Agent Interaction SDK Java Developer Guide

About the Code Examples

12/12/2025

Contents

About the Code Examples

This chapter introduces the code examples that accompany this developer's guide. It presents essential design considerations and also some initial tasks that an application undertakes in using the Agent Interaction (Java API).

Setup for Development

When you install Agent Interaction (Java API), be sure that you have the required tools, environment-variable values, and configuration data. See the Interaction SDK 7.6 Java Deployment Guide for details.
The Agent Interaction (Java API) product includes all Genesys libraries and third-party libraries needed for proper operation. See this page for downloading the code examples used in this guide.

Agent Interaction (Java API) Installation Directory

The installation directory contains the following subdirectories:

- The `ConfigLayerTemplates\` subdirectory contains the `.apd` files you must use to create proper client application objects in Genesys' Configuration Layer.
- The `doc\` subdirectory contains the Javadoc API Reference.
- The `lib\` subdirectory contains the `.jar` files for the Agent Interaction (Java API).

Source-Code Examples

Discussions in subsequent chapters of this guide refer to the supplied source-code examples. These examples are provided on the download page both in a `.tar.gz` and in a `.zip` archive file.
The code examples illustrate the use of the most common features of the Agent Interaction (Java API). The examples are not tested and are not supported by Genesys.
When you expand the `76sdk_exmpl_ixn_java-agent` archive file containing the code examples, you will find two directories, which divide code examples into two categories: standalone and server. Each directory contains java source files, as well as batch files and shell scripts designed for compiling and running the examples with reference to that directory structure.
The structure of the `StandAloneExamples` directory tree is:

- The top-level directory contains the following files:
  - `README.HTML` provides instructions for compiling and running the examples.
  - `compile.sh` and `compile.bat` are shell scripts (respectively for UNIX and for Windows) that, with a little editing, you can use to compile the examples. They take a single argument, which is the name of the example you want to compile (without the `.java` extension).
  - `go.sh` and `go.bat` are shell scripts (respectively for UNIX and for Windows) that, with a little editing, you can use to run the compiled examples. They take a single argument, which is the name of the compiled class you want to run.

- an agentInteraction.properties file (used by the Common class in Common.java).

- The agent/ directory contains the example source files.
- The classes/ directory is where the scripts store or access compiled classes.
- The doc/ directory contains the Javadoc API reference of the code examples.

The AgentServer directory has a different structure and contains a single example. This directory should be copied to the webapps directory of your Jakarta Tomcat server:

- The top-level directory contains the following files:
    - README.HTML provides instructions for compiling and running these examples.
    - .jsp files, that the Jakarta Tomcat Server runs.
- Both source and class files are stored in the WEB-INF/ directory, including shell scripts that you use to compile the example before you launch the Agent Server example.

Required Third-Party Tools

In order to develop applications with the Agent Interaction (Java API), you will need a compiler, such as the one delivered in the Java Platform, Standard Edition, Development Kit (JDK), version 1.7, from Sun Microsystems.
In this guide, JDK 7 was used to compile and run the code examples.

Important
No JDK prior to 1.7 versions is supported in AIL 7.6.6.

Environment Setup

In order to compile and run, the compiler or the JVM needs access to the libraries of the Agent Interaction Layer. They are located in the lib/ subdirectory of the Agent Interaction (Java API) product installation directory.
Set the following environment variables:

- Specify all Agent Interaction (Java API) .jar files in the CLASSPATH environment variable.
- Specify the location of the Java Runtime Environment in the JAVA_HOME environment variable.

Configuration Data

For the examples provided for this document to work, they need valid configuration data, including connections to servers and configuration objects such as Place, Dn, Agent, and so on.
See the Interaction SDK 7.6 Java Deployment Guide for configuration details. The following items are particularly important:

- The Application Name in the Configuration Layer is used by the AilLoader.

- Your application must use either the `Client` or the `Server` templates.
- Be sure the examples are using correct information for the Configuration Layer host and port, as well as correct information for configuration objects such as agents, places, media, and DNs.

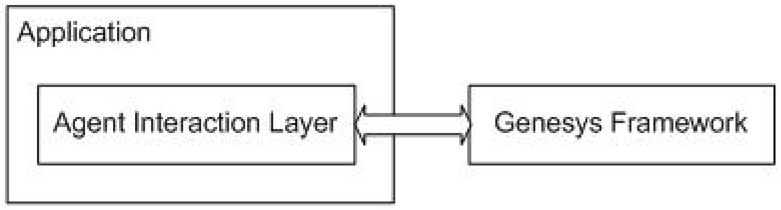
Application Development Design

The AIL library is designed to work across any network that presents TCP/IP access to Genesys servers.
You can create a stand-alone client application or an application service for multiple clients.

- A client application is represented in the Configuration Layer by the client application template.
- A server application is represented in the Configuration Layer by the server application template.

Client Architecture

If the AIL library is used for a client application (for example, for a stand-alone agent desktop application), then as a matter of deployment, the AIL library runs in the same JVM as do the client application code, the GUI, and other related processes. This is illustrated in the following figure.



Client Architecture

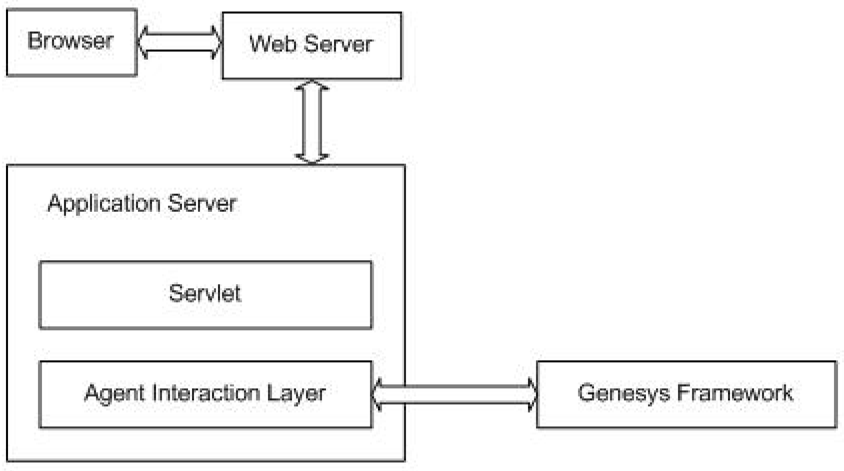
Usually, in this case, the AIL library manages only one agent per instance of the application. The library lives as long as the enclosing application.

Server Architecture

There are two variations that a server application may take:

- The classic server model, in which the application manages network connections, making itself available on a TCP port.
- The web server model, in which the application and AIL library are embedded in a web container, such as Tomcat.

In server applications in which the library is loaded in a web container, a presentation service instantiates the AIL library, which establishes the connections to the Genesys servers. This is shown in Web Container Server Architecture.

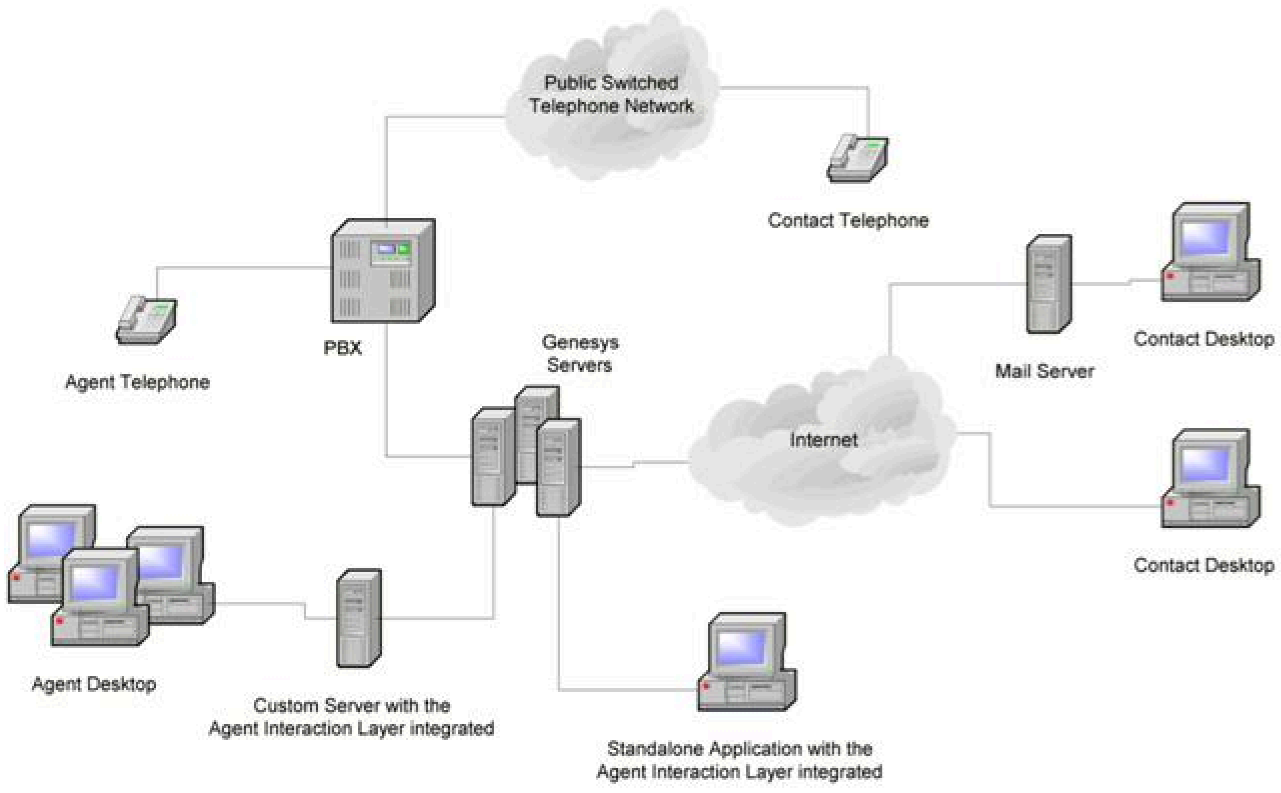


Web Container Server Architecture

Clients access the web server, which requests pages from the web container. A servlet requests data from the AIL library to build a page according to the current states and events.
Either type of server application is likely to handle multiple agents and sessions simultaneously. The AIL library is designed to support such activity. See Server Applications, for details on server development.

Topology

The Agent Interaction (Java API) uses a distributed architecture. It can connect to the Genesys servers, wherever they are, via TCP/IP. This is illustrated in Server Application Topology.
The network topology requirement is that the instance of the Interaction SDK library must be able to connect to Genesys servers.
The Agent Interaction (Java) library core listens to the events coming from such services as the Configuration Layer, the T-Server, the Interaction Server, and so forth. The core manages the status of the different objects, and it filters, preprocesses, and forwards consolidated events to the application using the library.
Event features are similar, whatever the media. Thus, the Agent Interaction (Java) library handles activities of all media in a similar way.


Public Switched Telephone Network
Contact Telephone
PBX
Agent Telephone
Genesys Servers
Mail Server
Contact Desktop
Internet
Contact Desktop
Agent Desktop
Custom Server with the Agent Interaction Layer integrated
Standalone Application with the Agent Interaction Layer integrated

Server Application Topology

Introducing the Standalone Code Examples

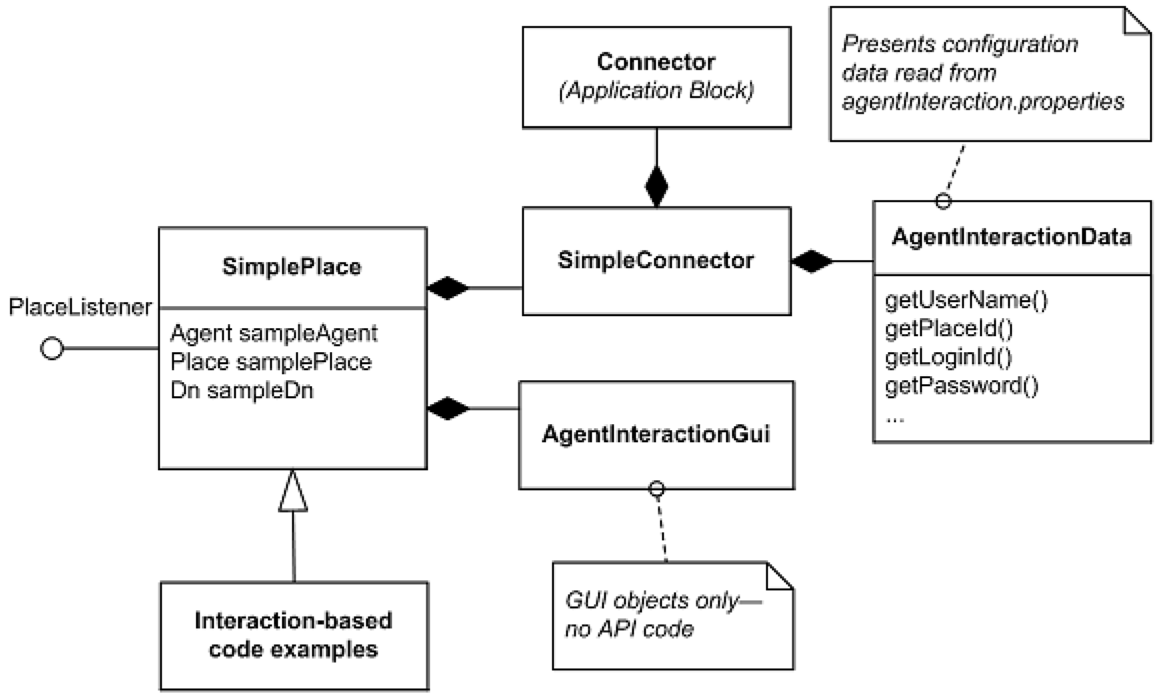
The set of standalone code examples was designed to be interactive. It was also designed to isolate API-related code from presentation code as much as possible. Both of these features should make it easier for you to learn the functionality of the Agent Interaction (Java API).

In order to isolate the API code, separate classes have been set up to read properties information and to create the application’s graphical user interface, as shown below.

As you are learning the API functionality, you can ignore the `AgentInteractionData` and `AgentInteractionGUI` classes.

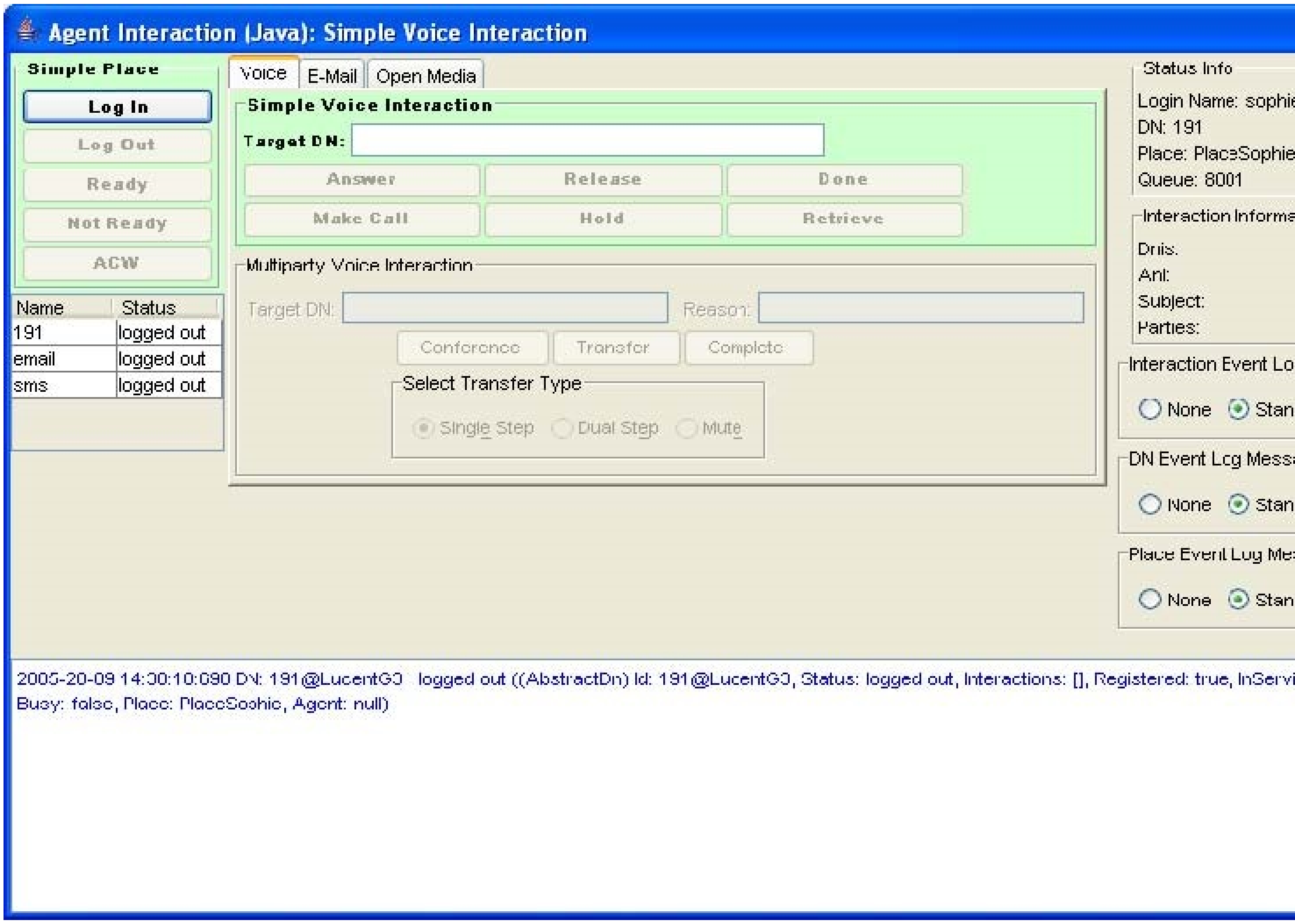
In 7.6, code examples include some of the Agent Interaction Application Blocks for Java, which present best practices for the Agent Interaction (Java API). All examples are built on top of the `Connector` application block that connects to the Agent Interaction Layer.

The examples include the `SimplePlace` class, which is explained in the next chapter. This class implements the `PlaceListener` interface and is the base class for the examples that demonstrate the use of various interaction and event types. `SimplePlace` also calls the GUI class and thereby makes various GUI components, or widgets, available to the examples.



Architectural Overview of the standalone Code Examples

A user interface is available for the standalone code examples. The window title shows the name of the current example, `SimplePlace`. The upper-left corner of the window has a light green background. This shows you which section of the GUI is active for the example you are working on. To the right are four tabs. When you are working on voice, e-mail, open media, or chat examples, the corresponding tab will be selected.



Name	Status
191	logged out
email	logged out
sms	logged out

Login Name: sophie
DN: 191
Place: PlaceSophie
Queue: 8001

Dnis:
Ani:
Subject:
Parties:

Select Transfer Type: [✓ Single Step] Dual Step Mute

Interaction Event Lo: None [✓ Stan]
DN Event Log Mess: None [✓ Stan]
Place Event Log Me: None [✓ Stan]

```
2005-20-09 14:30:10:090 DN: 191@LucentG3  logged out ((AbstractDn) Id: 191@LucentG3, Status: logged out, Interactions: [], Registered: true, InServi
Busy: false, Place: PlaceSophie, Agent: null)
```

User Interface for the Standalone Code Examples

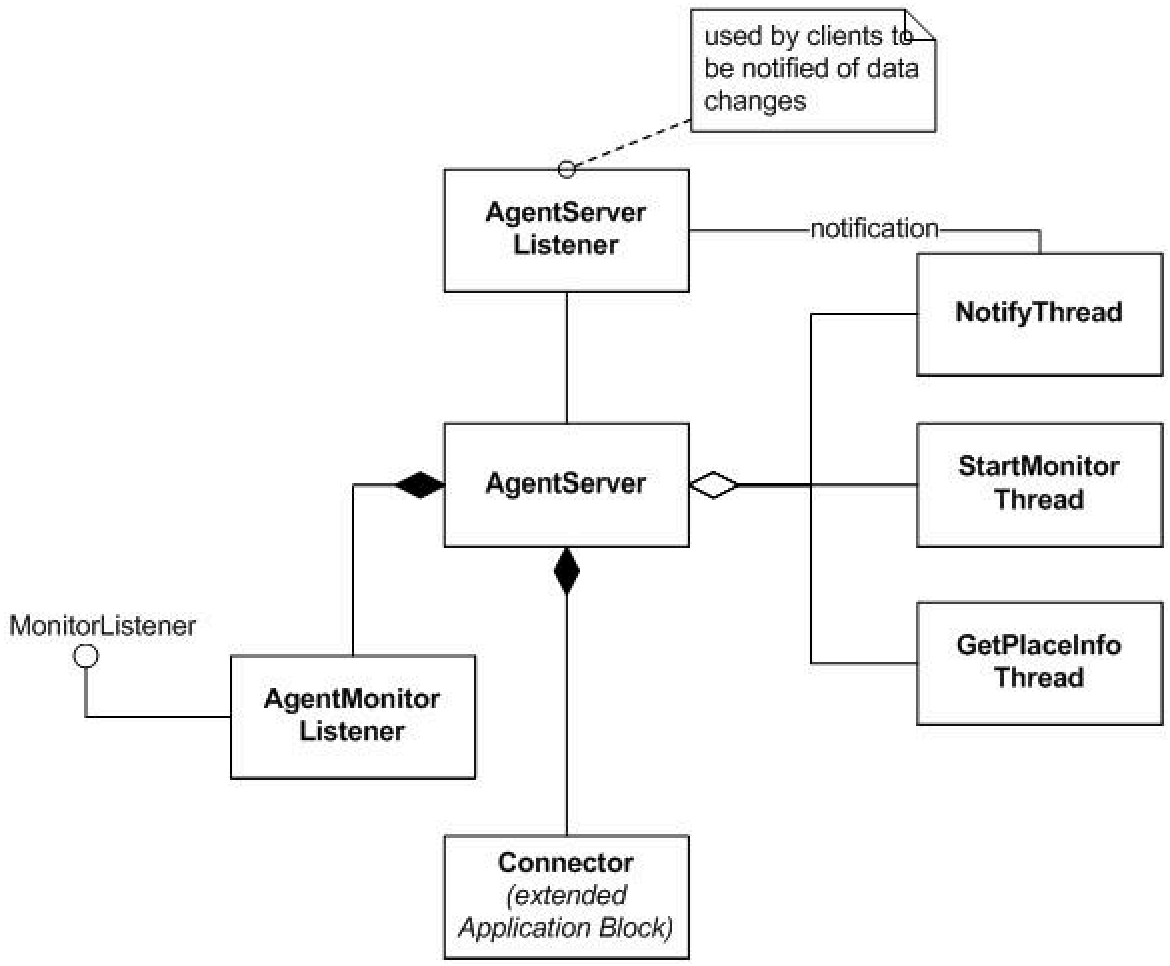
There are two main sections on the right side of the user interface. Status information is provided on the top, and three sets of radio buttons in the middle allow you to control the display of the event messages that appear in the user interface's bottom pane.

The examples generate DN, place, and interaction events. Using the radio buttons, you can display any of them, or none of them. You can also determine how much information you want displayed for each log event.

In order to make the event messages easier to tell apart, they have been assigned their own colors. DN events appear in blue, as shown in User Interface for the Standalone Code Examples. Place events appear in green and interaction events appear in red.

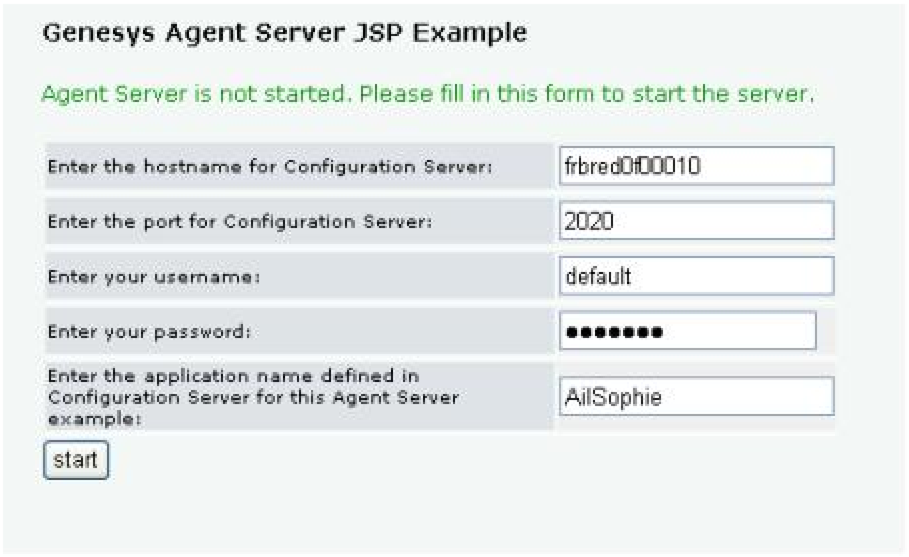
Introducing the Agent Server Code Example

The Agent Server code example was designed to be interactive, and to illustrate a possible implementation of a web server application on top of the Agent Interaction (Java API).
The source code of the Agent Server code example separates API code from the web dynamic code used to display the web graphical user interface, available in a web browser.
The Agent Server java classes include several thread implementation, used to collect or update data, and also to notify the clients with the changes reported by the Agent Server. Also, this code example extends the `Connector` application block, used to connect to the Genesys framework, as shown below.



Architectural Overview of the Agent Server Code Example

The servlet of the Agent Server code example is composed of several `.jsp` pages, that enable the user to send requests to the Agent Server, as shown below.



Genesys Agent Server JSP Example

Agent Server is not started. Please fill in this form to start the server.

Enter the hostname for Configuration Server:	frbred0f00010
Enter the port for Configuration Server:	2020
Enter your username:	default
Enter your password:	●●●●●●●
Enter the application name defined in Configuration Server for this Agent Server example:	AilSophie

Web User Interface for Starting the Agent Server

The above interface shows a `.jsp` page used to start the Agent Server. After the server is started, further `.jsp` pages are associated with a `jsp` session that registers for the being notified by the Agent Server. This makes possible to send agent login and logout requests, and also to pull events to get data changes.

Application Essentials

This section discusses some of the essential API features needed for every application. The discussion refers to the `SimplePlace.java` example in the `StandAloneExamples/agent/interaction/samples/` directory, or to the `Connector` Application Block.
Before running this example, be sure to edit the `agentInteraction.properties` file to specify the correct data in your Configuration Layer (host, port, application name, and so on). In addition to compiling `SimplePlace.java`, you will also need to compile Agent Interaction Java Application Blocks, and the following code example files: `AgentInteractionData.java` , `AgentInteractionGui.java` , and `SimpleConnector.java.`
Every application, whether client or server, must use the `AilLoader` class to get a reference to the `AilFactory` , passing correct configuration data arguments.
T he `Connector` Application Block is in charge of this task. `Connector` is a very simple class that shows how to get an instance of the `AilLoader` and then uses the `AilLoader`.getAilFactory() method to get the `AilFactory` interface.

Use AilLoader

The `AilLoader` is a wrapper to the startup process of the AIL library. Its primary goal is to pass configuration information to the core factory and to return an `AilFactory` interface on the `AilFactory` object in the library core.
Use the `AilLoader` object to make connections to Framework components, to set up logging, and to

get a reference to the `AilFactory` . Be sure that your arguments to the `AilLoader` constructor match the data in the Configuration Layer.

Get Configuration Data

Before you create a new `AilLoader` object, you must set or obtain the following minimum configuration data:

- Configuration Layer host name
- Configuration Layer port
- Backup Configuration Layer host name
- Backup Configuration Layer port
- Application login name to the Configuration Layer
- Application login password
- Interaction SDK application mode: either CLIENT or SERVER
- Connection checking time period
- Server request timeout limit

The following code snippet is from the `Connector` class:

```
// Connect to the Agent Interaction Layer
if(applicationParameters != null) {
   mAppParam = applicationParameters;
   mAilLoader = new AilLoader(
      mAppParam.getPrimaryHost(),
      mAppParam.getPrimaryPort(),
      mAppParam.getBackupHost(),
      mAppParam.getBackupPort(),
      mAppParam.getDefaultUsername(),
      mAppParam.getDefaultPassword(),
      mAppParam.getApplicationName(),
      AilLoader.ApplicationType.getApplicationType( mAppParam.getApplicationType().toInt()),
         mAppParam.getReconnectionPeriod(),
         mAppParam.getTimeout());
}
```

Set Up Logging

As you can see from the Javadoc description in the `com.genesyslab.ail` package, the `AilLoader` class includes methods for setting logging.
To set the logging level to `debug`, use this statement:

```
ailLoader.debug();
```

Likewise, to turn off the console log, issue this statement:

```
ailLoader.noTrace();
```

Similarly, you can issue a method that tells the Agent Interaction Layer not to output log messages to a file:

```
ailLoader.noLogFile();
```

AIL allows you to change the location of your log file. The default log file destination is `./ail.log`. You can specify a different log file directory location and a new filename for the log file, as shown in the `Connector` application block, like this:

```
if(mAilLoader != null) {
    if(file != null) {
        mAilLoader.setDefaultLogFileName(file);
    }
    if(path != null) {
        mAilLoader.setDefaultLogFilePath(path);
    }
}
```

Get a Reference to AilFactory

Use the `AilLoader` class to get a reference to the `AilFactory` interface.

```
// Initialize and return the AIL Factory
ailFactory = AilLoader.getAilFactory();
```

Use AilFactory

If you supply the correct parameters to `AilLoader` , the call to the `AilLoader`.getAilFactory() method triggers the startup process (if the core factory of the Agent Interaction Layer has not yet been instantiated). This is what AIL does:

- Creates the instance of the core Agent Interaction Layer factory.
- Creates and initializes connections to the Genesys Server.
- Initializes the caches.
- Returns an `AilFactory` interface to the core factory.

The `AilFactory` , when instantiated, requests an application name from the Configuration Layer. When the `getAilFactory()` method returns, the Agent Interaction Layer is initialized and ready. (If `NULL`, an error occurs. See `initexception.`)
Through the `AilFactory` interface, you can now instantiate almost all of the objects your application needs. When each object is created, the core assigns it a unique `String` identifier. Retrieving an object’s identifier is important if your application works with more than one instance of an object type (for instance, multiple DNs on a place).

Get Application Information

The `AilFactory.getApplicationInfo()` method returns an `ApplicationInfo` object that has methods for retrieving most of the data that the Configuration Layer has about your application. See the Javadoc description for the `ApplicationInfo` class in the `com.genesyslab.ail` package.

```
ApplicationInfo appInfo = ailFactory.getApplicationInfo();
```

The following code snippet shows some of the information available from `ApplicationInfo.`

```
int appID = appInfo.mApplicationDBID;
String appName = appInfo.mApplicationName;
String appVer = appInfo.mApplicationVersion;
Map appOpts = appInfo.mOptions;
```

Use Agent

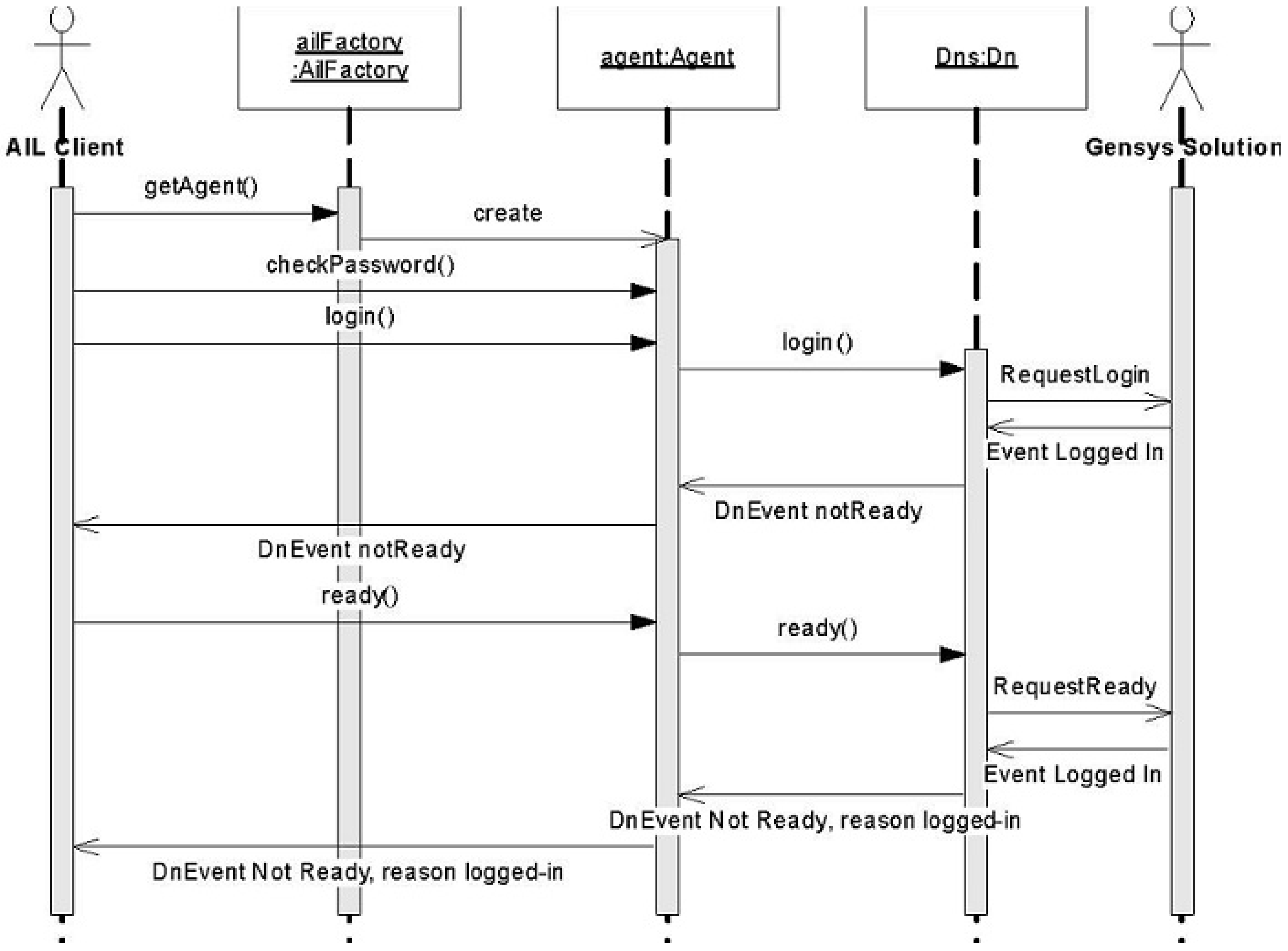
Once you have your `AilFactory` object, you can log in an agent to start working. An agent is a person sitting at a place. A place contains a set of DNs and media that the agent uses to work. The `Dn` object identifies an access point in a switch that can handle a phone call. The `Media` object identifies a media type that handles multimedia interactions.
The `Agent` and `Place` interfaces have two distinct sets of methods, one for agent activity occurring on the non-voice media of a place and one for agent activity occurring on the DNs of the place.
For example, to have your application associate an agent with a place's media, you would use the `loginMultimedia` method. This method enables you to specify the media types to link to the agent's session.
If, on the other hand, you want to allow the agent to use the place's DNs, your application can either:

- Use the `login` method, which attempts to associate the agent with every one of a place's DNs.
- Retrieve the place's DNs with the `Place.getDns()` method, and attempt to link the agent to a restricted set of DNs.

Important

Although agents can log into only a single place, they can also be linked to DNs or media types that are not associated with that place.

The event flow for an agent login on the DNs of a place is illustrated below.



Configuration and Logging In

This sequence diagram shows the `DnEvent` events received due to two consecutive agent actions, `login` and `ready`, performed on DNs. Those events propagate the status changes of the DNs as a result of the agents' actions.
Similarly, when performing agent actions on media, your application receives `PlaceEventMediaStatusChanged` events propagating media status changes.
The following code snippet gets an `Agent` object interface from the `AilFactory.getPerson()` method. It passes in a String that names an agent person in the Configuration Layer, and casts the returned person as an `Agent`. The `Agent`.checkPassword() method takes the agent's password as a String and returns `true` if the password is correct.

```
Place place = ailFactory.getPlace(place_name);
Agent agent = (Agent) ailFactory.getPerson(agentname);
if (agent.checkPassword(agentpassword)) {
   agent.login( place, loginId, agentpassword,
      queuename, Dn.Workmode.MANUAL_IN, null); }
```

The `Agent.login()` method activates the `login()` methods on the place and its DNs. The arguments to the `Agent.login()` method are:

- `place`—Place interface for the Agent's place.
- `loginId, agentpassword`, and `queuename`—String names for valid Configuration Layer objects.
- `Dn.Workmode.MANUAL_IN`—Constant value that sets the workmode for the default DN on the agent's place.
- The final argument, in this case `null`, is a `Map` object that stores reasons for the login event on the DN.

After the `Agent.login()` method successfully executes, the agent is logged in on all the DNs of its default place. The agent must change its status to `ready`. After the call to the `Agent.ready()` method, the agent is ready to receive voice events and interactions. This same `ready()` method is available on the `Agent`, `Dn`, and `Place` interfaces.

Receive Events

To receive events on an object requires a class that implements the appropriate listener interface and implements all the methods for the listener interface. The class must register as a listener on that object. Each event sends an appropriate event object to a well-known listener event-handling method in the class. The code for the handler inspects the inbound event object and takes appropriate action. This follows the Observer Design Pattern, similar to the listeners in the JDK.
Take, for example, the job of tracking event flow on an agent. The API features involved include the `Agent` interface (in the `com.genesyslab.ail` package) along with the `AgentEvent` interface and the `AgentListener` interface (in the `com.genesyslab.ail.event` package).

Agent Interface

To work with a particular agent, get an `Agent` interface from the `AilFactory` for the agent:

```
Agent mAgent;
mAgent = ailFactory.getPerson(strAgentName);
```

The `Agent` interface has many methods, but this discussion concerns its `addAgentListener()` method. The following call registers this class as a listener for events on `mAgent`

```
mAgent.addAgentListener(this);
```

AgentListener Interface

Create a class that implements the `AgentListener` interface and implements all of the `AgentListener` methods. There are several methods on the `AgentListener` interface: implement methods according to your application requirements. For instance, if your application needs to be updated with interaction status, you should implement the `handleInteractionEvent()` method.
For every interaction event on the agent, this handler method receives an `InteractionEvent` object, which stores current status and other updated information.
The other `AgentListener` methods can be empty if you are not interested in inspecting their inbound

event objects.

InteractionEvent Interface

The `InteractionEvent` interface supports a variety of methods; its `getStatus()` method returns the status of the interaction object at the time the event occurred:

```
Interaction.Status getStatus()
```

The handler code should pass the current state and other information (such as the `InteractionId` for the interaction) to another thread that can take appropriate actions. Design your handlers to return as quickly as possible, because the library core works with all handlers sequentially, waiting for each handler to return before working with the next handler.

Get Real Time Information

The `com.genesyslab.ail.monitor.Monitor` interface provides monitoring features for agent status. You can subscribe to an agent, and get real-time information about that agent's status which is available in the `AgentCurrentState` category from the Stat Server.
To get a `Monitor` instance, call the `AilFactory.getMonitor()` method, as shown here:

```
Monitor mMonitor = ailFactory.getMonitor();
```

Then, to monitor status changes, implement a `MonitorListener` class that receives `MonitorEvent` events, as shown in the following code snippet:

```
public class SimpleMonitorExample implements MonitorListener
{
   public SimpleMonitorExample (Monitor exampleMonitor, String object_id )
   {
      //Adding the listener
      exampleMonitor.subscribeStatus(ObjectType.PERSON, object_id,
Notification.CHANGES_BASED, this); }

      public void handleMonitorEvent(MonitorEvent event)
      {
         //Implementation of the listener method
         //...
      }
}
```

The Agent Server code example deals with the `com.genesyslab.ail.monitor` package. For further details, see Agent Server.