simple email service section.

You can see that the `ExtendedMobileEmailService_PortType_Stub.java` file contains an interface name `ExtendedMobileEmailService_PortType`, which defines two method named `getSubjects()` and `getMessage()`. These two methods represent the two operations of the extended email service (WSDL file of Listing 19).

As you can easily guess, the `ExtendedMobileEmailService_PortType` interface for the extended service corresponds to the `SimpleMobileEmailService_PortType` interface for the simple email service that you saw earlier in the Reflecting port type in stub files section.

We will explain how stub classes implement the `ExtendedMobileEmailService_PortType` interface, but first we'll show you how the stub classes have handled data of the WSDL file.

© Copyright IBM Corporation 2006. All rights reserved.

Data types of the extended email service

The extended email service stub files contain `SubjectsList`, `MessageIdentifier` and `Message` classes, which represent the `subjectsList`, `messageIdentifier` and `message` complex types of the extended service WSDL file of Listing 19 respectively.

To demonstrate how complex types are reflected in stub classes, we have copied `Message` class into Listing 20. The `Message` class represents the message complex type of the extended email service WSDL file.

Listing 20: Message class of extended email service

```
public class Message {
    protected java.lang.String subject;
    protected java.lang.String from;
    protected java.lang.String messageText;

    public Message() {
    }

    public Message(java.lang.String subject,
                   java.lang.String from,
                   java.lang.String messageText) {
        this.subject = subject;
        this.from = from;
        this.messageText = messageText;
    }

    public java.lang.String getSubject() {
        return subject;
    }

    public void setSubject(java.lang.String subject) {
        this.subject = subject;
    }

    public java.lang.String getFrom() {
        return from;
    }

    public void setFrom(java.lang.String from) {
        this.from = from;
    }

    public java.lang.String getMessageText() {
        return messageText;
    }

    public void setMessageText(java.lang.String messageText) {
        this.messageText = messageText;
    }
}
```

Note that the `Message` class contains three sets of setter and getter methods for the three fields of the complex type. For example, the `getSubject()` method does not take any parameter and returns a string representing the subject line of the email

© Copyright IBM Corporation 2006. All rights reserved.

message. Similarly, the `setSubject()` method sets the subject line into a `String` object.

These data type classes are used in implementing the `ExtendedMobileEmailService_PortType` interface. The `ExtendedMobileEmailService_PortType_Stub` class implements the `ExtendedMobileEmailService_PortType` interface of extended email service. Now you will see how the `ExtendedMobileEmailService_PortType_Stub` class uses the `Message` data type class while implementing client-side of the extended email service.

How getMessage() is implemented

The `ExtendedMobileEmailService_PortType_Stub.java` class contains the implementation of the `getSubjects()` and `getMessage()` methods for the extended email service.

The `getMessage()` implementation is similar to the `getSubjects()` method implementation that you saw in the Implementing getSubjects() section, with only two differences.

In order to explain the two differences, we have copied `getMessage()` implementation from the `ExtendedMobileEmailService_PortType_Stub.java` file into Listing 21.

Listing 21: getMessage() implementation

```
public enhanced.Message getMessage(java.lang.String subject, java.lang.String from)
        throws java.rmi.RemoteException {
    /**** Putting input parameters into an Object array ****/
    Object[] inputObject = new Object[2];
    inputObject[0] = subject;
    inputObject[1] = from;

    /**** Step 1 ****/
    Operation op = Operation.newInstance(_qname_getMessage,
                _type_messageIdentifier,
                _type_message);
    /**** Step 2 ****/
    _prepOperation(op);
    op.setProperty(Operation.SOAPACTION_URI_PROPERTY,
         "http://www.everdydaywebservices.com/extendedemailservice/getMessage");
    Object resultObj;
    try {
        /**** Step 3 ****/
        resultObj = op.invoke(inputObject);
    } catch (JAXRPCException e) {
        Throwable cause = e.getLinkedCause();
        if (cause instanceof java.rmi.RemoteException) {
            throw (java.rmi.RemoteException) cause;
        }
        throw e;
    }
```


© Copyright IBM Corporation 2006. All rights reserved.

```
    /**** Step 4 ****/
    enhanced.Message result;
    if (resultObj == null) {
        result = null;
    } else {
        result = new enhanced.Message();
        java.lang.String string;
        Object subjectObj = ((Object[])resultObj)[0];
        string = (java.lang.String)subjectObj;
        result.setSubject(string);
        java.lang.String string2;
        Object fromObj = ((Object[])resultObj)[1];
        string2 = (java.lang.String)fromObj;
        result.setFrom(string2);
        java.lang.String string3;
        Object messageTextObj = ((Object[])resultObj)[2];
        string3 = (java.lang.String)messageTextObj;
        result.setMessageText(string3);
    }
    return result;
}
```

The first difference appears in the code marked "Putting input parameters into an Object array" in Listing 21, in which the `getMessage()` method puts input parameters into an `Object` array named `inputObject`. This code segment was missing in the `getSubjects()` method implementation of Listing 17.

That's because the `getMessage()` method takes two input parameters, while the `getSubjects()` method took one parameter.

The `getSubjects()` method passed the input parameter directly to `Operation.invoke()` method (step 3 in Listing 17). On the other hand, the `getMessage()` method will pass the `inputObject` array (which wraps both the input parameters) to `Operation.invoke()` method (step 3 in Listing 21).

Note from both Listing 17 and Listing 21 that input parameters are passed to `Operation.invoke()` method either directly, as in Listing 17, or indirectly, as in Listing 21.

You can see the second difference by comparing step 4 of `getSubjects()` in Listing 17 with step 4 of getMessage() in Listing 21. Step 4 of `getMessage()` is more complicated than the previous step 4. That's because the `getMessage()` method has to extract different parts of the message from the SOAP response and set the parts into a `Message` object. This all is done in step 4 of Listing 21, which makes it a bit complicated.

The rest of `getMessage()` implementation is the same as `getSubjects()`.

Using stub classes of extended email service

Listing 22 shows the steps that a J2ME MIDlet will need to follow in order to use stub

© Copyright IBM Corporation 2006. All rights reserved.

classes of the extended email service to invoke `getMessage` operation.

Listing 22: Steps a MIDlet will follow to use stub classes of extended email service

```
/**** Step1 ****/
ExtendedMobileEmailService_PortType_Stub service =
  new ExtendedMobileEmailService_PortType_Stub();
/**** Step2 ****/
service._setProperty(SignatureServicePortType_Stub.SESSION_MAINTAIN_PROPERTY,
                     new Boolean(true));
/**** Step3 ****/
Message message = service.getMessage("Recent Project", "bob@mycompany.com");
/**** Step4 ****/
String messageText = message.getMessageText();
System.out.println("Message text:" +messageText);
```

You can compare these steps of Listing 22 with the earlier steps of the simple email service MIDlet of Listing 18. The two MIDlet's are similar but Listing 22 contains an extra step (step 4).

Note that `getMessage()` returns a `Message` object in step 3 of Listing 22. In step 4, you will process the `Message` object by calling its getter methods.

A MIDlet named `ExtendedEmailServiceMIDlet` is included in the source code download of this tutorial. The MIDlet contains all the code you have seen in this section in both source and compiled form.

The `ExtendedEmailServiceMIDlet` accompanies a Web service emulator that you can use to see the SOAP message from the MIDlet.

Securing the extended email service

You have seen how stub classes work for the simple and extended email services. You have also seen how a J2ME MIDlet uses the stub classes for the simple and extended email services.

Now it is time to start securing your extended email service. We call the secure version of the extended email service "secure mobile email service" or just "secure email service".

The first step is to write a WSDL file for the secure email service, as shown in Listing 23.

Listing 23: WSDL of the secure email service

```
<?xml version="1.0" encoding="UTF-8"?>
```


© Copyright IBM Corporation 2006. All rights reserved.

```
<definitions
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://www.everydaywebservices.com/secureemailservice"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    name="secureEmailService"
    targetNamespace="http://www.everydaywebservices.com/secureemailservice">

   <types>
    <schema xmlns="http://www.w3.org/2001/XMLSchema"
               targetNamespace="http://www.everydaywebservices.com/secureemailservice">

      <xsd:element name="subjectsList">
         <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="subjectLine" type="xsd:string" minOccurs="0"
                 maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:complexType>
      </xsd:element>

      <xsd:element name="senderEmailAddress">
         <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="senderEmailAddress" type="xsd:string"/>
              <xsd:element name="signature" type="tns:Signature" nillable="true"/>
            </xsd:sequence>
         </xsd:complexType>
      </xsd:element>

      <xsd:element name="messageIdentifier">
         <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="subject" type="xsd:string" nillable="true"/>
              <xsd:element name="from" type="xsd:string" nillable="true"/>
              <xsd:element name="signature" type="tns:Signature" nillable="true"/>
            </xsd:sequence>
         </xsd:complexType>
      </xsd:element>

      <xsd:element name="message">
         <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="subject" type="xsd:string" nillable="true"/>
              <xsd:element name="from" type="xsd:string" nillable="true"/>
              <xsd:element name="messageText" type="xsd:string" nillable="true"/>
            </xsd:sequence>
         </xsd:complexType>
      </xsd:element>

      <xsd:complexType name="Signature">
        <xsd:sequence>
           <xsd:element name="SignedInfo">
             <xsd:complexType>
             <xsd:sequence>
               <xsd:element name="CanonicalizationMethod" type="xsd:string"/>
               <xsd:element name="SignatureMethod" type="xsd:string"/>
               <xsd:element name="Reference">
                 <xsd:complexType>
                   <xsd:sequence>
                     <xsd:element name="DigestMethod" type="xsd:string"/>
                     <xsd:element name="DigestValue" type="xsd:string"/>
                     <xsd:element name="URI" type="xsd:string"/>
                   </xsd:sequence>
                 </xsd:complexType>
               </xsd:element>
             </xsd:sequence>
             </xsd:complexType>
           </xsd:element>
```


© Copyright IBM Corporation 2006. All rights reserved.

```
          <xsd:element name="SignatureValue" type="xsd:string"/>
          <xsd:element name="KeyInfo">
             <xsd:complexType>
                <xsd:sequence>
                 <xsd:element name="KeyName" type="xsd:string"/>
                </xsd:sequence>
             </xsd:complexType>
          </xsd:element>
       </xsd:sequence>
     </xsd:complexType>

   </schema>
  </types>

 <message name="senderEmailAddress">
   <part name="senderEmailAddress" element="tns:senderEmailAddress"/>
 </message>
 <message name="subjectsList">
   <part name="subjectsList" element="tns:subjectsList"/>
 </message>
 <message name="messageIdentifier">
   <part name="messageIdentifier" element="tns:messageIdentifier"/>
 </message>
 <message name="message">
   <part name="message" element="tns:message"/>
 </message>

 <portType name="secureMobileEmailService">
   <operation name="getSubjects">
     <documentation>
       Get subject lines of emails from a particular sender.
     </documentation>
     <input message="tns:senderEmailAddress"/>
     <output message="tns:subjectsList"/>
   </operation>
   <operation name="getMessage">
     <documentation> Get email message from your inbox. </documentation>
     <input message="tns:messageIdentifier"/>
     <output message="tns:message"/>
   </operation>

 </portType>

 <binding name="secureMobileEmailServiceBinding" type="tns:secureMobileEmailService">
   <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>

   <operation name="getSubjects">
     <input>
       <soap:body use="literal"/>
     </input>
     <output>
       <soap:body use="literal"/></output>
     <soap:operation
      soapAction="http://www.everdaywebservices.com/secureemailservice/getSubjects"/>
   </operation>
   <operation name="getMessage">
     <input>
       <soap:body use="literal"/>
     </input>
     <output>
       <soap:body use="literal"/>
     </output>
     <soap:operation
       soapAction="http://www.everdaywebservices.com/secureemailservice/getMessage"/>
   </operation>
 </binding>

 <service name="SecureMobileEmailService">
   <port name="secureMobileEmailServicePort"
```


© Copyright IBM Corporation 2006. All rights reserved.

```
            binding="tns:secureMobileEmailServiceBinding">
        <soap:address
            location="http://www.everdaywebservices.com/secureemailservice"/>
      </port>
    </service>
</definitions>
```

You can see that the WSDL file of Listing 23 contains definitions of a complex type named `Signature`. The `Signature` complex type is an input parameter to both `getSubjects` and `getMessage` operations defined in the secure email service.

The `Signature` complex type of Listing 23 defines the structure of a complete XML signature that carries authentication information for the user (e.g. Alice) trying to access your secure email service. Therefore, the `Signature` complex type contains all the child elements that you saw earlier in the Using XML signatures section, while discussing the secure SOAP message of Listing 7.

In the next part of this tutorial, you will use the WSDL file of Listing 23 to generate stub classes for the secure email service and enhance the stub classes to author the secure SOAP message of Listing 7.

Section 7. Summary

This tutorial has introduced the concept of securing wireless access to Web services.

You've seen sample application scenarios where you need to consume your Web services wirelessly, whether with or without security. We have discussed the architecture of integrating various technology components to enable security in WSA applications.

We have also seen a detailed analysis of stub and other WSA classes and demonstrated their functionality. We have provided an explanation of the WSDL and SOAP messages that WSA works with.

We saw the WSDL interface of a secure Web service. In Part 2 of this series, we will build security into these services.

© Copyright IBM Corporation 2006. All rights reserved.

Downloads

Description	Name	Size	Download method
Source code	ws-soa-securesoap1-source.zip	3KB	HTTP

Information about download methods

© Copyright IBM Corporation 2006. All rights reserved.

Resources

Learn

- Read official JSR-00172 J2ME Web Services Specification at the JCP Web site.
- Web Services APIs for J2ME, Part 1: Remote service invocation API (developerWorks, July 2004) introduces WSA architecture.
- Designing mobile Web services (developerWorks, January 2006) introduces best practices for designing wireless Web services.
- Read the official WSDL, SOAP, and XML Schema specifications at W3C Web site.
- Deploying Web services with WSDL (developerWorks, November 2001) discusses WSDL and SOAP in detail.
- Read the official XML Digital Signatures specification at W3C Web site.
- The tutorial Secure XML messaging with JMS (developerWorks, November 2005) discusses XML signature in detail.
- The series of articles on Securing Java Card applications (Part 1 and 2) demonstrates how to integrate Java Card technology into J2ME MIDlets using SATSA.
- Check out Security and Trust Services API for J2ME (SATSA) at the Sun Web site.
- Read the official specification for XML Information Set at W3C Web site.
- Learn about Java Card technology and Smart cards.

Get products and technologies

- Download J2ME Toolkit version Java 2, Micro Edition (J2ME) Wireless Toolkit 2.2 Release and Sun Java Wireless Toolkit 2.3 Beta Release from the Sun Web site.
- Java Card Development Kit from Sun lets you develop and test Java card applications.
- Download XML Security Suit for Java (XSS4J) from IBM alphaworks for securing XML data.
- Build your next development project with IBM trial software, available for download directly from developerWorks.

Discuss

- Participate in developerWorks blogs and get involved in the developerWorks

© Copyright IBM Corporation 2006. All rights reserved.

community.

About the author

Bilal Siddiqui

Bilal Siddiqui is an electronics engineer, an XML consultant, and the founder of XML4Java.com, a company focused on simplifying e-business. After graduating in 1995 with a degree in electronics engineering from the University of Engineering and Technology, Lahore, he began to design software solutions for industrial control systems. Later, he turned to XML and used his experience of programming in C++ to build Web- and Wap-based XML processing tools, server-side parsing solutions, and service applications. He is a technology evangelist and a frequently published technical author.

© Copyright IBM Corporation 2006. All rights reserved.