

Online banking with Apache Geronimo and Axis2, Part 1: The service: Laying down the framework

Skill Level: Intermediate

[Nicholas Chase \(ibmquestions@nicholaschase.com\)](mailto:ibmquestions@nicholaschase.com)
Freelance writer
Backstop Media, LLC

21 Mar 2006

Dive deep into the intricacies of using Apache Geronimo and Axis2 to build a complex Web services application. This three-part tutorial series walks you through building an example online banking Web service, documenting each step of the process so new users can quickly grasp the concepts and build a complete Web service and Web-interface client that connects to and communicates with the Web service. In the first installment, you get acquainted with the example Web service and the Web services that use WSDL, build and compile a WSDL file, and test and deploy it on Geronimo.

Section 1. Before you start

This three-part tutorial series is for you if you're interested in building a large Web service using Apache Geronimo and Apache Axis2. Learn how to build a WSDL file with several operations relevant to an online banking system. Then compile the WSDL file into a Web service using Axis2, and deploy and test it on Geronimo.

About this series

The example you create in this series is an online banking Web service. The Web service, created using WSDL and Axis2, will contain an Apache Derby database used to save transaction data, account data, and bill-paying data. You'll deploy Axis2 as a Web application on Apache Geronimo, and the new Web service will be an Axis archive deployed within Axis2. There are several possible commands in online

banking that you'll use in the example, including view accounts, search transactions, transfer funds, and bill paying. You'll also create a client Web application to connect to the Web service to query information and perform transactions -- basically to call the operations implemented in the Web service.

In this first installment of the three-part series, you build the online banking Web service's infrastructure in WSDL. In Part 2, you build the Web service's functionality by storing account and transaction information in an Apache Derby database. Finally, in Part 3, you create a fully functional client-Web-based user interface (UI) to the Web service where a user can log in, do online banking, and log out.

About this tutorial

This tutorial starts by introducing the example Web service and exposing the various Web services using WSDL. You use Axis2 to convert the WSDL file to Java™ code, where some simple functionality will be implemented to test communication between the client and the Web service. The test environment includes Axis2 deployed as a module on Geronimo, the Web service deployed as an archive within Axis2, and a simple Java client that communicates to the Web service.

Prerequisites

You need the following tools to follow along with this tutorial:

- Apache Geronimo -- Download [Apache Geronimo V1.0](#), on which you'll deploy an Axis2 Web service.
- Apache Axis2 -- Download the [version 0.94 Axis2 WAR and binary distributions](#), which you'll use to build a Web service from WSDL.
- Apache Ant -- Download [Apache Ant](#), which you'll need, because Axis2's WSDL2Java tool creates a handy Ant build.xml file for building the Web service, which makes building the Web service's source code a lot easier.
- Java code -- Download the latest Java 1.4 version, [Java 1.4.10](#), required by Geronimo, Axis2, and Ant.

This tutorial assumes you have basic knowledge of Java syntax and coding; however, no knowledge of Axis2 or specific Geronimo knowledge is assumed.

Section 2. Overview of the online banking Web service

The following sections give you a solid overview of the final product of this tutorial series, the example online banking Web service. This Web service covers several features of the online banking experience, including viewing accounts and transactions, transferring funds, applying for a loan, and paying bills online.

Accounts

After you log in to the Web service, you can see what accounts you have and their balances. For security, logging in involves sending a username and password and receiving a binary token for security. Thus, after you've logged in, all you need to send is the token for the Web service to recognize who you are. From there, you can click to view transactions in your account, which you'll look at next.

Transactions

Everything you do in a bank involves a transaction, and a history of each transaction -- which can view for each account -- is logged on your account. The Web service also lets you search transactions based on check number, amount, or date. This makes it easier for you to backtrack and look up transactions in your account come tax time. You can transfer funds from one of your accounts to another; this is logged as a transaction for the account.

Apply for a loan

This feature allows you to apply for a loan through the online banking Web service. You can enter the type of loan, how much you want, and how long you want to take to pay it off. After you've successfully submitted a loan application, you can view the status of each of your loans.

Pay bills

Online bill paying services have become very popular. They let you pay your bills through your bank accounts. Before you pay bills, however, you have to add the people or companies that you'll pay, called *payees*. The Web service allows you to view, add, and edit the payees on your account. With payees in your account, you can pay off debts and transfer funds to them. There are two types of supported payments: *one-time payments* and *recurring payments*. You can perform each type through the Web service. Each payment also has a date associated with it, and before a payment has been sent (a *pending payment*), you're allowed to edit its contents.

In the next section, you start building the WSDL of the online banking Web service.

Section 3. Data structures

The messages and operations you support in this Web service require several data structures for passing data back and forth, also known as *complex types* in WSDL. These data structures are transmitted via Simple Object Access Protocol (SOAP), and they are defined using WSDL. The operations for the Web service require six data structures; you start with the most common one, the transaction.

Transaction

In the context of your online banking Web service, two accounts (or an account and the bank) are the affected parties. For the purposes of your online banking application, a transaction has five possible properties. Take a look at them, as shown in [Listing 1](#). Now create a WSDL file, `OnlineBanking.wsdl`, and place the `Transaction` `complexType` inside of it.

The five elements in the `Transaction` `complexType` are shown in boldface in [Listing 1](#). All are required by writing `minOccurs="1" maxOccurs="1"`. This means that the element number can show up at least once and at most once, meaning exactly once. Notice that `checknumber` is not required, because `minOccurs="0"`, which means that that it can be omitted or appear exactly once in a `Transaction`. Also, look at the data types. Notice that `number` is defined as an `int`, `checknumber` and `int`, `date` and `date`, `header` and `string`, and `amount` and `float`.

Listing 1. The transaction

```
<xsd:complexType name="Transaction">
  <xsd:sequence>
    <xsd:element minOccurs="1" maxOccurs="1"
      name="number" type="xsd:int" />
    <xsd:element minOccurs="0" maxOccurs="1"
      name="checknumber" type="xsd:int" />
    <xsd:element minOccurs="1" maxOccurs="1"
      name="date" type="xsd:date" />
    <xsd:element minOccurs="1" maxOccurs="1"
      name="header" type="xsd:string" />
    <xsd:element minOccurs="1" maxOccurs="1"
      name="amount" type="xsd:float" />
  </xsd:sequence>
</xsd:complexType>
```

Next you look at the payee complex type.

Payee

Payees are those to whom you can pay your bills via online banking. Thus, there are several fields required to get the address and billing account information. Add the payee complex type, as shown in [Listing 2](#), to your WSDL file.

Listing 2. A payee

```
<xsd:complexType name="Payee">
  <xsd:sequence>
    <xsd:element minOccurs="1" maxOccurs="1"
      name="name" type="xsd:string" />
    <xsd:element minOccurs="1" maxOccurs="1"
      name="address1" type="xsd:string" />
    <xsd:element minOccurs="0" maxOccurs="1"
      name="address2" type="xsd:string" />
    <xsd:element minOccurs="0" maxOccurs="1"
      name="address3" type="xsd:string" />
    <xsd:element minOccurs="1" maxOccurs="1"
      name="city" type="xsd:string" />
    <xsd:element minOccurs="1" maxOccurs="1"
      name="state" type="xsd:string" />
    <xsd:element minOccurs="1" maxOccurs="1"
      name="zip" type="xsd:string" />
    <xsd:element minOccurs="1" maxOccurs="1"
      name="phoneAreaCode" type="xsd:int" />
    <xsd:element minOccurs="1" maxOccurs="1"
      name="phonePrefix" type="xsd:int" />
    <xsd:element minOccurs="1" maxOccurs="1"
      name="phoneSuffix" type="xsd:int" />
    <xsd:element minOccurs="1" maxOccurs="1"
      name="accountToPay" type="xsd:string" />
    <xsd:element minOccurs="1" maxOccurs="1"
      name="nameOnBill" type="xsd:string" />
    <xsd:element minOccurs="1" maxOccurs="1"
      name="alias" type="xsd:string" />
  </xsd:sequence>
</xsd:complexType>
```

To mail payments, you can imagine all the required data needed, which is shown in boldface in [Listing 2](#). Notice that `address2` and `address3`, the supplemental extra address lines, aren't required. Next, define the loan application complex type.

Loan application

How would we survive if banks didn't give out loans? The data structures defined here contain loan application information and loan status information. Start out by defining the loan application complex type, as shown in [Listing 3](#).

Listing 3. A loan application

```

<xsd:complexType name="LoanApplication">
  <xsd:sequence>
    <xsd:element minOccurs="1" maxOccurs="1"
      name="loanType" type="xsd:string" />
    <xsd:element minOccurs="1" maxOccurs="1"
      name="amount" type="xsd:float" />
    <xsd:element minOccurs="1" maxOccurs="1"
      name="termMonths" type="xsd:int" />
    <xsd:element minOccurs="1" maxOccurs="1"
      name="joint" type="xsd:boolean" />
    <xsd:element minOccurs="1" maxOccurs="1"
      name="percentFinanced" type="xsd:int" />
  </xsd:sequence>
</xsd:complexType>

```

Each field is comprised of data typically needed on a loan application. Next, add the loan application status to your WSDL, as shown in [Listing 4](#).

Listing 4. The status of a loan application

```

<xsd:complexType name="LoanApplicationStatus">
  <xsd:sequence>
    <xsd:element minOccurs="1" maxOccurs="1"
      name="loanNumber" type="xsd:string" />
    <xsd:element minOccurs="1" maxOccurs="1"
      name="approved" type="xsd:boolean" />
    <xsd:element minOccurs="1" maxOccurs="1"
      name="denied" type="xsd:boolean" />
  </xsd:sequence>
</xsd:complexType>

```

Every loan gets a number and approved or denied status associated with it. If it has been neither approved nor denied, then it hasn't been processed yet. Next up are the structures relating to paying bills.

Bill payments

Paying bills requires you to enter information, such as how much to pay, to whom, and whether it's a recurring payment. The two structures shown in the next two listings take care of that for you. Add the bill payment complex type to your WSDL file, as shown in [Listing 5](#).

Listing 5. A bill payment

```

<xsd:complexType name="BillPayment">
  <xsd:sequence>
    <xsd:element minOccurs="1" maxOccurs="1"
      name="payeeName" type="xsd:string" />
    <xsd:element minOccurs="1" maxOccurs="1"
      name="account" type="xsd:int" />
    <xsd:element minOccurs="1" maxOccurs="1"
      name="amount" type="xsd:float" />
    <xsd:element minOccurs="1" maxOccurs="1"

```

```

        name="date" type="xsd:date" />
<xsd:element minOccurs="1" maxOccurs="1"
  name="notes" type="xsd:string" />
<xsd:element minOccurs="1" maxOccurs="1"
  name="recurring" type="xsd:boolean" />
<xsd:element minOccurs="0" maxOccurs="1"
  name="recurringData"
  type="xsd1:RecurringBillPayment" />
</xsd:sequence>
</xsd:complexType>

```

The amount shown here is the amount you pay. If it's a recurring payment, it's how much you pay each time, so the date would then be the start date. Also, notice here that `recurring` specifies whether the payment is a recurring type, and if it is, `recurringData` is defined (notice that its `minOccurs` is defined as 0). The `recurringData` element refers to the `RecurringBillPayment` complex type, which is shown in [Listing 6](#). Add it to your WSDL.

Listing 6. Adding on recurring bill payment data

```

<xsd:complexType name="RecurringBillPayment">
  <xsd:sequence>
    <xsd:element minOccurs="1" maxOccurs="1"
      name="finalPayment" type="xsd:float" />
    <xsd:element minOccurs="1" maxOccurs="1"
      name="frequency" type="xsd:string" />
    <xsd:element minOccurs="1" maxOccurs="1"
      name="totalNumberOfPayments" type="xsd:int" />
    <xsd:element minOccurs="1" maxOccurs="1"
      name="recurringType" type="xsd:string" />
  </xsd:sequence>
</xsd:complexType>

```

The data in [Listing 6](#) simply completes the complex type shown in [Listing 5](#). Here, your last payment is the value of the `finalPayment` field, with `frequency` being how often you want to pay the bill. The `recurringType` field refers to whether the amount on the bill is fixed or not, which means you can change it before it's submitted the next time. Next you take a look at the various operations needed to be exposed through the WSDL.

Section 4. Expose the Web service: WSDL

Several operations make up the online banking Web service, and you'll learn about each one before you implement them. You'll get details about each operation, the messages, port types, and WSDL bindings.

Operations

Earlier you got [an overview of the online banking Web service](#). Now you'll learn about the operations needed to build such a Web service:

Login

- Logging in allows you to verify your credentials with the Web service and retrieve a token as proof that you've already been verified for future Web service requests.
- Two pieces of data are passed (username and password), and one is returned (a binary token).

Logout

- Logging out removes your token and username from the Web services list of authenticated tokens/usernames.
- Logging out only requires that you send your token; a confirmation Boolean variable is returned as a response.

LookupAccounts

- Look up your account information.
- Only the token is required, and a list of account numbers, types, and balances is returned.

LookupTransactions

- Look up transactions by date.
- Sending the token and account number, as well as an optional range of dates (date1 and date2), sends back an array of transaction complex types as a response (minOccurs="0" maxOccurs="unbounded").

SearchTransactions

- Look up transactions by check number or amount.
- This allows a more complex transaction search: Pass the account number, token, and either a range of check numbers or a range of amounts, and then an array of qualifying transaction complex types is returned as a response.

TransferFunds

- Transfer funds from one account to another.
- Transferring funds requires that you send the token, a source, a destination account number, and an amount. A confirmation Boolean variable is sent as a response.

LookupPayees

- View the payees associated with your online account.
- Send the token, and you receive an array of payee alias names as a response.

AddPayee

- Add payees to your online account.
- Send the token and a payee complex type, and a Boolean confirmation is sent as the response.

EditPayee

- Exactly like `AddPayee`, except the payee is updated rather than inserted into the underlying database.

ViewPayee

- View all the information associated with a payee in your account.
- Send the token and payee alias, and the payee complex type is retrieved as the response.

MakeBillPayment

- Pay a bill associated with one of the payees.
- Send the `BillPayment` complex type and token, and a confirmation Boolean variable is sent back as the response.

ChangeBillPayment

- Change a previously scheduled payment.
- Same as `MakeBillPayment`, except the payment is updated rather than inserted into the underlying database.

LookupPendingPayments

- View all pending payments.
- Send only the token, and you receive an array of `BillPayment` complex types as the response.

SubmitLoanApplication

- Submit a loan application.
- Send the `LoanApplication` complex type and token, and a confirmation `Boolean` variable is sent as the response.

ViewLoanApplicationStatus

- View the status of all loans associated with your online account.
- Send only your token, and you receive an array of `LoanApplicationStatus` complex types.

That defines them all. Keep in mind that there are way too many operations to define without running into redundancy and inducing boredom. Thus, you can refer to the WSDL file contained in the [Download](#) section later in this tutorial, and compare it to the operations listed above if you run into trouble. However, you'll walk through a few operations from start to finish as an example of what to do for the others.

Next you define the namespaces of your WSDL file.

Namespaces

The namespaces of your WSDL, though seemingly odd at first, are important, because they define the context of each tag throughout the WSDL. Place the following before the data structures you've already added to the `OnlineBanking.wsdl` file, as shown in [Listing 7](#).

Listing 7. Setting up the WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
  name="OnlineBanking"
  targetNamespace=http://www.example.com/OnlineBanking
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:tns="http://www.example.com/OnlineBanking"
  xmlns:xsd1="http://www.example.com/OnlineBanking/xsd"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" >
```

```

<!-- Types -->
<wsdl:types>
  <xsd:schema
    targetNamespace="http://www.example.com/OnlineBanking/xsd"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    elementFormDefault="qualified">

```

The `xsd1:` namespace specifies that all the structures defined in the `<xsd:schema` tag belong to the `xsd1:` namespace (`http://www.example.com/OnlineBanking/xsd`) and need to be referenced accordingly, similar to how the `RecurringBillPayment` complex type was referred to in [Listing 6](#). The `tns`, or `targetNamespace`, refers to all messages defined in the top-level WSDL (`http://www.example.com/OnlineBanking`). Next you define the element tags for requests and responses.

Requests

The rest of this section walks you through creating the WSDL for the `Login` and the `LookupTransactions` operations. The request is the SOAP object sent out to the Web service, and it can contain various internal hierarchies, such as the data structures you created earlier in this tutorial. Define the login request after the `RecurringBillPayment` complex type, as shown in [Listing 8](#).

Listing 8. The login request

```

<xsd:element name="Login">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element minOccurs="1" maxOccurs="1"
        name="username" type="xsd:string" />
      <xsd:element minOccurs="1" maxOccurs="1"
        name="password" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

```

Exactly one username and password are sent in this element. That's it! Next, define the lookup transactions element, as shown in [Listing 9](#).

Listing 9. The lookup transactions request

```

<xsd:element name="LookupTransactions">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element minOccurs="1" maxOccurs="1"
        name="accountNumber" type="xsd:int" />
      <xsd:element minOccurs="0" maxOccurs="1"
        name="date1" type="xsd:date" />
      <xsd:element minOccurs="0" maxOccurs="1"
        name="date2" type="xsd:date" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

```

```

        <xsd:element minOccurs="1" maxOccurs="1"
                    name="token" type="xsd:base64Binary" />
    </xsd:sequence>
</xsd:complexType>
</xsd:element>

```

Here you have a lookup transaction request that takes an `accountNumber` and the `token`, a `base64Binary` type. You'll see later that the `base64Binary` type translates to a `byte[]` in the Java programming language. Next, define the responses.

Responses

Clients create and send requests. The Web service receives the requests and creates a response according to the request. Define the login response, as shown in [Listing 10](#).

Listing 10. The login response

```

<xsd:element name="LoginResponse">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element minOccurs="1" maxOccurs="1"
                  name="valid" type="xsd:boolean" />
      <xsd:element minOccurs="1" maxOccurs="1"
                  name="token" type="xsd:base64Binary" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

```

A successful login response gives the value of `true` for the `valid` variable and a binary token in the `token` variable. Next, take a look at the lookup transactions response. Define it, as shown in [Listing 11](#).

Listing 11. The lookup transactions response

```

<xsd:element name="LookupTransactionsResponse">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element minOccurs="0" maxOccurs="unbounded"
                  name="transactions" type="xsd1:Transaction" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

```

This response returns an array of transaction elements. Notice the use of the `xsd1:` namespace here. You're simply referring to the `Transaction` complex type you defined earlier in [Listing 1](#) for the array type. Next -- the messages.

Messages

Messages are sent over to the Web service and back, and these messages contain the SOAP requests and responses you defined earlier in the [Responses](#) section. Define the `Login` request and response messages, as shown in [Listing 12](#).

Listing 12. Login request and response messages

```
<!-- Messages -->
<wsdl:message name="Login">
  <wsdl:part name="parameters" element="xsd1:Login" />
</wsdl:message>
<wsdl:message name="LoginResponse">
  <wsdl:part name="parameters" element="xsd1:LoginResponse" />
</wsdl:message>
```

It doesn't matter what you put in as the name attribute of the `<wsdl:part>` tag within the message, so the items to note are in boldface. The login request message refers to the `Login` element you defined previously, and the `LoginResponse` response message, too, refers to the `LoginResponse` element you defined previously. Next, define the lookup transactions request and response, as shown in [Listing 13](#).

Listing 13. Lookup transaction request and response messages

```
<wsdl:message name="LookupTransactions">
  <wsdl:part name="parameters" element="xsd1:LookupTransactions" />
</wsdl:message>
<wsdl:message name="LookupTransactionsResponse">
  <wsdl:part name="parameters"
    element="xsd1:LookupTransactionsResponse" />
</wsdl:message>
```

Now you'll begin to notice the redundancy. Just like the login request and response, you name the lookup transactions request and response and refer to the associated lookup transaction request and response element you just defined in this section. The other requests and responses should follow this same template. Next you define the port type for your online banking Web service.

Port types

Port types specify the operations that exist for a given Web service. Here you define the port type and the two login and lookup transactions operations. Define the port type and the login operation, as shown in [Listing 14](#).

Listing 14. The port type and login operation

```
<!-- Port type -->
<wsdl:portType name="OnlineBankingPortType">

  <wsdl:operation name="Login">
    <wsdl:input message="tns:Login" />
    <wsdl:output message="tns:LoginResponse" />
  </wsdl:operation>
```

First name the port type, which you need when you specify the WSDL bindings. The `tns:` prefix on `Login` and `LoginResponse` specifies that the messages should be found in the `tns` namespace, defined in [Listing 15](#). Now define the lookup transactions operation.

Listing 15. The lookup transactions operation

```
<wsdl:operation name="LookupTransactions">
  <wsdl:input message="tns:LookupTransactions" />
  <wsdl:output message="tns:LookupTransactionsResponse" />
</wsdl:operation>
```

Again, just like the `Login` operation, defining the `LookupTransactions` operation is very similar. Next up is the WSDL binding.

WSDL binding

The WSDL binding specifies the format of the messages, the protocol type for the Web service, and the operations exposed on this port. Define the WSDL binding and the binding specifics for the login operation, as shown in [Listing 16](#).

Listing 16. WSDL login operation bindings

```
<!-- Binding -->
<wsdl:binding name="OnlineBankingPortBinding"
  type="tns:OnlineBankingPortType">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
    style="document" />

  <wsdl:operation name="Login">
    <soap:operation
      soapAction="http://www.example.com/OnlineBanking/Login"
      style="document" />
    <wsdl:input>
      <soap:body use="literal" />
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal" />
    </wsdl:output>
  </wsdl:operation>
```

First, name the binding. Now notice the type. It refers to the port type you defined in [Listing 14](#). The `<soap:binding>` tag specifies the transport method and style for the

Web service, which are http and document, respectively. The `<soap:body>` specifies how the input and output messages are encoded. They will be encoded as literal in this case. Now define the WSDL bindings for the lookup transactions operation, as shown in [Listing 17](#).

Listing 17. WSDL bindings for the lookup transactions operation

```
<wsdl:operation name="LookupTransactions">
  <soap:operation
soapAction="http://www.example.com/OnlineBanking/LookupTransactions"
style="document" />
  <wsdl:input>
    <soap:body use="literal" />
  </wsdl:input>
  <wsdl:output>
    <soap:body use="literal" />
  </wsdl:output>
</wsdl:operation>
```

Again, this is similar to the login operation. Other operations should follow this template. Now on to the last piece of the WSDL, the service tag.

The service tag

The service tag defines what ports are exposed by this Web service. Define the service definition, as shown in [Listing 18](#).

Listing 18. The service definition

```
<!-- Service -->
<wsdl:service name="OnlineBankingService">
  <wsdl:port name="OnlineBankingPort"
    binding="tns:OnlineBankingPortBinding">
    <soap:address
location="http://localhost:8080/axis2/services/OnlineBankingService" />
  </wsdl:port>
</wsdl:service>
```

First you name the service and then define an exposed port, `OnlineBankingPort`, which refers to the WSDL binding you just defined in [Listing 17](#). The address specifies the URL on Geronimo in which to find the Web service, which is the URL to send and receive SOAP requests from.

The next section compiles the WSDL into Java code and deploys it on Geronimo.

Section 5. Compile and deploy the Web service

Now you create Java code that implements the Web service defined in the WSDL and deploy it on Geronimo with Axis2.

WSDL2Java

To create the Java implementation of your Web service, run the WSDL2Java tool on your WSDL file. Do so by typing the following on one line from the command line (in Microsoft® Windows®):

```
<axis2-binary-install-dir>/bin/WSDL2Java -d xmlbeans -uri  
OnlineBanking.wsdl  
-p com.ibm.axis2.onlinebanking -ss -sd
```

Or (in Linux®):

```
sh <axis2-binary-install-dir>/bin/WSDL2Java.sh -d xmlbeans  
-uri OnlineBanking.wsdl -p com.ibm.axis2.onlinebanking -ss -sd
```

Now type the following (on one line) to create a client stub for testing (in Windows):

```
<axis2-binary-install-dir>/bin/WSDL2Java -d xmlbeans -uri  
OnlineBanking.wsdl  
-p com.ibm.axis2.onlinebanking
```

Or (in Linux):

```
sh <axis2-binary-install-dir>/bin/WSDL2Java.sh -d xmlbeans  
-uri OnlineBanking.wsdl -p com.ibm.axis2.onlinebanking
```

That's it! The Java source files should now be created under the `./src` directory. A handy Apache Ant build.xml file will also be created in the current directory. Next you fill in the basic Web service skeleton.

Fill in the skeleton

To avoid getting nasty errors when you test the Web service, you need to fill in the Web service skeleton such that it never returns `null`. Take a look at the Web service skeleton located at `./src/com/ibm/axis2/onlinebanking/OnlineBankingPortTypeSkeleton.java`. Open it, and examine it. You should notice the `Login` method somewhere in there, as shown

in [Listing 19](#).

Listing 19. The original Web service

```
public com.example.www.onlinebanking.xsd.  
    LoginResponseDocument Login  
    (com.example.www.onlinebanking.xsd.  
     LoginDocument param8 ){  
    //Todo fill this with the necessary business logic  
    return null;  
}
```

Notice that the return type is a `LoginResponseDocument`, and the parameter to the method is a `LoginDocument` (the request). Fill in a simple definition by returning an empty `LoginResponseDocument`, as shown in [Listing 20](#).

Listing 20. Simple definition

```
public com.example.www.onlinebanking.xsd.  
    LoginResponseDocument Login  
    (com.example.www.onlinebanking.xsd.LoginDocument param8 ){  
    LoginDocument.  
        Login req =  
            param8.getLogin();  
  
    LoginResponseDocument res =  
        LoginResponseDocument.Factory.newInstance();  
    LoginResponseDocument.  
        LoginResponse res2 =  
            res.addNewLoginResponse();  
  
    return res;  
}
```

Get the actual `Login` request object from `param8` (may not be the same name generated by Axis2 on your system) in the first line. Next, create a new `LoginResponseDocument`, add a new `LoginResponse` in the third line, and return the empty `res` object, the `LoginResponseDocument`.

Also, remember to place at the top, under the package declaration, the following import statement:

```
package com.ibm.axis2.onlinebanking;  
import com.example.www.onlinebanking.xsd.*;  
...
```

This way you won't get `class not found` errors when you build and deploy the Web service.

Deploy Axis2 and the Web service on Geronimo

First off, you need to build the Web service. Go to the directory where you ran the WSDL2Java tool, and type:

```
ant jar.server
```

That'll create an OnlineBankingService.aar file at `./build/lib/OnlineBankingService.aar`. Now deploy this Web service Axis2 archive file by first starting Geronimo. Boot up Apache Geronimo by typing the following:

```
java -jar <geronimo-install-dir>/bin/server.jar
```

Now deploy the Axis2 .war file you downloaded from Apache by copying the `axis2.war` file over to the `<geronimo-install-dir>/deploy` directory. Open a browser to `http://localhost:8080/axis2/Login.jsp`.

Log in using `admin` and `axis2` as the username and password, respectively. Now point your browser to `http://localhost:8080/axis2/upload.jsp`.

Click **Browse**, and find the `OnlineBankingService.aar` file, then click **Upload**. In about 20 to 30 seconds, the Web service should be ready to go. In the meantime, the next section shows you how to create a simple client for testing your Web service's initial functionality.

Section 6. Create and test a simple client

To test your Web service, you define a simple client that tests it out and make sure that the SOAP messages are being transferred correctly.

Import class dependencies

The first step in creating the client is importing the class dependencies. Create a file, `Client.java`, and place it at `./src/Client.java`. Begin defining it, as shown in [Listing 21](#).

Listing 21. Importing class dependencies

```
import com.example.www.onlinebanking.xsd.*;
import com.ibm.axis2.onlinebanking.*;

import java.util.*;

public class Client{

    public static void main(java.lang.String args[]){
```

Most of the objects you'll use are in the first two import statements, the client stub in particular, as well as the request and response objects and the other data structures you defined in the [Data structures](#) section. Next you begin defining the main function, starting by declaring the client stub.

Create the client stub

The client stub gives you a handle to the Web service now running on Geronimo. Create it, as shown in [Listing 22](#).

Listing 22. The client stub

```
public static void main(java.lang.String args[]){
    OnlineBankingPortTypeStub stub = null;
    try{
        stub = new OnlineBankingPortTypeStub(null,
        "http://localhost:8080/axis2/services/OnlineBankingService");
    ...
    } catch(Exception e){
        e.printStackTrace();
    }
}
```

You simply initialize the `OnlineBankingPortTypeStub` object by passing it the URL to the Web service. Next you create some initial functions that you'll use in calling and testing your Web service.

Call the Web service

Because this part can also be redundant, this panel walks you through only the API for calling the `Login` and `LookupTransactions` operations in the Web service. The rest should be easy with this template and the function calls defined in [Listing 25](#). Start by defining the method for calling the `Login` operation, as shown in [Listing 23](#).

Listing 23. Calling the Login operation

```
/* LOGIN */
public static byte[] login(OnlineBankingPortTypeStub stub,
                          String username, String password){
    try{
        LoginDocument reqDoc00 = LoginDocument.
            Factory.newInstance();
        LoginDocument.Login reqDoc01 = reqDoc00.addNewLogin();

        LoginResponseDocument resDoc00 = stub.Login(reqDoc00);
        LoginResponseDocument.LoginResponse resDoc01 =
```

```
        resDoc00.getLoginResponse();
        return null;
    } catch(Exception e){
        e.printStackTrace();
    }
    System.out.println("Authorization failed");
    return null;
}
```

This simply creates an empty `LoginDocument` request, sends it to the Web service, and receives an empty object. If all goes well, no errors should occur. [Listing 24](#) shows the sample principle for defining the `LookupTransactions` operation.

Listing 24. Calling the `LookupTransactions` operation

```
/* LOOKUPTRANSACTIONS */
public static boolean lookupTransactions(OnlineBankingPortTypeStub
                                       stub){
    try{
        LookupTransactionsDocument reqDoc00 =
            LookupTransactionsDocument.Factory.newInstance();
        LookupTransactionsDocument
            LookupTransactions reqDoc01 =
            reqDoc00.addNewLookupTransactions();

        LookupTransactionsResponseDocument resDoc00 =
            stub.LookupTransactions(reqDoc00);
        LookupTransactionsResponseDocument
            LookupTransactionsResponse resDoc01 =
            resDoc00.getLookupTransactionsResponse();

        return true;
    } catch(Exception e){
        e.printStackTrace();
    }
    return false;
}
```

Create similar templates for all the other operations so that you can test them in preparation for Part 2 of this tutorial series. You call the methods next by testing all of the Web service operations at once (see [Listing 25](#)).

Listing 25. Testing all the Web service operations at once

```
...
    login(stub);
    lookupAccounts(stub);
    lookupTransactions(stub);
    searchTransactions(stub);
    viewPayee(stub);
    makeBillPayment(stub);
    changeBillPayment(stub);
    lookupPendingPayments(stub);
    addPayee(stub);
    lookupPayees(stub);
    editPayee(stub);
    transferFunds(stub);
    submitLoanApplication(stub);
```

```
        viewLoanApplicationStatus(stub);
        logout(stub);
    } catch(Exception e){
        e.printStackTrace();
    }
}
```

Call each method, passing in the stub, and you're ready to build and run the client.

Build the client, and test the Web service

Now you'll want to build the Web service again. As before, type the following to do so:

```
ant jar.server
```

To run the client, add a new Ant task to the build.xml file, as shown in [Listing 26](#).

Listing 26. Adding a new Ant task for running the client

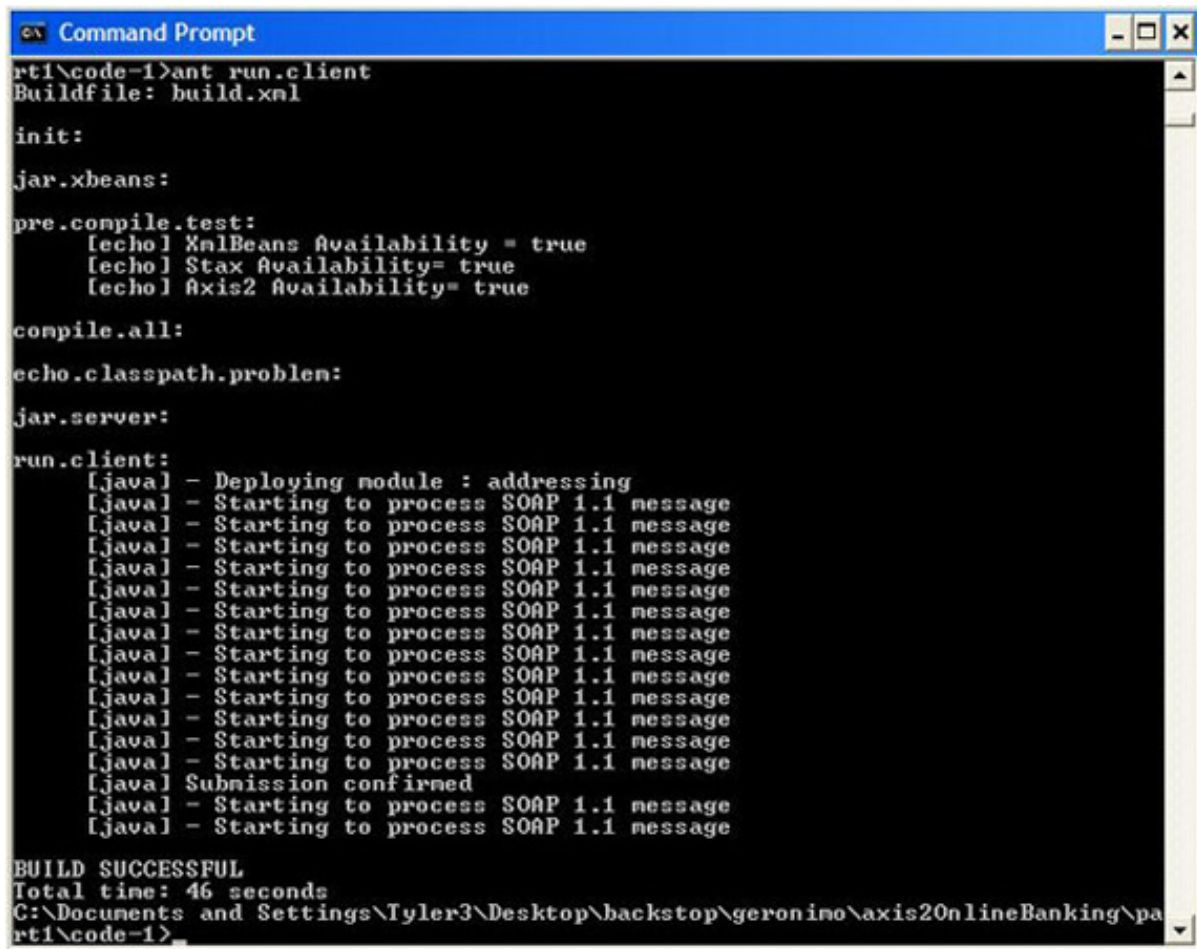
```
<!-- new target for running the client -->
<target depends="jar.server" name="run.client">
  <java classname="Client" fork="true">
    <classpath>
      <path refid="axis2.class.path"/>
      <path location="\${lib}/${name}.aar"/>
      <path location="\${lib}/${xbeans.packaged.jar.name}"/>
    </classpath>
  </java>
</target>
```

This creates a new task that runs the client, so run the client by typing:

```
ant run.client
```

You should get the output from the console, as shown in [Figure 1](#), from running the client.

Figure 1. Running the client



```
rt1\code-1>ant run.client
Buildfile: build.xml

init:

jar.xbeans:

pre.compile.test:
    [echo] XmlBeans Availability = true
    [echo] Stax Availability= true
    [echo] Axis2 Availability= true

compile.all:

echo.classpath.problen:

jar.server:

run.client:
    [java] - Deploying module : addressing
    [java] - Starting to process SOAP 1.1 message
    [java] - Starting to process SOAP 1.1 message
    [java] - Starting to process SOAP 1.1 message
    [java] - Starting to process SOAP 1.1 message
    [java] - Starting to process SOAP 1.1 message
    [java] - Starting to process SOAP 1.1 message
    [java] - Starting to process SOAP 1.1 message
    [java] - Starting to process SOAP 1.1 message
    [java] - Starting to process SOAP 1.1 message
    [java] - Starting to process SOAP 1.1 message
    [java] - Starting to process SOAP 1.1 message
    [java] - Starting to process SOAP 1.1 message
    [java] - Starting to process SOAP 1.1 message
    [java] - Starting to process SOAP 1.1 message
    [java] - Starting to process SOAP 1.1 message
    [java] Submission confirmed
    [java] - Starting to process SOAP 1.1 message
    [java] - Starting to process SOAP 1.1 message

BUILD SUCCESSFUL
Total time: 46 seconds
C:\Documents and Settings\Tyler3\Desktop\backstop\geronimo\axis2onlineBanking\pa
rt1\code-1>
```

The server console output (the console Geronimo is running in) should yield the output shown in [Figure 2](#).

Figure 2. Server output

```

Command Prompt - java -jar server.jar
WAR: geronino/servlets-examples-tomcat/1.1-SNAPSHOT/car
WAR: geronino/welcome-tomcat/1.1-SNAPSHOT/car

Web Applications:
http://your-a9279112e3:8080/
http://your-a9279112e3:8080/OnlineBankingClient
http://your-a9279112e3:8080/axis2
http://your-a9279112e3:8080/console
http://your-a9279112e3:8080/console-standard
http://your-a9279112e3:8080/daytrader
http://your-a9279112e3:8080/jsp-examples
http://your-a9279112e3:8080/juddi
http://your-a9279112e3:8080/ldap-demo
http://your-a9279112e3:8080/remote-deploy
http://your-a9279112e3:8080/servlets-examples

Geronimo Application Server started
22:58:39,218 INFO [Hot Deployer] Undeploying OnlineBankingClient.war
Undeployed com/ibm/axis2/onlinebanking

22:58:39,750 INFO [Hot Deployer] Undeploying axis2.war
Undeployed axis2

22:58:48,703 INFO [Hot Deployer] Deploying axis2.war
22:58:50,140 WARN [TomcatModuleBuilder] Web application does not contain a WEB-INF/geronino-web.xml deployment plan. This may or may not be a problem, depending on whether you have things like resource references that need to be resolved. You can also give the deployer a separate deployment plan file on the command line.
Deployed axis2 @ http://your-a9279112e3:8080/axis2

22:59:13,062 INFO [DeploymentEngine] Deploying module : addressing
[JAM] Warning: You are running under a pre-1.5 JDK. JSR175-style source annotations will not be available
23:00:49,390 INFO [StAXSOAPModelBuilder] Starting to process SOAP 1.1 message
23:00:53,703 INFO [StAXSOAPModelBuilder] Starting to process SOAP 1.1 message
23:01:01,156 INFO [StAXSOAPModelBuilder] Starting to process SOAP 1.1 message
23:01:03,078 INFO [StAXSOAPModelBuilder] Starting to process SOAP 1.1 message
23:01:07,921 INFO [StAXSOAPModelBuilder] Starting to process SOAP 1.1 message
23:01:09,593 INFO [StAXSOAPModelBuilder] Starting to process SOAP 1.1 message
23:01:11,046 INFO [StAXSOAPModelBuilder] Starting to process SOAP 1.1 message
23:01:12,562 INFO [StAXSOAPModelBuilder] Starting to process SOAP 1.1 message
23:01:14,078 INFO [StAXSOAPModelBuilder] Starting to process SOAP 1.1 message
23:01:15,312 INFO [StAXSOAPModelBuilder] Starting to process SOAP 1.1 message
23:01:17,234 INFO [StAXSOAPModelBuilder] Starting to process SOAP 1.1 message
23:01:20,484 INFO [StAXSOAPModelBuilder] Starting to process SOAP 1.1 message
23:01:23,171 INFO [StAXSOAPModelBuilder] Starting to process SOAP 1.1 message
23:01:24,531 INFO [StAXSOAPModelBuilder] Starting to process SOAP 1.1 message
23:01:26,500 INFO [StAXSOAPModelBuilder] Starting to process SOAP 1.1 message

```

Now, you might notice that it takes a long time to run the client the first time, which is because the Web service needs to load and cache itself on your machine. Future runs of the client are much faster, and you'll notice an incredible increase in speed.

So far, so good! The Web service is working!

Section 7. Summary

You've completed the WSDL, all the way from the primitive SOAP structures to the

WSDL bindings. You've also successfully compiled it into Java code using Axis2's WSDL2Java tool, and you've tested and deployed it on Geronimo.

Come back for Part 2 to develop the Java code, generated by Axis2, and create a full-fledged online banking Web service. Soon after, in Part 3, you'll create a Web-based client for communicating with the Axis2 Web service, so stay tuned.

Downloads

Description	Name	Size	Download method
Part 1 source code	os-ag-onbank1.zip	9KB	HTTP

[Information about download methods](#)

Resources

Learn

- Check out an article on [developing Web services using Axis2](#) (developerWorks, November 2005).
- Access the [Apache Ant manual](#) and the [Apache Axis2 0.94 documentation](#).
- Get [Apache Axis documentation](#).
- Visit the [Apache Geronimo home page](#).
- Get involved with Apache Geronimo by joining the [Apache Geronimo mailing lists](#).
- Join the [Axis2 mailing lists](#).
- See what developer.com has to say about the [new Axis2 deployment architecture](#).
- Visit the [Axis2 Wiki](#).
- Check out the developerWorks [SOA and Web services zone](#), which provides articles and tutorials on Web services and Web services standards.
- Get a guide to the theory and implementation of WSDM in the tutorial, "[Understand Web Services Distributed Management \(WSDM\)](#)" (developerWorks, July 2005).
- Visit the [developerWorks Open source zone](#) for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.
- Check out the developerWorks [Apache Geronimo project area](#) for articles, tutorials, and other resources to help you get started developing with Geronimo today.
- Check out the [IBM Support for Apache Geronimo](#) offering, which lets you develop Geronimo applications backed by world-class IBM support.
- Find helpful resources for beginners and experienced users at the [Get started now with Apache Geronimo](#) section of developerWorks.
- Browse all the [Apache articles](#) and [free Apache tutorials](#) available in the developerWorks Open source zone.
- Browse for books on these and other technical topics at the [Safari bookstore](#).

Get products and technologies

- Download [Apache Axis2](#), which provides a SOAP platform for building Web

services-based applications.

- Innovate your next open source development project with [IBM trial software](#), available for download or on DVD.
- Download [Apache Geronimo, Version 1.0](#).
- Download your free copy of [IBM WebSphere® Application Server Community Edition V1.0](#) -- a lightweight J2EE application server built on Apache Geronimo open source technology that is designed to help you accelerate your development and deployment efforts.

Discuss

- [Participate in the discussion forum for this content](#).
- Stay up to date on Geronimo developments at the [Apache Geronimo blog](#).

About the author

Nicholas Chase

Nicholas Chase has been involved in Web site development for companies such as Lucent Technologies, Sun Microsystems, Oracle, and the Tampa Bay Buccaneers. Nick has been a high school physics teacher, a low-level radioactive waste facility manager, an online science fiction magazine editor, a multimedia engineer, an Oracle instructor, and the chief technology officer of an interactive communications company. He is the author of several books, including [XML Primer Plus](#).