

# Deploy OpenLaszlo apps on Apache Geronimo with Tomcat

Skill Level: Intermediate

[Tyler Anderson \(tyleranderson5@yahoo.com\)](mailto:tyleranderson5@yahoo.com)  
Freelance writer

14 Mar 2006

With the rise of the popular rich user interface (UI) language, OpenLaszlo, and the parallel rise of Apache Geronimo, OpenLaszlo developers will want to deploy their applications on Geronimo. This tutorial documents the process necessary to configure the internals of Apache Tomcat within Geronimo to be able to run the OpenLaszlo application server, and hence deploy and run OpenLaszlo applications within Geronimo. You'll implement a sample OpenLaszlo application to prove the functionality of the OpenLaszlo application server running on Apache Geronimo.

## Section 1. Before you start

This tutorial is for you if you're interested in developing OpenLaszlo applications to run on Apache Geronimo. In it, you deploy the OpenLaszlo development kit on Geronimo. To prove OpenLaszlo functionality, you also develop an example application, written in OpenLaszlo, that communicates with a Web service that you construct using Axis2 and a Web Services Description Language (WSDL) file. You write the core Web service using Java code, then the Web service communicates with an Apache Derby database.

## About this tutorial

You're going to write an application in OpenLaszlo and deploy it, along with the OpenLaszlo application server, on Apache Geronimo. The example application is a client that communicates with a Web service by asking questions. The Web service responds by providing a random answer supplied to it by Geronimo's built-in Derby

database. Unique questions asked also get stored in the Derby database for later analysis.

## Prerequisites

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

- Apache Geronimo -- After you download the Java™ platform, [download Geronimo 1.0](#). Select the Geronimo 1.0 with Tomcat distribution in either the .zip (Microsoft® Windows® or Linux®) or the .tar.gz (Linux) format from the Binaries section.
- OpenLaszlo -- OpenLaszlo is the new buzz word for Web-based rich UIs. This tutorial uses this language, so you need to download the absolute latest version (this tutorial uses version 3.2) of the [OpenLaszlo development kit](#). Find it by scrolling down to the Nightly Builds section, then select the version in the Dev Kit column (currently about 50MB).
- Apache Axis2 -- In this tutorial, you build a Web service from WSDL using Axis2. Axis2 is a great way to automatically build a Web service from WSDL. Also, make sure to get the WAR distribution. Download the [0.94 Axis2 .war file and binary distributions](#) from Apache.
- Apache Ant -- Axis2 creates a slick Ant build.xml file for building the Web service. [Download Ant](#) from Apache.
- Database -- This tutorial uses Geronimo's built-in database, Apache Derby. However, you still need to download the drivers you'll be using to connect to the database. Download the [IBM Cloudscape® \(IBM DB2® JDBC Universal Driver, for Cloudscape/Derby\)](#) from IBM.
- The Java platform -- Geronimo, OpenLaszlo, Axis2, and Derby require the Java platform. Java 1.4.08 or 1.4.09 is best for this tutorial. Download the [Java code](#) from Sun.

This tutorial assumes you have basic knowledge of Java syntax and coding. However, no knowledge of OpenLaszlo, Derby, Axis2, or Geronimo specifics are assumed.

---

## Section 2. Overview and setup

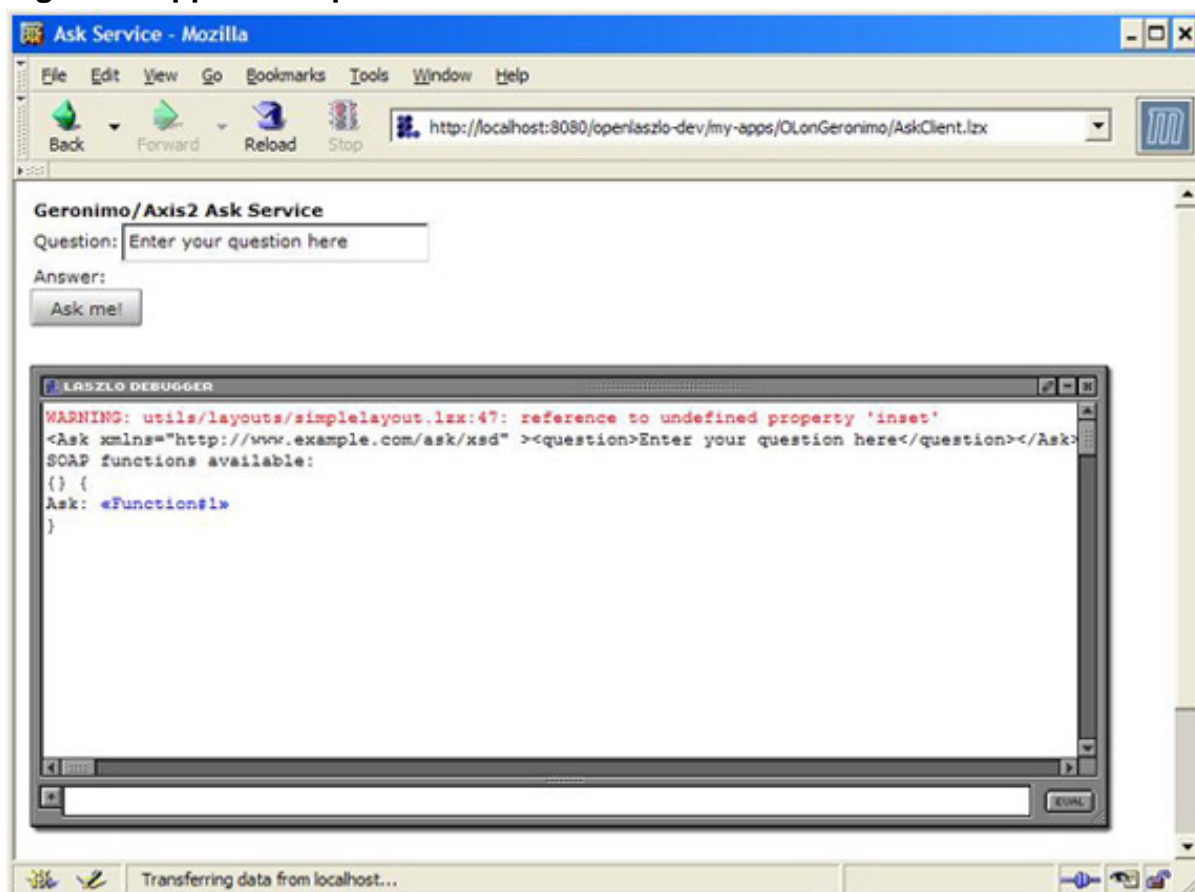
Here you learn more about how each technology and the application tie together.

You use Derby in this tutorial, so set that up here.

## Overview

The final application has a place for entering questions that get answered via Web services and are then displayed back on the screen. You write the application using OpenLaszlo's built-in Simple Object Access Protocol (SOAP) functionality and Axis2. Axis2 houses the Web service as a separate deployment that can be accessed through the WSDL and OpenLaszlo's SOAP functions. Take a look at a preview of the application in [Figure 1](#).

**Figure 1. Application preview**



Notice that the answers received in the debug window get displayed in the label by the `Answer :` label.

First you need to set up the Derby database, then write a Web service in WSDL, which you compile into Java code using Axis2. Then you define the Web service in Java code, compile and bundle it with Axis2, and deploy it on Geronimo. The Web service is then alive and ready for use by your rich OpenLaszlo UI.

## Set up the Derby database

This tutorial uses Derby as packaged and built into Geronimo. After you've downloaded the latest Geronimo version from Apache, start Geronimo by typing:

```
java -Xmx500M -jar <geronimo-install-dir>/bin/server.jar
```

The `-Xmx500M` is needed; it allows Geronimo to have more memory, because OpenLaszlo is quite large. It should then take some time for the server to boot up and get ready. When it has completely installed, you need to connect to the Derby database using the `ij` tool. Type the following in a new console to start the `ij` tool:  
`java org.apache.derby.tools.ij.`

You should now be at the `ij` prompt. Type the following to connect to and create the LASZLO database:

```
connect
'jdbc:derby:net://localhost:1527/LASZLO;create=true:user=laszlouser;
password=laszlopass';
```

The above connects you to Derby, while creating a new database, LASZLO, with user (`laszlouser`) and password (`laszlopass`).

## Introduce the LASZLO database

Here you use the Derby database to store information received from the OpenLaszlo UI. Each time a user enters a question that hasn't been asked before, that question gets stored into the Derby database. Also, you use the database to store random answers, which the Web service later uses as a pool of answers to send back to the OpenLaszlo UI. Here are the tables you're going to use:

- `answers` -- This table stores all the answers you might like in your database. This tutorial simply returns a random answer. However, you could extend the functionality by adding logic that returns intelligent answers, similar to online bots.
- `questionsAsked` -- This table, empty initially, holds all the unique questions asked by visitors to your OpenLaszlo UI. You can then go into and analyze which questions were asked and the answers provided by the Web service.

That completes your list of tables. Let's move on to creating them.

## Create the tables

With the Derby LASZLO database created, you can begin creating your tables. At the same `ij` prompt you connected to the database with, type the following to create the answers table: `create table answers (answer varchar(255));` .

This creates the answers table where you can add as many fun and crazy answers as you want. Now create the questionsAsked table:

```
create table questionsAsked (question varchar(1024),
                             answerGiven varchar(255),
                             primary key(question, answerGiven));
```

This table has two primary keys, because you don't need to see duplicates in your table as you analyze it. Next you initialize the answers table with initial data.

## Initialize the answers table

Here's where you add answers to the answers table. You can use the SQL below and add some other ones if you like. Go back to the `ij` prompt, and type the following to initialize the answers table, as shown in [Listing 1](#).

### Listing 1. Initializing the answers table with answers

```
insert into answers values('oh, I dont know that one'),
                          ('The answer is that you must research'),
                          ('Yes'),
                          ('HMMMMMMMMMMMMMMMM *My brain has timed out*'),
                          ('Maybe'),
                          ('No'),
                          ('Clearly, my responses are limited, \
so you must ask the right questions :P'),
                          ('Thats the question Ive been asking all my life');
```

Here you have eight unique answers. You can add more answers at any time using the exact same SQL as above; just change the answers in between the quotes. Next, you begin writing the WSDL for the Web service.

---

## Section 3. Write the Web service in WSDL

Your OpenLaszlo application relies on the WSDL you write in this section, as would any other standard application that implements SOAP. The WSDL defines what types of operations are sent and what operations are received, as well as the URL that the SOAP messages are to be sent to. Note that the WSDL you create for this tutorial is simple and is meant to show you how to quickly get a basic Web service up using Axis2. First start by setting up the namespaces.

## Set up the namespaces

WSDL defines several namespaces. They tell applications, such as Axis2, what data you're referencing and so on. Create a file named `ask.wsdl`, and begin defining it, as shown in [Listing 2](#). This sets up your namespace and the other namespaces that you reference throughout the example. Note that you use the `wSDL` namespace on nearly every tag. The `targetNamespace` defines the namespace of this WSDL document. You'll see others used throughout this section.

### Listing 2. Starting the WSDL definition

```
<?xml version="1.0" encoding="utf-8"?>
<wSDL:definitions
  name="ask"
  targetNamespace="http://www.example.com/ask"
  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/ask"
  xmlns:xsd="http://www.example.com/ask/xsd"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:wSDL="http://schemas.xmlsoap.org/wSDL/">
  ...
</wSDL:definitions>
```

Next you define your schema.

## Define the schema

WSDL has a standard basic schema defined in your `schemas` xmlns, `http://www.w3.org/2001/XMLSchema`. It contains simple data types, such as string, int, boolean, and so on. Here's where you define your own schema by using the default schema as building blocks. Continue defining the `ask.wsdl` file by adding a schema, as shown in [Listing 3](#).

### Listing 3. Defining a schema

```
...
  xmlns:wSDL="http://schemas.xmlsoap.org/wSDL/">
  <wSDL:types>
    <xsd:schema
```

```

    elementFormDefault="qualified"
    targetNamespace="http://www.example.com/ask/xsd"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    <xsd:element name="Ask">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element minOccurs="1" maxOccurs="1"
            name="question" type="xsd:string" />
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="AskResponse">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element minOccurs="1" maxOccurs="1"
            name="answer" type="xsd:string" />
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
</wsdl:types>
...

```

Notice it has two main elements: Ask and AskResponse. Ask contains one element -- the question of type string. AskResponse also contains one string -- the answer. Ask is the incoming message, and AskResponse is the outgoing message, as you'll see next.

## Ask messages

Most Web service transactions consist of an incoming message and an outgoing message. Define your messages, as shown in [Listing 4](#).

### Listing 4. Defining the messages

```

...
</wsdl:types>

<wsdl:message name="Ask">
  <wsdl:part name="parameters" element="xsd1:Ask" />
</wsdl:message>
<wsdl:message name="AskResponse">
  <wsdl:part name="parameters" element="xsd1:AskResponse" />
</wsdl:message>
...

```

Here you've defined two messages: Ask and AskResponse. They don't have to be the same name as the element they reference in the `element` attribute of the `part` tag. Notice the use of the `xsd1:` namespace here, which is the schema you recently defined. That defines all the messages for your Web service.

## Ask port type

Every message comes and goes through an operation on a port. Here you define a port type and an operation on the port type. A port type can have multiple operations; however, due to the simplicity of this Web service, you should only define one. Define your port type, as shown in [Listing 5](#).

### Listing 5. Defining the operation in your port type

```
...
</wsdl:message>

<wsdl:portType name="AskPortType">
  <wsdl:operation name="Ask">
    <wsdl:input message="tns:Ask" />
    <wsdl:output message="tns:AskResponse" />
  </wsdl:operation>
</wsdl:portType>
...
```

You have defined an Ask operation within the AskPortType with incoming messages defined as tns:Ask -- the tns: being the namespace you defined in [Listing 2](#) with the output message tns:AskResponse. Next is the port binding.

## Ask port binding

As you bind a port type, you specify the operations it has and the transportation method. Define your port type binding, as shown in [Listing 6](#).

### Listing 6. Binding a port type

```
...
</wsdl:portType>

<wsdl:binding name="AskPortBinding"
  type="tns:AskPortType">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
    style="document" />
  <wsdl:operation name="Ask">
    <soap:operation soapAction="http://www.example.com/ask/Ask"
      style="document" />
    <wsdl:input>
      <soap:body use="literal" />
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal" />
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
...
```

Notice that the opening wsdl:binding tag has a type attribute that references the AskPortType you just created. Next, the soap:binding defines how the SOAP message is transported and the style of that transportation. You have one operation, Ask. A standard operation has three children. The first is how the operation is handled via SOAP. Each operation needs an action, which you've defined as

`http://www.example.com/ask/Ask` here. The other two children define only an input and an output. The WSDL knows the contents of these through the `AskPortType` that this binding declaration is binding. Only binding specifics are added here. Next is the service definition.

## The ask service

This panel defines the service that your OpenLaszlo application looks at to determine what URL to send its SOAP requests to. Complete your WSDL definition by defining the service, as shown in [Listing 7](#).

### Listing 7. Defining the AskService

```
...
</wsdl:binding>

<wsdl:service name="AskService">
  <wsdl:port name="AskPort" binding="tns:AskPortBinding">
    <soap:address
      location="http://localhost:8080/axis2/services/Ask" />
  </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

A service can define several ports, with each port having a port binding. Notice that the above `AskPort` has the `AskPortBinding` you defined earlier. The only thing left is to define where the Web service is, defined in the `soap:address` tag. Now your OpenLaszlo application knows where to send SOAP messages. Next you use Axis2 to turn this WSDL into a live Web service.

---

## Section 4. Use Axis2 to develop the Web service

Axis2 is a handy tool that helps automate the process of developing a Web service and making it alive to the online world. Here you use Axis2 to create the service skeleton and fill it in with Java code.

### Create the Web service stubs from WSDL

With the WSDL completely defined, you're now going to create the Java classes for your Web service. Axis2 comes with a handy tool for doing just that. You can find it in the `<axis2-bin-install-dir>/bin` as either a Linux or Windows script. Create your Java classes using the `WSDL2Java` tool by typing the following:

```
WSDL2Java -uri wsdl\ask.wsdl -ss -p com.ibm.laszlo.ask
```

Note that the above places the code at the following base package: `com.ibm.laszlo.ask`. The `-ss` switch creates the service skeleton you look at next.

## The service skeleton

The service skeleton is the only auto-generated file that you manipulate. Why are all these files needed? It all comes down to the culture of software development: Why do something on your own in one hour when you can get it auto-generated in one minute?

The `AskPortTypeSkeleton.java` file is the service skeleton I just referred to. You put all your Java code for the Web service in this file (see [Listing 8](#)).

### Listing 8. The Axis2 auto-generated service skeleton

```
/**
 * AskPortTypeSkeleton.java
 *
 * This file was auto-generated from WSDL
 * by the Apache Axis2 version: 0.94 Jan 11, 2006 (08:01:58 LKT)
 */
package ibm.laszlo.ask;
/**
 * AskPortTypeSkeleton java skeleton for the axisService
 */
public class AskPortTypeSkeleton {

    /**
     * Auto generated method signature
     *
     * @param param0
     */
    public com.example.www.ask.xsd.AskResponseDocument Ask
        (com.example.www.ask.xsd.AskDocument param0 ){
        //Todo fill this with the necessary business logic
        return null;
    }
}
```

Notice that [Listing 8](#) has an `Ask` method just like the `Ask` operation you defined in the `wsdl:binding` tag in your WSDL definitions file. The `AskDocument` is a wrapper for the `Ask` element you defined in your schema. You'll see how this works in this section. Next you import class dependencies.

## Import dependencies

You use several classes in this Web service, so you import them here for usage later, which is a standard Java practice. Begin modifying the auto-generated skeleton, `AskPortTypeSkeleton.java`, found at `com/ibm/laszlo/ask/AskPortTypeSkeleton.java`, as shown in [Listing 9](#).

### Listing 9. Importing class dependencies

```
package com.ibm.laszlo.ask;

import com.example.www.ask.xsd.*;

import java.sql.*;
import java.util.Properties;
import java.lang.String;
import java.lang.Double;

/**
 * Auto generated java skeleton for the service by the Axis code generator
 */
...
```

The first two import statements import the `AskDocument` and `AskResponseDocument` classes, as well as all the classes defined in the default schema (`org.xmlsoap.schemas.*`). The next two import the SQL classes you need to connect to the database, and the last two tell the compiler that `String` means `java.lang.String`, a standard Java class, and not the `String` class defined by the schema. Next you set up the DB2 driver.

## Set up the DB2 driver

This Web service connects to Derby using the IBM DB2 Universal Driver (see [Prerequisites](#)). Continue defining the skeleton file, as shown in [Listing 10](#).

### Listing 10. Setting up the IBM DB2 Universal Driver

```
public class AskPortTypeSkeleton {

    static Connection conn = null;

    static{
        Driver driver = null;
        try {
            driver = (Driver)
                (com.ibm.db2.jcc.DB2Driver.class).newInstance();
        } catch(Exception e) {
            throw new IllegalArgumentException("Unable to load, " +
                "instantiate, or register driver " +
                driver + ": " + e.getMessage());
        }
    }
    ...
}
```

The code in [Listing 10](#) sets up an instance of the `DB2Driver` class and makes sure no exceptions occur. Notice that this is being done in a `static{ }` statement,

which means that this code gets executed only once as the class loads.

## Create the database connection

With the driver loaded and ready, you're ready to connect to the database. Do so now, as shown in [Listing 11](#).

### Listing 11. Connecting to the LASZLO database

```
...
static Connection conn = null;
...
        driver + ": " + e.getMessage());
    }
    try {
        Properties prop = new Properties();
        prop.put("user", "laszlouser");
        prop.put("password", "laszlopass");
        conn = driver.
            connect("jdbc:derby:net://localhost:1527/LASZLO;",
                prop);
    } catch(Throwable e) {
        e.printStackTrace();
        System.out.println("Database connection failed");
    }
}
...
```

First you must define the user (laszlouser) and password (laszlopass) associated with your database in the `Properties` class that you defined back in the [Set up the Derby database](#) section. Next, you connect to the database, storing the connection in the static `conn` variable for later use by the `Ask` operation.

## Retrieve the question sent from the client

Now when the Laszlo client sends an `Ask` message to your Web service through the `Ask` operation, it's packaged as a SOAP message, and the Axis2 Web service receives and processes it. You can extract the asked question, as shown in [Listing 12](#).

### Listing 12. Extracting the question asked from the SOAP message.

```
    }
    public AskResponseDocument Ask
        AskDocument param0 ){
        AskDocument.Ask req = param0.getAsk();
        String questionAsked = req.getQuestion();
        System.out.println("QUESTION ASKED: " + req.getQuestion());
    }
    ...
```

Notice that the `Ask` class is housed within the `AskDocument` class and retrieved through `param0.getAsk()`. Now you can get the question and display it in the server output with the last two lines. Next you query the database and select a random answer.

## Retrieve a random answer from Derby

Now that you have the question, you could implement some artificial intelligence and get an intelligent answer. For simplicity, however, you'll just get an answer randomly from the database. Continue defining the `Ask` method, as shown in [Listing 13](#).

### Listing 13. Retrieving a random answer from the database

```
...
    System.out.println("QUESTION ASKED: " + req.getQuestion());

    String answer = null;
    boolean answerRetrieved = false;
    try{
        String sql = "select * from answers";
        PreparedStatement statement = conn.prepareStatement(sql);
        ResultSet result = statement.executeQuery();
        String answers[] = new String[100];
        int i = 0;
        while(result.next()){
            answers[i++] = result.getString("answer");
        }
        answer = answers[(int)(Math.random() * (double)i)];
        answerRetrieved = true;
    }
    ...
```

The answer you choose is stored in the `answer` variable, and you know an answer was successfully retrieved when `answerRetrieved` gets set to `true` on the last line. The answers are retrieved by the SQL statement stored in the `sql` variable. You can loop through the `ResultSet` returned, as shown in the `while` statement. Notice that each answer gets stored in the `answers` array, and a random answer is stored in the `answer` variable by obtaining an index using the `Math.random()` function, `[(int)(Math.random() * (double)i)]`. Next you store a (question, answer) *tuple*, an ordered set of values, back into Derby.

## Insert the question asked into Derby

After you've successfully retrieved an answer from the database, you can store the question and associated answer back into Derby for later analysis. You do this by setting up a prepared `insert` statement with two unknowns and setting the question and answer, respectively, as shown in [Listing 14](#).

### Listing 14. Storing the question and associated answer back into Derby

```
...
    answer = answers[(int)(Math.random() * (double)i)];
    answerRetrieved = true;

    sql = "insert into questionsAsked values (?, ?)";
    statement = conn.prepareStatement(sql);
    statement.setString(1, questionAsked);
    statement.setString(2, answer);
    statement.execute();
} catch (Exception e) {
    if (!answerRetrieved)
        e.printStackTrace();
}
...

```

After you set up and execute the SQL statement, the operation is complete. If any exceptions occur, let the user know if the answer was not properly set (`answerRetrieved` would equal `false`).

## Return a response

Now you return a response with your code, which funnels its way back to the calling SOAP client. Complete the `AskPortTypeSkeleton.java` file, as shown in [Listing 15](#).

### Listing 15. Returning an AskResponse

```
...
        e.printStackTrace();
    }

    AskResponseDocument res =
        AskResponseDocument.Factory.newInstance();
    AskResponseDocument.AskResponse res2 = res.addNewAskResponse();
    res2.setAnswer(answer);
    return res;
}
}

```

First, set up a wrapper for the `AskResponse` using the `AskResponseDocument` class, and then set the `answer` variable within the `AskResponse` class. Finally, return the `AskResponseDocument`. That completes the Web service! Next, you build the Java classes and deploy the example, packaged inside Axis2, on Geronimo.

## Build the Web service

Now you should have downloaded the WAR distribution from Apache, so you should have an `axis2.war` file. UnJAR it by typing:

```
jar -xvf axis2.jar
```

To build all of the Java classes, place the two .jar files contained in the IBM DB2 Universal Driver download in the <axis2-war-install-dir>/WEB-INF/lib and the <axis2-binary-install-dir>/lib directory. Then execute the following Ant command to build the Web service: `ant jar.server .`

This compiles the Java files in the Web service. Next you package and deploy the Web service in Axis2 and Geronimo, respectively.

## Deploy the Web service within Axis2

With all of your source compiled, you can now package it in Axis2 and then deploy Axis2 on Geronimo. First you need to set it up for deployment.

Your Ask Web service's Axis2 archive file should now exist in the ./build/lib directory created by Ant. Install it within Axis2:

```
cp ./build/lib/AskService.aar <axis2-war-install-dir>/WEB-INF/services
```

Now change directory to the <axis2-war-install-dir>. Now you're ready to reWAR Axis2: `jar -cf axis2.war * .`

Geronimo should still be running, so move the axis2.war to the <geronimo-install-dir>/deploy directory for deployment:

```
mv axis2.war <geronimo-install-dir>/deploy
```

Geronimo's new hot deployer should activate and deploy your Web service. You can make sure that it's active, along with its Ask operation, at the URL <http://localhost:8080/axis2/listServices.jsp>.

Sweet! Geronimo, the ask Web service, and Derby are ready to go. On to the Laszlo application!

---

## Section 5. Deploy OpenLaszlo on Geronimo

Before you start coding, you first need to deploy the OpenLaszlo application server and find the live directory of Geronimo where OpenLaszlo is so you can code there. That way you won't have to keep redeploying the OpenLaszlo WAR with each small change you make to your code, because the OpenLaszlo WAR file is hefty at around 50MB.

## Define OpenLaszlo's Geronimo deployment plan

Without a Geronimo deployment plan, OpenLaszlo won't function at all on Geronimo, so follow along carefully. First, unWAR the OpenLaszlo .war file you downloaded from the link in the [Prerequisites](#) section, and give it a Geronimo deployment plan:

```
mkdir openlaszlo-dev
cd openlaszlo-dev
jar -xvf openlaszlo-3.2.war
```

This puts the entire server into the openlaszlo-dev directory, which you should now refer to as <openlaszlo-install-dir>. Add a geronimo-web.xml file, and save it as <openlaszlo-install-dir>/WEB-INF/geronimo-web.xml, as shown in [Listing 16](#).

### Listing 16. Defining OpenLaszlo's Geronimo deployment plan

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
  xmlns="http://geronimo.apache.org/xml/ns/web"
  configId="openlaszlo-dev"
  >
  <hidden-classes><filter>org.jdom</filter></hidden-classes>

  <hidden-classes><filter>org.apache.axis</filter></hidden-classes>
  <context-root>/openlaszlo-dev</context-root>
  <context-priority-classloader>false</context-priority-classloader>
</web-app>
```

The reason you can't just deploy this WAR without a Geronimo deployment plan is because Geronimo and OpenLaszlo use two WARs with the same names. However, OpenLaszlo absolutely *has* to use its own version, but by default, Geronimo feeds OpenLaszlo its own. Thus, you need a Geronimo deployment plan. Most important in the plan, shown in [Listing 16](#), are the three lines in bold. The hidden-class and filter tags define which JARs cannot be loaded by a parent, meaning that they are loaded from OpenLaszlo's WEB-INF/lib directory as desired. The context-priority-classloader being false tells Geronimo to not make context the priority in loading classes, so other classes in OpenLaszlo's WEB-INF/lib directory, like log4j or others, aren't loaded specifically, but instead are loaded from Geronimo's repository.

## Deploy the OpenLaszlo application server

You can now reWAR OpenLaszlo and deploy it on Geronimo:

```
cp openlaszlo-dev.war <geronimo-install-dir>/deploy
```

**Important note:** Things will absolutely not work without the `geronimo-web.xml` Geronimo deployment file. Be sure you followed the steps shown earlier in this section.

You can now see Geronimo's hot deployer kick in and deploy it.

To test and make sure that you deployed OpenLaszlo successfully, open your browser to the URL `http://localhost:8080/openlaszlo-dev/my-apps/copy-of-hello.lzx`.

Hello Laszlo! should appear in small text at the upper left corner of the browser.

## OpenLaszlo's config-store location

Let's go find the live config-store directory where OpenLaszlo was deployed. This allows you to edit an actual directory and see your changes in real time without having to redeploy OpenLaszlo. Take a look at the `<geronimo-install-dir>/config-store/index.properties` file. A snippet of mine is shown in [Listing 17](#).

### Listing 17. Looking at index.properties

```
#Tue Jan 24 06:50:50 PST 2006
geronimo/j2ee-security/1.1-SNAPSHOT/car=4
openlaszlo-dev=75
...
geronimo/hot-deployer/1.1-SNAPSHOT/car=20
...
geronimo/webconsole-tomcat/1.1-SNAPSHOT/car=29
...
axis2=80
...
```

Here you can see the config-store directory number where each deployed module lies. The most important one is the `openlaszlo-dev` entry. [Listing 17](#) shows that the `openlaszlo-dev.war` file was deployed and is stored at `<geronimo-install-dir>/config-store/75/war`. Yours will show a different number, so go to the directory corresponding to the number shown in your `index.properties` file, and then go into the `my-apps` directory. Create a directory called `OLonGeronimo`. This is where you will store your OpenLaszlo application:

```
cd 75/war/my-apps
mkdir OLonGeronimo
```

Everything's all set. On my system, I'll create my OpenLaszlo application at the `<geronimo-install-dir>/config-store/75/war/my-apps/OLonGeronimo/` directory.

You can browse to the <http://localhost:8080/openlaszlo-dev/my-apps/OLonGeronimo> directory as proof that that is indeed where OpenLaszlo was deployed.

It's empty now, but it soon won't be!

---

## Section 6. Build the rich user interface using OpenLaszlo

OpenLaszlo is a new and emerging technology, great for designing rich UIs. Here you design a simple UI using OpenLaszlo and its built-in SOAP extensions to ask your Web service questions and receive very random answers.

### The basic layout and debug

Starting an OpenLaszlo application is simple. Define a file, `AskClient.lzx`, as shown in [Listing 18](#).

This defines the canvas. OpenLaszlo has incredible built-in debug support. All you have to do is set `debug="true"` as an attribute in the `canvas` tag, as shown in [Listing 18](#). To set the title at the top of the browser, `<title>` tags in HTML, set the `title` attribute in the `canvas` tag. The second line defines information on how the debug window will appear, as long as `debug="true"` in the `canvas` tag.

The third line, the `view` tag, is the form that will be used to communicate with the Axis2 Web service that you've created. Notice that it will start out as invisible, as defined by its `visible="false"` attribute. Later in this section you'll make it visible once the Web service loads. The `layout="simple"` attribute tells OpenLaszlo to place components one after another.

#### Listing 18. Defining the OpenLaszlo canvas

```
<canvas debug="true" width="750" title="Ask Service">
  <debug y="120" x="0" width="700" height="300" />

  <view x="10" y="10" name="form" visible="false" layout="simple" >
  ...
  </view>
</canvas>
```

### Heading and question box

Here you define more of the rich UI. Create a heading and editable text box for the

questions, as shown in [Listing 19](#).

### Listing 19. Creating a heading with the question box

```
<view x="10" y="10" name="form" visible="false" layout="simple" >
  <text><b>Geronimo/Axis2 Ask Service</b></text>

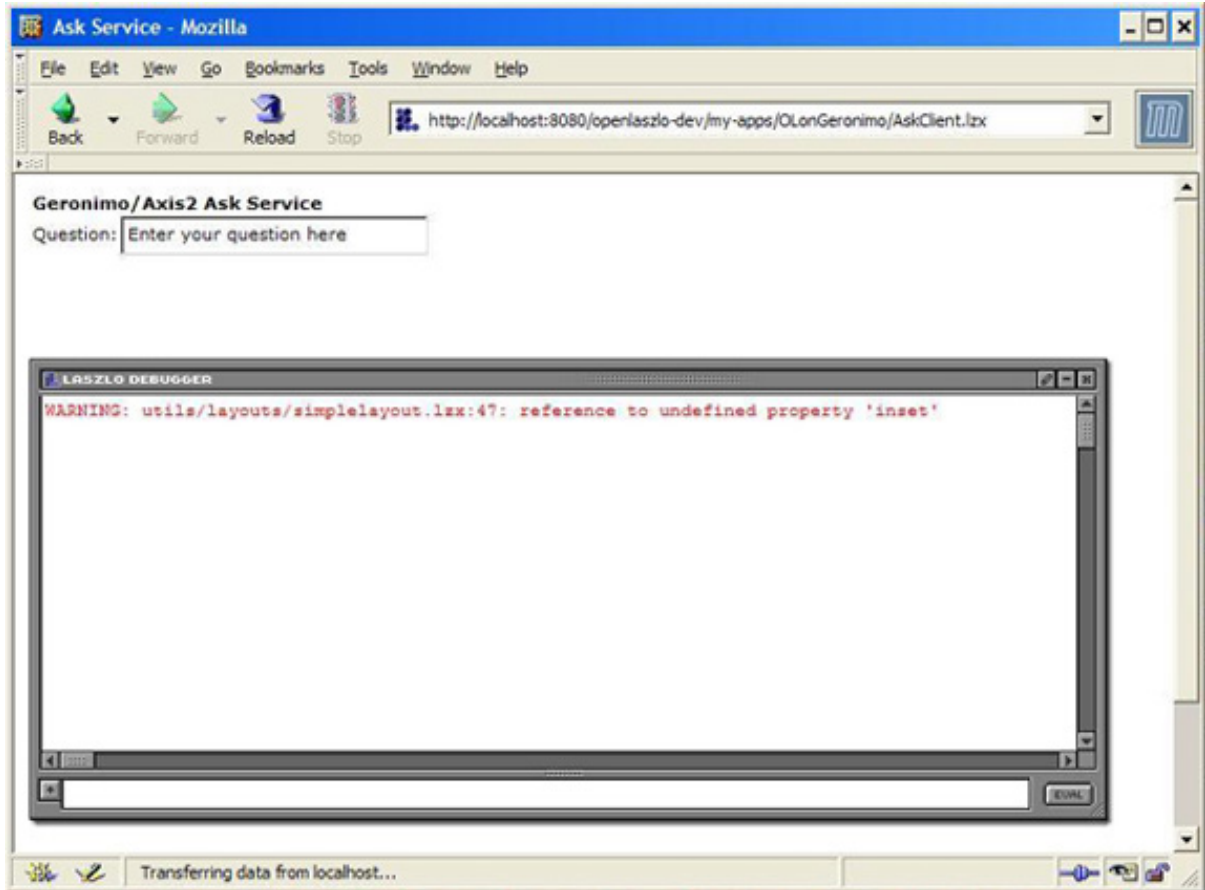
  <view layout="axis: x" >
    <text y="3">Question:</text>
    <edittext width="200" id="question"
      text="Enter your question here">
      <method event="onfocus">
        question.setText('');
      </method>
    </edittext>
  </view>
  ...
```

New code is noted in boldface. A heading is simple to create, as shown on the first line. The text box in the second view field is of notable value here. Its layout is a simple layout along the x axis, so each succeeding component is displayed horizontally along the x axis. Notice that the first tag is a label displaying `Question:`, and the `edittext` tag beside it is where users can enter their questions.

You can also define events. Notice that within the `edittext` tag, an `onfocus` event is tapped into. This means that whenever the text box is clicked and made the focus, the code within executes. Given that the default text is `Enter your question here`, when this box is then clicked, that default text clears so that the user can enter a question.

The current application can be seen in [Figure 2](#) (set `visible="true"` on the view tag with `name="form"`).

### Figure 2. Heading and question box



## Answer and button

The answer label is the label that stores the answer when it comes back from the Web service. The button invokes the OpenLaszlo SOAP client and sends your SOAP message off to the Axis2 Web service. Finish the form, as shown in [Listing 20](#).

### Listing 20. Defining the answer label and the submit button

```

...
</view>

<view layout="axis: x" >
  <text>Answer:</text>
  <text width="600" id="answer"/>
</view>

  <button text="Ask me!" onclick="canvas.ask.Ask.invoke()" />
</view>

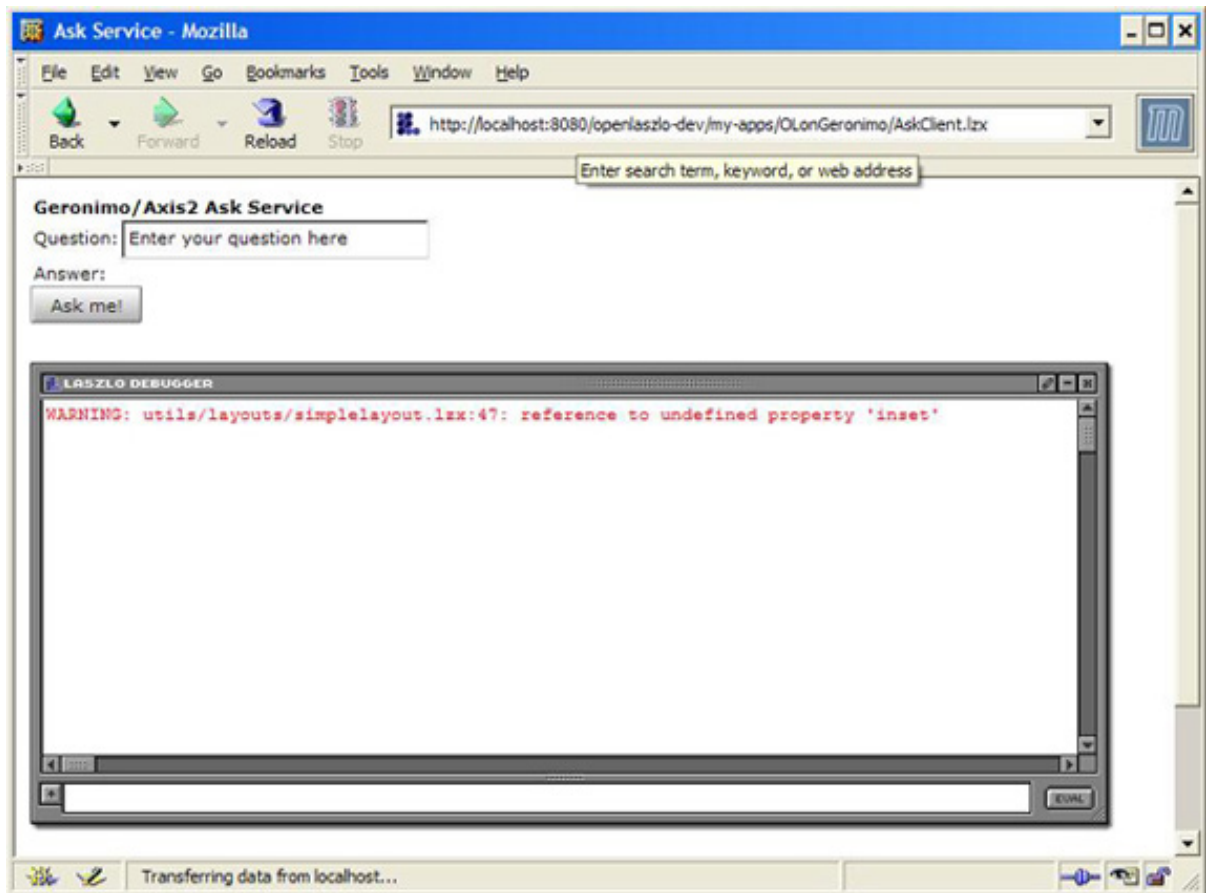
</canvas>

```

[Figure 3](#) shows a preview of the application with the above additions (again, set

visible="true" on the view tag with name="form").

**Figure 3. Adding the answer labels and button**



It doesn't do anything yet; in fact, pressing the button causes errors to show in the debug window, because there's a bit left to define.

## SOAP interface

It's time to set up the SOAP client. Define it above the form and below the debug tag, as shown in [Listing 21](#).

### Listing 21. Defining the SOAP client

```

...
<debug y="120" x="0" width="700" height="300" />
<soap name="ask"
      wsdl="http://localhost:8080/axis2/services/AskService?wsdl">
...
</soap>
<view x="10" y="10" name="form" visible="false" layout="simple" >
...

```

The rest of your code can now refer to the SOAP client through `canvas.ask`, which you can see in [Listing 21](#). The client also looks at the WSDL you put up in the axis2 WAR. Your client's all set up; now on to defining its internals.

## Handling load and data events

After the SOAP client loads the WSDL, the `onload` event fires. When you send a SOAP message to the service, the `ondata` event fires when a response comes back. Define the code for these two events within the SOAP client tags, as shown in [Listing 22](#).

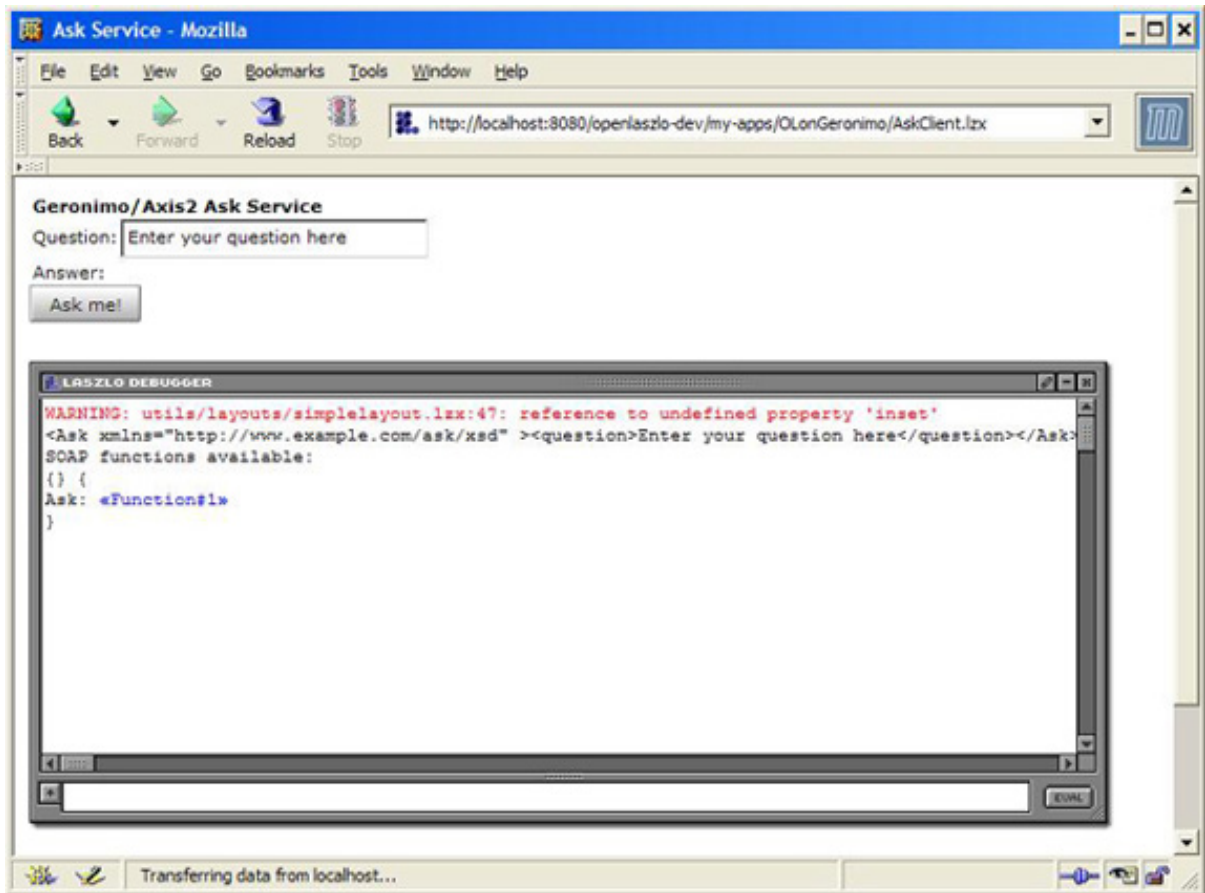
### Listing 22. Defining code for the onload and ondata events

```
<method event="onload">
  Debug.write('SOAP functions available:');
  Debug.inspect(this.proxy);
  canvas.form.setAttribute('visible', true);
</method>

<method event="ondata" args="value">
  answer.setText(value);
  Debug.write(value);
</method>
```

You can now set the `visible` attribute of the `view` tag named `form` back to `false`. The code in [Listing 23](#) sets it to `true` with the code in the second line in boldface. The first line in boldface displays the functions that are available in the Debug window. When data comes back from the service, it's written to the Debug window and to the `answer` label. You can see what I mean in [Figure 4](#).

### Figure 4. Loading the WSDL



You can see the `Ask` function show up in the Debug window. Pressing the button still causes errors, because there's one more step in sending off the SOAP message, which is up next.

## Create the SOAP document

Everything's all in place. Now you just need to create the `remotecall` and `createDocument` methods. Define them within the SOAP client, as shown in [Listing 23](#).

### Listing 23. Creating the `remotecall` and `createDocument` methods

```

                <remotecall funcname="Ask">
    <param value="$ { canvas.ask.createDocument(parent.name,
                                                question.text) }" />
</remotecall>

<method name="createDocument" args="soapCall, question">
  <![CDATA[
    if (soapCall == null) return;
    var s = '<' + soapCall + ' xmlns=" ' +

```

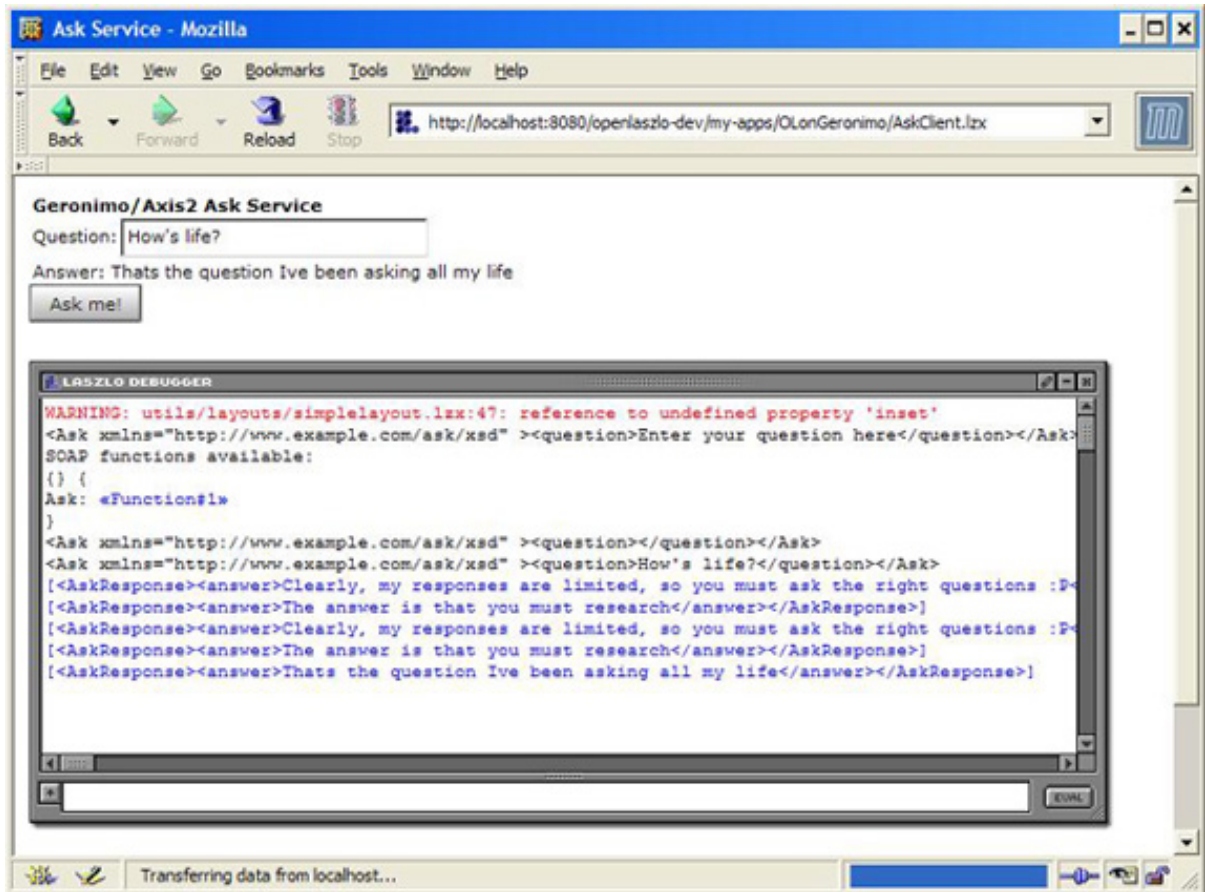
```

        'http://www.example.com/ask/xsd' + ' " >' +
        '<question>' + question + '</question>' +
        '</' + soapCall + '>';
    Debug.write(s);
    return s;
  ]]>
</method>

```

Remember that the `remotecall`, with `funcname="Ask"`, gets called by the button during its `onclick` event. This remote call sends the XML created by the `createDocument()` method to the Web service via SOAP. The data comes back in the `ondata` event, shown back in Listing 22. You can preview the final application in Figure 5.

**Figure 5. The final working app**



That completes the application! However, because things don't always work 100% the first time, let's take a look at the `onerror` and `ontimeout` events.

## Handling errors and timeout

Sometimes the remote Web service isn't set up or working properly, so defining the `onerror` and `ontimeout` events can be helpful for debugging. Define them within

the SOAP client with everything else, as shown in [Listing 24](#).

### Listing 24. Defining onerror and ontimeout events

```
<method event="onerror" args="error">
    Debug.write('error:', error);
</method>

<method event="ontimeout" args="error">
    Debug.write('timeout:', error);
</method>
```

Great, you've completed the application, complete with debugging! You can also take a look at the server output from the console you started Geronimo in for testing.

---

## Section 7. Summary

You've learned how to implement an OpenLaszlo application running on Apache Geronimo. You were able to reWAR OpenLaszlo with a Geronimo deployment plan. And you were able to develop your application within Geronimo's config-store, enabling you to avoid redeploying Geronimo with each change to your code. Finally, you developed your application to communicate with a Web service developed using Axis2 and WSDL via OpenLaszlo's built-in SOAP functionality.

Take a look at the [Resources](#) section to get involved more with OpenLaszlo.

## Downloads

Description	Name	Size	Download method
Example source code	OLonGeronimo.source.zip	5KB	<a href="#">HTTP</a>

[Information about download methods](#)

# Resources

## Learn

- Visit [OpenLaszlo.org](http://OpenLaszlo.org) for the latest OpenLaszlo downloads, news, articles, and announcements.
- Visit the [OpenLaszlo wiki](#).
- Read another developerWorks tutorial about using [OpenLaszlo and Eclipse](#).
- 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.

## Get products and technologies

- 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.](#)

## About the author

Tyler Anderson

Tyler Anderson graduated with a degree in computer science from Brigham Young University in 2004 and graduated with a Master of

Science degree in computer engineering in December 2005, also from Brigham Young University. He is currently an engineer for Stexar Corp., based in Beaverton, OR. You can reach the author at [tyleranderson5@yahoo.com](mailto:tyleranderson5@yahoo.com).