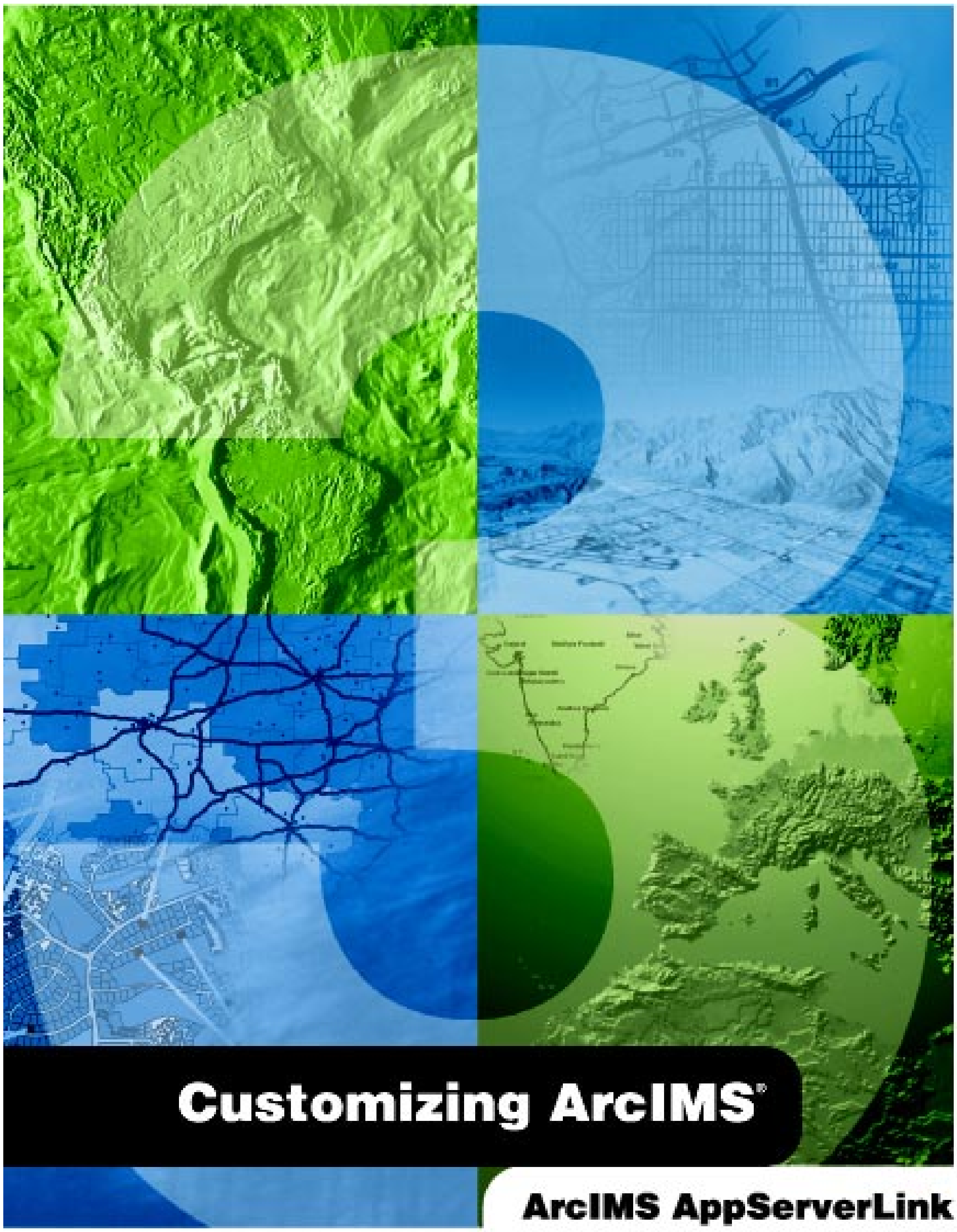


Customizing ArcIMS®

ArcIMS AppServerLink

ESRI ArcGIS™

Integrating a Java application with the ArcIMS AppServerLink

IN THIS DOCUMENT

Additional references

- *For information on installing the ArcIMS AppServerLink, see the section 'Step 3: Installing ArcIMS; ArcIMS Custom Application Server Connectors' in the* ArcIMS 3.1 Installation Guide*.*
- *Documentation describing the AppServerLink class and methods is located in the ArcIMS install directory under <ArcIMS Installation Directory>/Documentation/AppServerLink/Classes. Open Index.html to view the text.*
- *The* ArcXML Programmer's Reference Guide *is a necessary reference and can be found in the install directory under <ArcIMS Installation Directory>/Documentation/ArcXML_Guide. Open Arcxmlguide.htm. As this is an optional install component, you may need to install it.*
- Customizing ArcIMS, AppServerLink *is one in a series of books that describes customizing ArcIMS. This series also covers customization using the HTML and Java viewers and the ActiveX® and ColdFusion® Connectors.*

If you're using Sun™ Java™ to create a custom application and you want to incorporate geographic data in it, you can use the ESRI® ArcIMS® AppServerLink to connect your Java application directly to the ArcIMS Application Server. The Application Server is an integral part of ArcIMS. The Application Server processes client requests sent to it and, if necessary, routes those requests to the appropriate Spatial Server for further processing. By connecting your Java application to the Application Server, you connect to all the functionality provided by the Spatial Server—such as the ability to view and query your geographic data. Thus, you don't have to write your Java application completely from scratch; instead, you can rely on ArcIMS—through the AppServerLink—to provide the geographic operations you need. Here are a few examples of how you might use the AppServerLink:

- You're creating a Java application (like ArcExplorer™ 3) that allows people within your organization to browse maps.
- You're creating a Java applet that is embedded in the web pages of your corporate Intranet. The applet downloads to the client machine at run time, providing access to and a display of geographic data.
- You're using JavaServer Pages™ (JSP™) to build a web site that allows people outside your organization to access information that contains geographic content. Additionally, your application logic may reside in server-based resources—such as JavaBeans™—to provide additional Internet security.

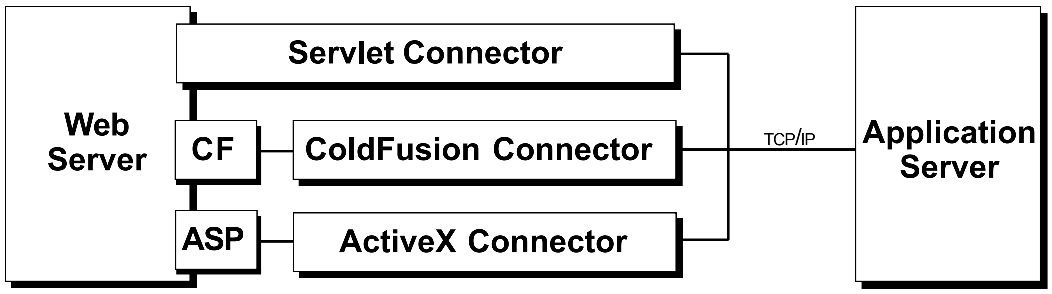
What is the AppServerLink?

The AppServerLink is a JavaBean; specifically, it is a reusable software component that exposes a set of methods and properties that you can set and access from your Java application. These methods allow you to programmatically establish a connection with an ArcIMS Application Server and begin sending ArcXML requests to it. Once the Application Server receives the request, it processes it and returns the appropriate response. With that response, your application can react accordingly.

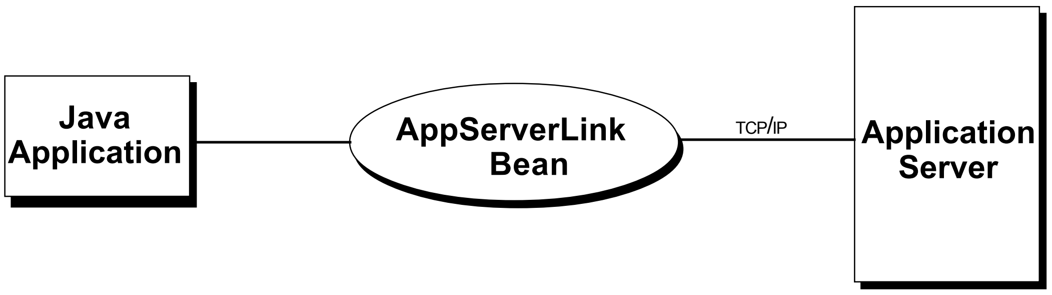
For example, suppose you've created a simple data browsing application that allows a user to pan and zoom around a map. To zoom in, the user pushes a button on your application's user interface. Your application translates that button press into an ArcXML request that gets sent via the AppServerLink to the Application Server, which in turn passes it on to the Spatial Server for processing. Based on this request, the Spatial Server generates a new zoomed-in display of the area of interest and informs the Application Server that the request has been processed. Your application can then display this new image back to the user.

Using the AppServerLink

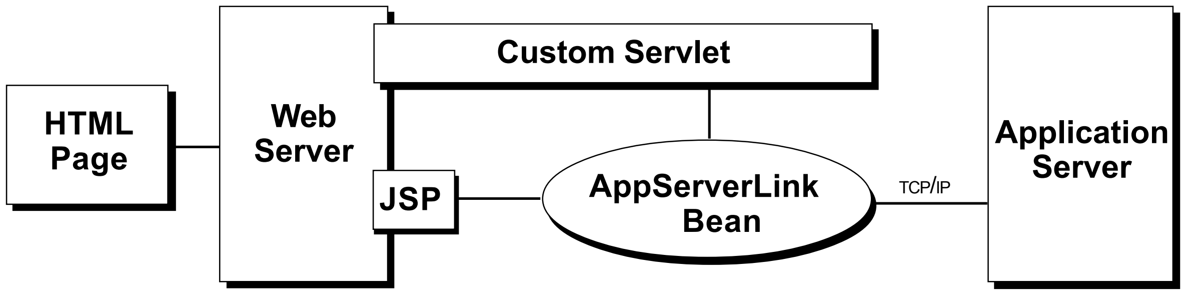
The main purpose of ArcIMS is to serve geographic content over the Internet. As such, it is designed to work with various web servers (e.g., iPlanet™ Web Server and Microsoft® IIS), allowing clients to access and display spatial data in a web browser. A web server communicates to ArcIMS—specifically, the Application Server—through a *connector*. ArcIMS provides several connectors: Servlet connector, ColdFusion connector, and the ActiveX connector. The Servlet connector is the standard connector used for ArcIMS.



While a connector can plug in to a Web server and immediately begin sending ArcXML requests to the Application Server, the AppServerLink is simply the conduit through which your application can send requests and receive responses. You have to build the send and receive mechanisms into your application, utilizing the AppServerLink. For example, you might create a Java application that invokes the functionality in the AppServerLink bean as illustrated in the figure below.



Alternatively, if you're designing an Internet application and your custom business logic resides behind a Web server, you can utilize the AppServerLink by creating a custom servlet or by using JSP. The business logic of your application would generate the necessary geographic content that your application needs through the AppServerLink.



While you can access the AppServerLink bean directly from your application as the figures above illustrate, you'll most likely want to encapsulate the functionality of the AppServerLink in one or more custom beans. By doing so, you can mask the coding complexity of the AppServerLink.

For example, the AppServerLink sends ArcXML requests to the Application Server; using the bean directly requires in-depth knowledge of ArcXML requests and their syntax. Instead, you might create a custom "map" bean that exposes a method to create a map of a particular geographic extent. This method could be invoked by passing in two sets of geographic coordinates that represent the bounding box of the area to be displayed. The returned value from this method could be the URL that points to an image of the desired area. Thus, it's the job of the custom map bean to establish the connection to the Application Server (through the AppServerLink) and construct the appropriate ArcXML request to send to it to generate the map image. Ultimately, it will be much easier for other developers to reuse this bean because they don't have to learn the specific methods of the AppServerLink or ArcXML requests and syntax.

The AppServerLink bean methods

The AppServerLink bean contains a number of methods that you'll use to send information to or hold information returned from the Application Server. Those that you'll use are listed below. For more detailed documentation on these methods, open Index.html, located in the ArcIMS install directory at <ArcIMS Installation Directory>/Documentation/AppServerLink/Classes.

setAppServerHost (String newAppServerHost)—sets the Application Server machine name

setAppServerPort (int newAppServerPort)—sets the Application Server port number through which messages are sent

sendAXLRequest (String serviceName, String AXLRequest, String customService)—sends an ArcXML request to the specified MapService

getResponseAXL ( )—returns the ArcXML response sent from the Application Server

sendHeader (String header, boolean useCustomStream)—sends a specific header request to the Application Server, for example, cmd=ping

getAppServerHost ( )—returns the previously set Application Server machine name

getAppServerPort ( )—returns the previously set port number of the Application Server

getResponseHeader ( )—returns the HTTP header from the Application Server, if available

getServiceName ( )—returns the name of the MapService

Developing Java solutions using the AppServerLink

The AppServerLink consists of several Java classes archived in the file arcims_appserverlink.jar. Among these classes is an AppServerLink class. By creating an instance of the AppServerLink class within a custom Java application, you can use the AppServerLink's public methods to communicate with the Application Server.

The following list of steps will help you to develop a Java solution that uses the AppServerLink. Each step is explained in more detail in the sections below.

1. Add the AppServerLink library to the classpath.
2. Import the arcims_appserverlink and create an AppServerLink object.
3. Specify the Application Server host and port.
4. Send ArcXML requests to the Application Server.
5. Retrieve the ArcXML response from the Application Server.
6. Parse the ArcXML response.
7. Retrieve additional information with AppServerLink class methods.
8. Implement your Java solution.

Note: Code examples given in this section can go in a JavaBean, a Java applet, or a Java application.

Step 1: Add the AppServerLink library to the classpath

In order to use the AppServerLink in any development environment, you first need to add the library into your Java project. How you add this library depends on the development environment being used.

Borland JBuilder (available on Windows® and Solaris™): Choose Project properties from the Project menu. A dialog appears with three tabs under the Paths tab. Click on Required Libraries and choose Add. In the select one or more libraries dialog, choose New to add the AppServerLink as a new library to the IDE. Type in a name for the library such as AppServerLink. Choose the Source tab and click Add. Browse to the location of the arcims_appserverlink.jar file. For example:

On Windows: <ArcIMS Installation Directory>\Connectors\AppServerLink\arcims_appserverlink.jar

On Solaris: $AIMSHOME/Middleware/AppServerLink/arcims_appserverlink.jar

Click OK and click OK again. The arcims_appserverlink.jar file has now been added as a library to your JBuilder project.

Microsoft Visual J++ (available on Windows): From the Project menu, choose <your project name> Properties. In the Properties dialog box, select the Classpath tab and click New. Inside the text box type the entire path to the arcims_appserverlink.jar file. For example:

<ArcIMS Installation Directory>\Connectors\AppServerLink\arcims_appserverlink.jar

Click OK and click OK again. The arcims_appserverlink.jar file is now added as a library to the Visual J++ project.

Sun Java 2 SDK: When compiling Java code with Sun's Java 2 SDK, put the arcims_appserverlink.jar in the classpath. For example:

On Windows,

Compile: javac –classpath <ArcIMS Installation Directory>\Connectors\AppServerLink\arcims_appserverlink.jar myfile.java

Execute: java –cp <ArcIMS Installation Directory>\Connectors\AppServerLink\arcims_appserverlink.jar myfile

On UNIX®,

Compile: javac –classpath $AIMSHOME/Middleware/AppServerLink/arcims_appserverlink.jar myfile.java

Execute: java –cp $AIMSHOME/Middleware/AppServerLink/arcims_appserverlink.jar myfile

Step 2: Import the arcims_appserverlink and create an AppServerLink object

Import the AppServerLink class into a Java class and create an AppServerLink object that uses the methods provided.

Example of importing the AppServerLink class:

```
import com.esri.aims.mtier.beans.AppServerLink;
```

Example of creating an AppServerLink object in your Java file:

```
AppServerLink appClient = new AppServerLink();
```

Step 3: Specify the Application Server host and port

Before any request is sent through the AppServerLink, you must specify the host machine and port for the Application Server. The example below illustrates setting the AppServerLink class's AppServerHost and AppServerPort fields. These values are held in state within the AppServerLink object until the client session ends. Each instantiation of the AppServerLink class from a client creates a new session.

```
import com.esri.aims.mtier.beans.AppServerLink;
public class Map {

  AppServerLink appClient = new AppServerLink();
  public void setHostName(String hostName) {      // Setting the host name.

    appClient.setAppServerHost(hostName);

  }
  public void setHostPort(int port) {        // Setting the server port.

    appClient.setAppServerPort(port);
  }
}
```

Step 4: Send ArcXML requests to the Application Server

After the Application Server host and port number have been set, ArcXML requests can be sent to the Application Server through the AppServerLink object.

The example below demonstrates appending pieces of ArcXML to a StringBuffer. Once all of the ArcXML has been appended, a variable is set to equal the string value of the StringBuffer. The variable (ArcXML string) is then passed through the sendAXLRequest method to the AppServerLink.

```
import com.esri.aims.mtier.beans.AppServerLink;

public class Map {

AppServerLink appClient = new AppServerLink();
public String serviceinfoResponse;
public String serviceInfoAXL;

public void setHostName(String hostName) {              // Setting the host name.

   appClient.setAppServerHost(hostName);

}
public String getHostName() {

   return appClient.getAppServerHost();

}

   public void setHostPort(int port) {           // Setting the server port.

      appClient.setAppServerPort(port);

   }

   public void setServiceInfo(){
      StringBuffer sb = new StringBuffer();
      sb.append("<?xml version=\"1.0\" encoding=\"UTF-8\"?>");
      sb.append("<ARCXML version=\"1.1\">");
      sb.append("<REQUEST>");
      sb.append("<GET_SERVICE_INFO fields=\"false\" envelope=\"false\" renderer=\"false\"
      />");
      sb.append("</REQUEST>");
      sb.append("</ARCXML>");

      serviceInfoAXL = sb.toString();
      sendAXLRequest(this.getServiceName(), serviceInfoAXL);
      sendAXLRequest(this.getHostName(), serviceInfoAXL);
   }

   public void sendAXLRequest(String serviceName, String AXLRequest){
      appClient.sendAXLRequest(serviceName, AXLRequest, null);
   }
}
```

Step 5: Retrieve the ArcXML response from the Application Server

After sending a request to the Application Server, use the getResponseAXL() method to obtain the ArcXML response as a string.

```
public void sendAXLRequest(String serviceName, String AXLRequest){

   appClient.sendAXLRequest(serviceName, AXLRequest, null);
   String AXLResponse = appClient.getResponseAXL();
}
```

Step 6: Parse the ArcXML response

Parsing the response as a string

Once you've received the response, you can parse through the responses content and pull out the information your application needs. Below is an example of how to parse through a string to get the extents of the image, as well as the URL to the image's location. String parsing is done by creating beginning and ending positions of individual parts of the string and then creating a new string based on the characters that fall between the beginning and ending positions.

```
public void getImageAsUrl(String ImageAXLResponse){

   String parseAXLResponse = ImageAXLResponse;
   String eminX;
   String eminY;
   String emaxX;
   String emaxY;

   int ebminX = parseAXLResponse.indexOf("minx") + 6;
   int eeminX = parseAXLResponse.indexOf("miny") - 2;
   eminX = String.valueOf(parseAXLResponse.substring(ebminX, eeminX));
   int ebminY = parseAXLResponse.indexOf("miny") + 6;
   int eeminY = parseAXLResponse.indexOf("maxx") - 2;
   eminY = String.valueOf(parseAXLResponse.substring(ebminY, eeminY));

   int ebmaxX = parseAXLResponse.indexOf("maxx") + 6;

   int eemaxX = parseAXLResponse.indexOf("maxy") - 2;
   emaxX = String.valueOf(parseAXLResponse.substring(ebmaxX, eemaxX));

   int ebmaxY = parseAXLResponse.indexOf("maxy") + 6;

   int eemaxY = parseAXLResponse.indexOf("OUTPUT") - 7;
   emaxY = String.valueOf(parseAXLResponse.substring(ebmaxY, eemaxY));

   int beginMapUrl = parseAXLResponse.indexOf("url") + 5;

   int endMapUrl = parseAXLResponse.indexOf(".jpg", beginMapUrl);

   String mapURL = parseAXLResponse.substring(beginMapUrl, endMapUrl);

}
```

Parsing the response as XML

Parsing the ArcXML response using an XML parser proves to be a faster and cleaner method of parsing. XML parsing requires converting the ArcXML response in bytes to a java.io.InputStream, then creating an XML document from the java.io.InputStream. Once the ArcXML response is an XML document, the document can then be parsed by nodes and attributes. The values of the nodes and attributes can then be set to specific variables.

```
import com.sun.xml.tree.XmlDocument

public void getImage(String AXLResponse){

  java.io.InputStream AXLStream = new
  java.io.ByteArrayInputStream(AXLResponse.getBytes());
  XmlDocument xmlDocument = XmlDocument.createXmlDocument(AXLStream, false);
  Element parent = xmlDocument.getDocumentElement();
  String tagName = parent.getTagName();

   if(tagName.equals("ARCXML")){
        NodeList arcxmllist = parent.getChildNodes();
        NodeList responselist = arcxmllist.item(1).getChildNodes();
        NodeList imglist = responselist.item(1).getChildNodes();

        if(imglist.item(i).getNodeName().equalsIgnoreCase("OUTPUT")){

           NamedNodeMap outputnode = imglist.item(i).getAttributes();

           for(int c = 0; c < outputnode.getLength(); c++){
             if(outputnode.item(c).getNodeName().equalsIgnoreCase("url")){
                mapurl = outputnode.item(c).getNodeValue();
             }
           }
        }
     }
   }
}
```

Step 7: Retrieve additional information with AppServerLink class methods

The AppServerLink class contains five get methods that return information about the state of a bean:

getAppServerHost ( )—returns the previously set Application Server machine name

For example, the following code prints out the name of the host machine where the AppServerLink expects to find the Application Server running.

```
public getServerName(){
   String hostname = appClient.getAppServerHost();
   System.out.println(hostname);
}
```

getAppServerPort ( )—returns the previously set port number of the Application Server

getServiceName ( )—returns the name of the MapService

getResponseAXL ( )—returns the ArcXML response sent from the Application Server

getResponseHeader ( )—returns the HTTP header from the Application Server, if available

For example, the following code is sent in the response for CMD=ping:

```
appClient.setAppServerHost ("electriclake");
appClient.setAppServerPort (5300);

String queryString = "CMD=ping";

out.println ("GET-like test: <br>");
out.println ("Header:\n" + queryString + "<br>");
appClient.sendHeader (queryString, false);
String reply = appClient.getResponseAXL ();
out.println ("Reply:\n" + reply + "<br>);
String header = appClient.getResponseHeader ();
```

Step 8: Implement your Java solution

Once you have developed, compiled, and tested your Java solution (applet, beans, or application), you are ready to implement it in a web site. Make sure that you have ArcIMS up and running. Then, install the AppServerLink and your JavaBean(s) or applet onto the web server machine. In the classpath of your web server or servlet engine, enter the path to the arcims_appserverlink.jar and your JavaBean. Verify that you are calling the correct server name and MapService names for the main map and the overview map. For a JSP implementation, be sure to enable JSP in the web server or servlet engine and place your web site in the web server's document root directory.

Some sample applications

To help you learn about using JavaBeans and JSP to communicate with the AppServerLink, run one of the Java and JSP Viewer samples provided with ArcIMS. The samples implement JavaBeans that use the AppServerLink. These beans are accessed within sample JSP web sites.

Advanced Sample—shows a variety of functions using earthquake data

Geocode Sample—demonstrates geocoding against a streets layer

Extract Sample—demonstrates extracting specified layers

For setup instructions, navigate to

On Windows: <ArcIMS Installation Directory>\Samples\Java_JSP\Sample_Java_and_JSP_Viewers_Setup.html

On UNIX: $AIMSHOME/Samples/Java_JSP/Java_JSP_Samples_setup.html