Agent Interaction SDK Java Developer Guide

Server Applications

12/13/2025

Contents

Server Applications

This chapter introduces principles to write agent server applications developed on top of the Agent Interaction (Java API).
As explained in About the Code Examples, Genesys is developing two sets of examples. This chapter will detail the server code example that demonstrates these voice interactions. It consists of the following sections:

Five Rules to Build an AIL Server Application

Now that you have been introduced to the Agent Interaction (Java API), it is time to outline the rules you will need to observe if you wish to develop a server application.
There are five basic things you will need to do in your server applications:

- **Get the AilFactory singleton.** When your application gets the reference on this factory, your application should test whether it is `null` to get the corresponding exception, as shown in the `getAilFactory()` method of the `Connector` application block.

```
mAilFactory = AilLoader.getAilFactory();
if(mAilFactory == null) {
   ServiceException _es = AilLoader.getInitException();
   throw new RequestFailedException("AilFactory is not initialized " + _es);
}
```

> **Important**
>
> Using the Connector application block ensures that your server application properly handles connection.

- **Manage the AilFactory singleton.** At runtime, your application deals with a unique instance of the `AilFactory`. If you need to restart the AIL library and its connections to Genesys servers, first kill your instance of `AilFactory` by calling the `AilLoader.killFactory()` method, as shown in the `release()` method of the `Connector` application block.

```
mAilLoader.killFactory();
```

If this method call succeeds, you can get a new reference on the `AilFactory` singleton as previously detailed in this section.

> **Important**
>
> Genesys recommends the use of ailLoader.getFactory() in your AIL client application (instead of having a reference to the singleton throughout the code). This decreases the risk of reference issues associated with killFactory usage.

- **Keep references on AIL objects.** If your server application gets a reference on an AIL object, for example a `Place` instance by calling the `AilFactory.getPlace()` method, this instance exists as long as you keep its reference alive. When the reference no longer exists, the object is garbage-collected. In terms of performance, if your application is likely to use this object often, your application should keep a reference to it to avoid having to rebuild the instance. Building AIL objects is time consuming, as it requires collecting data from Genesys servers.
- **Implement multi-threading for event-handling.** As explained in Threading, a `Publisher` thread ensures that AIL events are published sequentially with respect to their time order. So, in listeners' methods, your server application should use multi-threading to process events, and thereby avoid deadlocks.
- **Implement a PlaceListener or an AgentListener**. If your server application listens to a place, you are sure to get all interaction, DN, and media events that occur on the place.

The `Agent Server` code example has been designed to provide you with a very simple server that makes stand out two of these rules, that is, multi-threading implementation and `AilFactory` management through the `Connector` application block.
Now it is time to see how they are implemented in the `Agent Server` example.

Agent Server Example

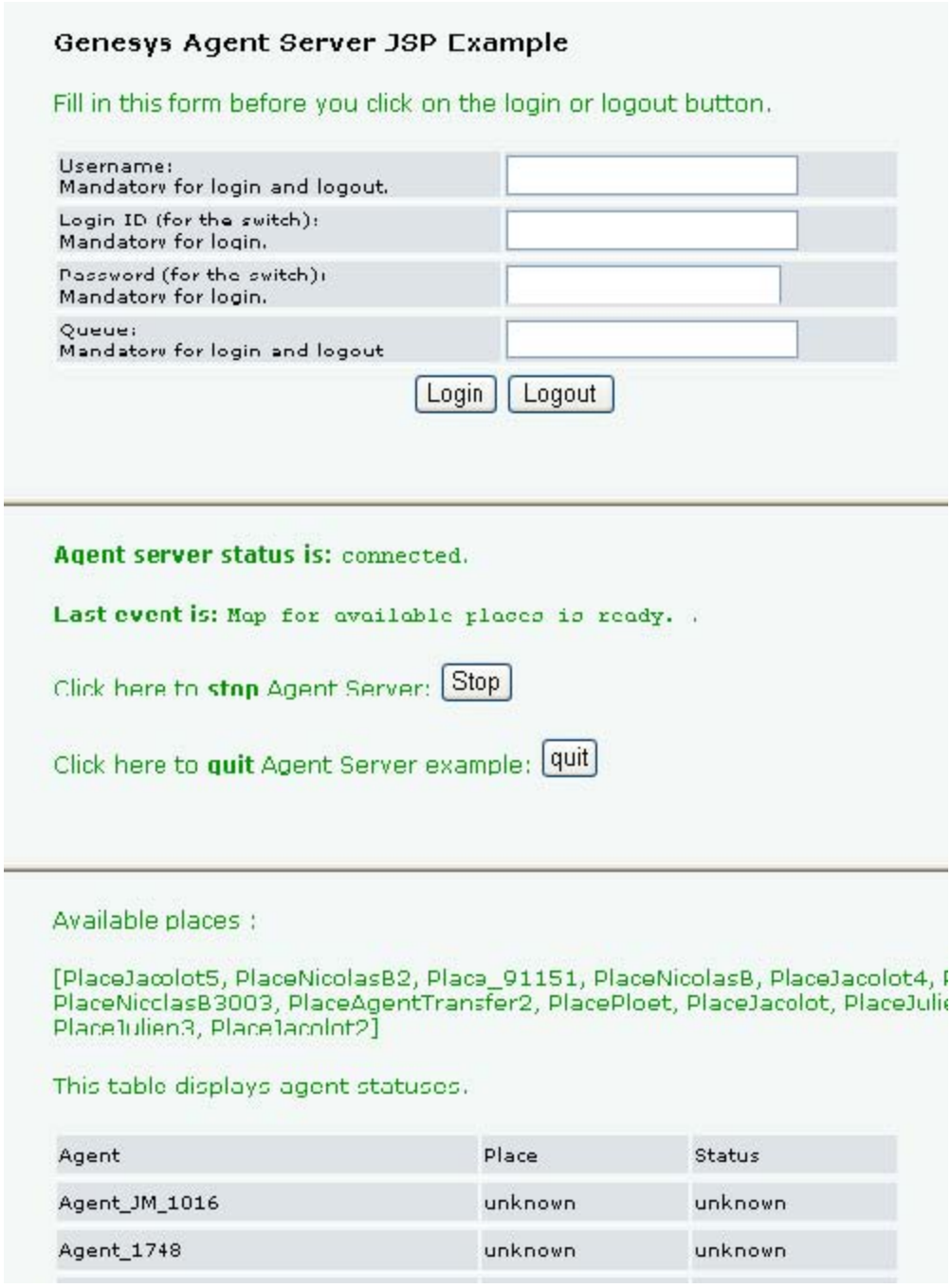
The `Agent Server` code example contains two types of files to be installed on a Tomcat server:

- The java source files, used to build the server application, as shown in the Architectural Overview of the Agent Server Code Example.
- The JSP files, which compose the servlet part of this example. They provide clients with a GUI and manage both client sessions and requests for the server.

When the user loads the `main_frame.jsp` page in a browser, a form appears to launch the server if it is not started, as shown in Agent Server at Startup.

Agent Server at Startup

When the Agent Server is running, it provides the JSP client application with agents' monitoring status. The JSP client application displays these status, registers for monitoring changes, and includes a frame that can send login and logout requests to the Agent Server, as shown in Agent Server is Started.



Agent Server is Started

Because the `AgentServer` example uses classes of the `com.genesyslab.ail.monitor` package to monitor agent status, this example does not deal with `PlaceListener` or `AgentListener` classes.

Connect to AIL

The Agent Server example uses an extended `Connector` application block to perform AIL connection. This extension consists in adding a few lines of code to the inner `ServiceListenerAdapter` class of the `Connector` application block, in order to start monitoring agent status as soon as the statistic service is available.
The following code snippet show the source code added to the `ServiceListenerAdapter.handleServiceStatusChanged()` method.

```
/// Added source code specific to AgentServer example
if(service_type == ServiceStatus.Type.STAT && agentServer != null)
{
   if(service_status == ServiceStatus.Status.ON)
      agentServer.setMonitorListener(true);
   else
      agentServer.setMonitorListener(false);
}
```

> **Important**
>
> The extended `Connector` application block corresponds to the `agentserver.Connector` class of the `Agent Server` code example.

Implement Multi-Threading

The `AgentServer` class implements three inner threads to handle Agent Interaction (Java API) events:

- `GetPlaceInfoThread` to collect place data.
- `StartMonitorThread` to start monitoring agent status.
- `NotifyThread` to handle monitor events.

Collect Place Data

At the server's startup, the `AgentServer()` constructor retrieves two types of data:

- The list of available agents to build a map that client application will display.
- The list of default places for further login actions.

```
//Get agent summaries to build the status map
buildStatusMap();
//Get default places for future login actions
GetPlaceInfoThread p = new GetPlaceInfoThread();
p.start();
```

To get the list of available agents, the `buildStatusMap()` method retrieves agent summaries by calling the `AilFactory.getAgentSummaries()` method. This method returns light objects and is not time-consuming.

```
statusMap= new HashMap();
Iterator itAgents = factory.getAgentSummaries().iterator();
while(itAgents.hasNext())
{
   String id = ((PersonSummary) itAgents.next()).getId();
   statusMap.put(id, new String[]{"unknown","unknown"});
}
```

Because there is no method to get a list of places in the Agent Interaction (Java API), the `AgentServer()` constructor runs a **GetPlaceInfoThread** thread to get an `Agent` instance for each agent of the status map and retrieve the associated default place name, if it exists. This operation is time-consuming because the AIL library has to build numerous `Agent` and `Place` instances, which are not light objects.

```
class GetPlaceInfoThread extends Thread {
   public void run()
   {
      //...
      Iterator itAgents = statusMap.keySet().iterator();

      while(itAgents.hasNext())
      {
         String agentId = (String) itAgents.next();
         Agent myAgent =(Agent) factory.getPerson (agentId);
         Place p = myAgent.getDefaultPlace() ;
         if(p!= null)
            placeVector.add(p.getId());
      }
   }
}
```

Start Monitoring Agent Status

The `ServiceListenerAdapter.handleServiceStatusChanged()` method calls the `AgentServer.setMonitorListener()` method to start monitoring agent statuses if the statistic service is available.
This method creates a `StartMonitorThread` thread which retrieves person summaries and registers an `AgentMonitorListener` for each agent, as shown here:

```
Iterator itAgents = factory.getAgentSummaries().iterator();
monitorListener = new AgentMonitorListener();

while(itAgents.hasNext())
{
   PersonSummary itAgentSummary = (PersonSummary) itAgents.next();
   try
   {
      monitorManager.subscribeStatus(IdObject.ObjectType.PERSON, itAgentSummary.getId(),
monitorManager.getChangesBasedNotification(1), monitorListener);
   }
   catch(RequestFailedException __e)
   {
      System.out.println(__e.getMessage());
   }
}

dataAvailable = true;
notifyListeners("Agent Server is monitoring agent information.");
```

For further details about the `AgentMonitorListener` implementation, see below.

Handle Monitor Event

The AgentServer class implements the MonitorListener interface to handle MonitorEvent events that occur on monitored agents, as shown here:

```
class AgentMonitorListener implements MonitorListener
{
   public void handleMonitorEvent(MonitorEvent event)
   {
      System.out.println("Event: "+event.getClass().toString());
      if(event instanceof MonitorEventAgentStatus)
      {
         MonitorEventAgentStatus agentEvent = (MonitorEventAgentStatus) event;
         NotifyThread th = new NotifyThread(agentEvent);
         th.start();
      }
   }
}
```

When a MonitorEvent occurs, the AgentServer instance creates a NotifyThread which updates the agent status map and notifies all registered clients with this event, as shown in this code snippet:

```
class NotifyThread extends Thread {
   MonitorEventAgentStatus agentEvent;
   public NotifyThread(MonitorEventAgentStatus m_agentEvent)
   {
      agentEvent = m_agentEvent;
   }
   public void run()
   {
      MonitorEventAgentStatus.AgentStatus agentStatus = agentEvent.getStatus();
      HashMap status = getAgentStatusMap(0);
      status.put(agentEvent.getUserName(), new String[]{agentEvent.getPlaceId(),
agentStatus.toString()});
      notifyListeners(agentEvent.toString());
   }
}
```

The AgentServer.notifyListener() method parses the content of the listenerVector vector which contains the listeners that registered to get notified of this event.

```
Iterator itListeners = listenerVector.iterator();
while(itListeners.hasNext())
{
   AgentServerListener listener = (AgentServerListener) itListeners.next();
   listener.handleEvent(msg);
}
```

Submit Login Requests

The AgentServer class does not requires a place name to perform login actions. It uses the list of default places built at startup by the GetPlaceInfoThread thread to perform a login action (see Collect Place Data).

```
String availablePlace = (String) placeVector.get(0);
String log_message = new String( );
```

```
Place m_place = factory.getPlace(availablePlace);
m_agent.login(m_place,loginId, password,
queue, Dn.Workmode.MANUAL_IN,null,null);
placeVector.remove(availablePlace);

notifyListeners(availablePlace+" is no longer available.");
```

Wrapping up

Previous sections list how the Agent Server class implements Agent Interaction (Java API) to manage data and events. Now, let's see how client applications, that is, servlets made of JSP files, interact with the `AgentServer` instance.
The main page of the JSP client application is `main_frame.jsp` . If the server is not started, it loads the `startServerForm.jsp` page which displays a form to be filled in (refer to Agent Server at Startup). When the user clicks on the `Submit` button, the `start.jsp` page creates a new `AgentServer` instance, as shown here:

```
//source from start.jsp
AgentServer agentServer = new AgentServer(configServerHost,configServerPort,
userName,userPassword,applicationName);
application.setAttribute("agentServer",agentServer);
```

Then, the `main_frame.jsp` page can create a client session to connect to the `AgentServer` instance, and loads four JSP pages in frames.

```
//source from main_frame.jsp
<frame src="loginForm.jsp"/>
<frame src="stopServerForm.jsp"/>
<frame src="displayStatus.jsp"/>
<frame src="pullJob.jsp"/>
```

The `loginForm.jsp` page displays a GUI to fill in before submitting a login or logout request. According to the clicked button, it runs the`login.jsp` or `logout.jsp` page, which submits a login or logout request to the server.

```
//source from login.jsp
String msg = agentServer.login(username,loginID,password,queue);
```

The`StopServerFrom.jsp` page displays event information and includes buttons to quit the client application or to stop the Agent Server. When the user clicks the `Stop` button, the `stop.jsp` page is loaded an stops the Agent Server, as shown here:

```
//source from stop.jsp
agentServer.stopAgentServer();
```

The displayStatus.jsp page is in charge of displaying agent monitoring information. Because the agent and place information is not available when the server starts, the servlet tests whether it can retrieve data by calling the`agentServer.isDataAvailable()` method as shown in the following code

snippet:

```
// source from DisplayStatus.jsp
HashMap agentStatusMap = null;
Vector placeVector = null;
if(agentServer.isDataAvailable())
{
   agentStatusMap = agentServer.getAgentStatusMap(index);
   placeVector = agentServer.getPlaceVector();
}
```

To get events (including monitor events), the client application registers an `AgentServerListener` by calling the `AgentServer.addListener()` method.

```
// source from DisplayStatus.jsp
AgentServerListener listener = (AgentServerListener)
session.getAttribute("agentServerListener");
if(listener == null)
{
   listener = new AgentServerListener();
   agentServer.addListener(listener);
   session.setAttribute("agentServerListener", listener);
}
```

The `pulljob.jsp` page is in charge of pulling events every second, as shown here:

```
// source from pulljob.jsp
while(! event)
{
   //...
   AgentServerListener listener = (AgentServerListener)
session.getAttribute("agentServerListener");
   if(listener != null)
   {
      event = listener.gotEvent;
      String msg = listener.getEvent();
      if(msg != null)
      {
         session.setAttribute("event",msg);
      }
   }else
      break;
      Thread.sleep(1000);
   }
}
```

When the `pulljob.jsp` page pulls an event, the application reloads the `main_frame.jsp` page to refresh all frames.