Agent Interaction SDK Services Developer Guide

Genesys Interaction SDK 7.6.6

12/29/2021

Table of Contents

Agent Interaction SDK Services Developer Guide

Welcome to the *Agent Interaction SDK Services Developer's Guide*. This document introduces you to the concepts, terminology, and procedures relevant to the Agent Interaction Service Proxy Libraries.

This guide presents an overview of the architecture and communication protocols, the setup procedures for client development, and the product's API capabilities via Genesys Interface Server (GIS).

> **Warning**
> This document is valid only for the **7.6.6** release(s) of this product.

The Agent Interaction Service Proxy Library for .NET provides you with access to the Agent Interaction Layer (AIL) Java API through the Genesys Interface Server (GIS).

Intended Audience

This document, primarily intended for developers who are familiar with Simple Object Access Protocol (SOAP), Hypertext Transfer Protocol (HTTP), XML (Extensible Markup Language) technologies, assumes that you have a basic understanding of:

- Computer-telephony integration (CTI) concepts, processes, terminology, and applications.
- Network design and operation.
- Your own network configurations.

You should also be familiar with these tools:

- XML Schema
- SOAP (Simple Object Access Protocol)
- WSDL (Web Services Description Language)

Depending on the technology choice for client development, working knowledge of Java or of some other Web Services client-side programming language may be required.
Developers should be familiar with the Genesys Framework and with the AIL library.

Chapter Summaries

In addition to this preface, this document contains the following chapters:

- About Agent Interaction SDK Services. This chapter provides an overview of the Agent Interaction SDK Services architecture.
- About the Examples. This chapter introduces techniques for developing your application based on code snippets in the documentation.
- Data Transfer Object. This chapter discusses the use of Data Transfer Objects (DTO).
- The Event Service. This chapter discusses the management of events.
- The Agent Service. This chapter discusses agent features and management.
- Place, DNs, and Media. This chapter covers how media are working in a place with the services, how to monitor them and how to deal with the specific voice media, that is DNs.
- The Interaction Service. This chapter presents the management for characteristics common to all interactions, no matter their media.
- Voice Interactions. This chapter discusses voice interactions' management.
- E-Mail Interactions. This chapter discusses e-mail and collaboration features.
- Chat Interactions. This chapter discusses chat interactions' management.
- The Contact Service. This chapter discusses the management of contacts.
- The Callback Service. This chapter covers the management of callback records.
- The SRL Service. This chapter presents how to provide agents with standard responses to help them process interactions.
- The Outbound Service. This chapter covers the management of outbound campaigns.
- Expert Contact. This chapter covers the management of expert features.
- Additional Services. This chapter covers the management of additional services.
- Best Coding Practices. This chapter reviews the rules you would otherwise find throughout this book for developing a high-performance application on top of the Agent Interaction Services.
- The Agent Status Example. This chapter covers the implementation of the agent status example.

Table of Content

Change History

This section lists all the changes between the 7.6.5 and 7.6.6 versions of this document. This section lists all the changes between the 7.6.5 and 7.6.6 versions of this document.

Version 7.6.610.00

Page name	State	Additional details

About Agent Interaction SDK Services	Updated	• Updated sections
		  • Overview
		  • Components
		  • Development Platform
		  • Production Runtime Platform

About the Examples	Updated	• New sections
		  • XML Optional Attributes
		  • Optional Attributes
		  • XML Optional Attributes
		  • Optional Attributes
		• Updated sections
		  • Service Factory
		  • Access Services
		  • XML Configuration File for .NET
		  • XML Configuration File Example
		  • HTTP Redirections
		  • XML Configuration File for Java
		  • Services Integrated in an Agent Application
		• Deleted sections
		  • XML Optional GSAP

Page name	State	Additional details
		Attributes • Optional GSAP Attributes • XML Optional SOAP Attributes • Optional SOAP Attributes • GSAP
AdditionalServices	Updated	• Updated sections • History Information

About Agent Interaction SDK Services

The Agent Interaction Services Libraries product provides developers with the services of the Agent Interaction SDK, as well as developer essentials (such as documentation and code examples) to assist you in creating an agent application capable of handling interactions of all media types.
To run any agent applications developed with this SDK, you need to install the Genesys Integration Server (GIS), which exposes the Agent Interaction Services.

Overview

To develop successful client applications with the Agent Interaction SDK, you can:

- Use the microsoft .NET Framework SDK version 2.0, 3.5, 4.0 & 4.5 to create a C# based application.
- Use the Apache AXIS toolkit version 1.4 to create stubs for a Java-based application.
- Your applications might have some of the following purposes:
- A contact center agent desktop application to let agents interact with Genesys software and handle interactions processed by the contact center.
- An application that integrates third-party software with Genesys software.
- Other applications specific to your needs.

Components

The Agent Interaction Services Libraries product includes the following components.

- The Agent Interaction Service Proxy Library for .NET—provides the .NET proxy that includes the SOAP protocol to communicate with GIS.
- The Agent Interaction Service Proxy Library for Java—provides the Java proxy that includes the SOAP protocol to communicate with GIS.
- The *Agent Interaction SDK 7.6 Services API Reference for the .NET Proxy* in CHM format, covering the Agent Interaction Services .NET API.
- The *Agent Interaction SDK 7.6 Services API Reference for the Java Proxy* in HTML format, covering the Agent Interaction Services Java API.
- This online *Developer's Guide*.

This set of components supports an application that lets you manage agent and interaction features, as well as services such as voice, outbound campaigns, and callback.
The Agent Interaction Service API is designed to allow development of applications that have specific requirements for the custom manipulation of particular service features. The communication protocol used to interact with GIS depends on the library that you use for your development.

Platform Requirements

The platform requirements for developing your application are a little different from those needed to use your application.

Development Platform

For .NET development:

- Microsoft .NET Framework SDK, version 2.0, 3.5, 4.0, and 4.5, available at http://msdn.microsoft.com/netframework/
- Microsoft Visual Studio

For Java development:

- Apache AXIS toolkit, version 1.4, available at http://xml.apache.org/axis/index.html
- Java Development Kit (JDK), version 1.7.

Production Runtime Platform

For .NET development:

- Microsoft .NET Framework, version 2.0, 3.5, 4.0, and 4.5, available at http://msdn.microsoft.com/netframework/

For Java development:

- Apache AXIS toolkit, version 1.4, available at http://xml.apache.org/axis/index.html
- Java Runtime Environment (JRE), version 1.7.

Scope of Use

Typical usage scenarios include:

- Managing agent activity:
  - Implement login and logout functionality.
  - Implement ready, not-ready, and after-call-work features.
- Handling voice interactions (depending on your switch's available features):
  - Make an outgoing call.
  - Answer an incoming call.
  - Hold and retrieve a call.
  - Transfer a call.

    - Alternate calls.
    - Initiate, enter, and leave a conference.
- Handling the callback feature:
    - Accept, reject, or cancel a request.
    - Accept and dial a callback request.
    - Reschedule the record.
- Handling e-mail:
    - Create and send an e-mail.
    - Answer an e-mail.
    - Transfer an e-mail.
    - Pull an e-mail from a workbin.
- Handling outbound campaigns:
    - Add a new record to the campaign.
    - Request a record.
    - Cancel a record.
    - Reject a record.
- Using the Standard Response Library:
    - Get standard responses.
    - Get standard response information about categories.
    - Get standard responses by category.
    - Manage favorite standard responses.
- Managing contacts:
    - Add a contact.
    - Remove a contact.
    - Modify contact information.
    - Search a contact.
- Using contact history information.
- Using the workbin features to store interactions:
    - Get the workbin’s queues and views.
    - Put an interaction in the Workbin.
- Using the system features to get options about the application used in the Configuration Layer.

Architecture

On the Genesys Interface Server side, the exposed services deal with the Genesys Framework and perform the client-side services' requests.

Service-Oriented Architecture

The Service-Oriented Architecture (SOA) is a specific type of distributed system in which features are exposed through services. When you use the Agent Interaction Service Libraries, you are dealing with service interfaces that do not manage anything locally. Each service defines a specific feature of your distributed system. Data management and actions are performed by GIS, and you are concerned only with the interface descriptions.

Multithreaded

The Agent Interaction Service Libraries are thread-safe and therefore your application can run in multithreaded environments. In particular, parallel threads can make calls to the same services' methods at the same time without encountering issues.

Synchronization

Your application establish a link with GIS, exposing the service which performs your client-application requests. The communication with GIS is synchronous.

Connectivity

Connections to Genesys servers are maintained by GIS. Your client-side application can be notified of servers' statuses, namely any loss of a connection.
GIS can maintain connections to multiple T-Servers.
GIS is designed to work in a single-tenant environment. It is possible to create a multi-tenant application, but all configuration layer objects that your application uses must be specified in the `Tenants` tab of the application, and these names must be unique.
For further information, refer to the*Genesys Integration Server 7.6 Deployment Guide* .

Framework Compatibility

GIS connects to the following Genesys servers in the Genesys Framework:

- Configuration Server—Configuration Server (and the Configuration Layer generally) stores configuration information such as application parameters, or objects description such as DNs, places, or persons. The library core monitors the configuration server to update modifications. The library provides full integration with Genesys Configuration Layer objects such as Agent, Place, and DN.
- T-Server—Genesys T-Servers handle telephony requests and events by communicating with switches.

> For voice-only mode, your application should connect with a Configuration Server, at least one T-Server, and, optionally, a Contact Server (included with Multimedia).

Supported switches (and their corresponding T-Servers) include: Nortel Meridian 1, Nortel Symposium, Alcatel 4400, Lucent G 3, Siemens Hicom, Genesys IP Media Exchange, Aspect, DMS 100, MD 110, and NEC Apex.

Genesys Multimedia Compatibility

GIS connects to the Genesys Multimedia components, such as Interaction Server and Universal Contact Server, and provides full multimedia support for e-mail, chat, and open media interactions. Connectivity to handle these interactions involves the following Multimedia components:

- Interaction Server—This server manages e-mail, chat, and open media interaction information along with the Genesys Framework.
- Chat Server—This server manages chat interactions between agents and web visitors.
- Universal Contact Server (UCS)—This database server is used to retrieve e-mails, history, and contact information. It also allows manipulation of the contact history and the standard response library. This server is optional for an application designed to run in voice-only configuration.

For e-mail handling, GIS should connect with a Configuration Layer, a UCS, and an Interaction Server (the last two included with Multimedia).
For chat handling, GIS should connect with a Configuration Layer, a Chat Server, a UCS, and an Interaction Server (the last three being included with Multimedia).
For open media interaction handling, GIS should connect with a Configuration Layer, a UCS, and an Interaction Server (the last two included with Multimedia).

Outbound Campaign Support

GIS also connects to the Genesys Outbound Solution:

- Outbound Campaign Server—This server controls and organizes outbound campaigns.

For Outbound Campaign handling, GIS should connect with a Configuration Layer, an Outbound Campaign Server, and at least one T-Server.

Voice Callback Support

GIS connects to the Genesys Universal Callback Solution:

- Callback Server—This server controls and organizes callback records.

For Voice Callback handling, GIS should connect with a Configuration Layer, a Callback Server, and at least one T-Server.

About Proxies and Examples

This release of the documentation includes code snippets in most chapters. These code snippets illustrate many product features and can serve as a basis for developing your own applications.

> **Important**
> To download the code samples, see this page.

The code snippets in this *Developer's Guide* are in C#, but there are differences across these examples depending on the generated proxy and the language. For instance, in the *Agent Interaction SDK Services API Reference for the .NET Proxy,* C# service interfaces are defined in accordance with the following rule: `I<service_name>Service`. In the generic *Agent Interaction SDK 7.6 Services API Reference for the Java Proxy,* on the other hand, service interfaces are defined in accordance with the following rule: `<service_name>Service`.

> **Important**
> To download the API references, see this page.

You can do any of the following:

- Use a toolkit to generate a proxy from the provided WSDL files.
- Use one of the provided .NET proxies available on the product CD in the `tools/` directory.
- Use one of the provided Java proxies available on the product CD in the `tools/` directory.

Then, use the chosen proxy to build your desktop application using the Agent Interaction SDK Service that the Agent Interaction Services Libraries expose.

Generating a SOAP Proxy

You can use a toolkit to generate a SOAP proxy from the provided WSDL files—for example, Apache AXIS toolkit, version 1.1 or 1.3, for Java development (for further information, see: `http://ws.apache.org/axis/java/user-guide.html`).
With a SOAP proxy, use the GIS session service to connect your client application and to set options. Refer to the *Statistics SDK 7.6 Web Services Developer's Guide* for further details about the session service, and see this chapter's API Overview for further details about available options.

Opening a Session

The first step your Agent Interaction SDK Services client application must perform is to open a session in GIS to get a session ID, which must be passed in the URL of all SOAP requests. As your application creates services, for each service, specify the ENDPOINT_ADDRESS_PROPERTY and the session ID as shown in the following code snippet.

```
/// creation of an agent service using a stub created with /// Apache Axis toolkit 1.1

import com.genesyslab.www.ail.*;
import com.genesyslab.www.ail.agent.*;

//Creating a gis session - GIS server location set when
//generating the stub
SessionServiceServiceSoapBindingStub sessionService =
(SessionServiceServiceSoapBindingStub) new
SessionServiceServiceLocator().getSessionServiceService();

// Time out after a minute
sessionService.setTimeout(60000);
Identity id = new Identity();
id.setPrincipal("example");
id.setCredentials("");
sessionId = sessionService.login(id);

System.out.println("sessionId= " + sessionId);
sessionService._setProperty( sessionService. ENDPOINT_ADDRESS_PROPERTY,
sessionService._getProperty( sessionService.ENDPOINT_ADDRESS_PROPERTY)
+ "?GISsessionId=" + sessionId);

// Accessing Services
String[] value = sessionService.getServices(new java.lang.String[] {
"GIS_INTERACTIONSERVICE"});

AgentServiceSoapBindingStub agentService =
   (AgentServiceSoapBindingStub) new AgentService_ServiceLocator().getAgentService();

agentService.setTimeout(60000);

/// Property used to pass session id in requests
agentService._setProperty(
agentService.ENDPOINT_ADDRESS_PROPERTY,
agentService._getProperty( agentService.ENDPOINT_ADDRESS_PROPERTY) +"?GISsessionId=" +
sessionId);

// then using agent service is similar to C#
// logging an agent

LoginVoiceForm loginVoiceForm = new LoginVoiceForm();
loginVoiceForm.setLoginId(loginId);
loginVoiceForm.setWorkmode(WorkmodeType.AFTERCALLWORK);
MediaInfoError[] values = agentService.login(agentId, placeId, loginVoiceForm, null);
```

Using the .NET Proxy

You can use the provided .NET proxy to minimize session management tasks and to simplify service creation. This proxy is available in the tools/ directory on the GIS Product CD.

This section presents how to connect to GIS, and how to use XML and options for instantiating this connection.

Service Factory

The `com.genesyslab.ail.ServiceFactory` class is the entry point for the .NET proxy. You must create a `ServiceFactory` object in order to connect. The connection can be synchronous or asynchronous, according to the method called:

- `ServiceFactory.createServiceFactory()`—At creation, the factory instance tries to connect synchronously to GIS. If the connection fails, it raises an exception.
- `ServiceFactory.asyncCreateServiceFactory()`—After the factory creation, the factory instance tries to connect asynchronously to GIS until a connection succeeds or the factory is released. To monitor the connection status, you must specify an `IServiceFactoryListener` listener at factory creation.

When you create the factory (synchronously, or asynchronously), you must specify parameters to configure your connection:

- You can fill a `Hashtable` and pass it at `ServiceFactory` creation. See XML Configuration File for .NET for details about options.
- You can use an XML file to configure your `ServiceFactory` object.

Using an XML file is simple: write your own XML file that defines the factory parameters, or use the default `ail-configuration.xml file; then, indicate the factory parameters to be used. The following code snippet shows the ServiceFactory creation based on the WebServicesFactory factory defined in the ail-configuration.xml file.`

```
// Instantiation of a ServiceFactory to make the connection
ServiceFactory myServiceFactory =
   ServiceFactory.createServiceFactory("WebServicesFactory", null, null);
```

See XML Configuration File for .NET for further details.

> **Important**
>
> An `ail-configuration.xml` file is available on the GIS product CD in the `tools/` directory.

Access Services

To access the available services, you create them by calling the `createService()` method of your instantiated factory, as shown in the following code snippet.

```
IXxxService iservice =
myServiceFactory.createService(typeof(IXxxService), null) as IXxxService;
```

If the `Hashtable` parameter is `null` in the `createService()` call, the Agent Interaction Service layer takes into account the current context of the factory. Otherwise, the Agent Interaction Service layer

uses the specified `Hashtable` for service creation.
The following code snippet shows how to create an agent service:

```
IAgentService myAgentService = myServiceFactory.createService(typeof(IAgentService), null) as
IAgentService;
```

> **Important**
> Your application can typically use the current factory context for service creation.

XML Configuration File for .NET

In your XML configuration file, or in the default `ail-configuration.xml` file, you must specify for the `factory` tag one of the following two attributes with their `url` option, according to the protocol used to communicate with GIS:

- For SOAP:
  - `WebServicesFactory` —The factory name.
  - `url` option—The value is `http://[Server Address]:[Server Port]/gis`.

The following sections present the optional attributes, based on proxy type, attached to the mandatory attributes specified above.

XML Optional Attributes

The following table shows all the attributes that you can define.

Optional Attributes

Name	Type	Description
`UseCookieContainer`	bool	Specifies whether or not the use of cookie containers is alloed. By default, it is set to `false`. You must set it to `true` to manage `http` sessions. This is mandatory for enabling high availability.
`BackupUrls`	string	A list of backup URLs to be used in case of disconnection, separated by commas as shown in this example: `"[http://[host1][:port1]/gis,http://[host2][:port2]`
`Timeout`	int	The timeout interval for an XML web

Name	Type	Description
		service client that waits for a synchronous XML web service request, to complete, in milliseconds. The default value is `100000` milliseconds.
`NbRetriesOnFailure`	string	The maximum number of reconnection attempts when calling a service method. The default value is `0` .
`RetryPeriodOnFailure`	string	The period in milliseconds between two reconnection attempts.
`ThreadPool.MaxWorkerThreads`	int	Indicates the maximum number of worker threads allowed at runtime. You must increase this number if your application makes multiple calls to service method, especially if the calls concern the `IEventService.getEvents` method.
gis.asynchronousConnectionInterval	int	Specifies the time period in seconds (`30` seconds by default) between two connection attempts. This option is used in case your application connects asynchonously.
`gis.checkSessionInterval`	int	The check session interval, in seconds. A value of `0` means no check.
`gis.username`	string	The GIS user name to log in the factory. Refer to Configuration Layer documentation for details.
`gis.password`	string	The GIS password to log in the factory. Refer to Configuration Layer documentation for details.
`gis.tenant`	string	The GIS tenant to use with the factory. Refer to Configuration Layer documentation for details.
`gis.sessionId`	string	The GIS session identity to use with the factory. If you use this option, do not use `gis.username` , `gis.password` , and `gis.tenant` .
`notification.HTTPport`	int	The notification HTTP port. The default value is `0`, in which case the remote system chooses an open port on your behalf.

Name	Type	Description
notification.createHTTPchannel	bool	Specifies whether to create an HTTP channel. The default value is true.
notification.objectURI	string	Specifies the remote object Universal Resource Identifier (URI). By default, the URI is generated by the WebServiceFactory.
notification.reachableURL	string	The reachable URI from the server.
service-point-manager.defaultConnectionLimit	int	The service point manager's connection limit. The default value is 2.
service-point-manager.maxServicePointIdleTime	int	The service point manager's maximum idle time. The default value is 900,000 milliseconds (15 minutes).

XML Configuration File Example

The following is an example of an XML configuration file for a SOAP connection:

```
<?xml version="1.0"?>
<configuration default-factory="WebServicesFactory" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance">

<factory name="WebServicesFactory" classname="com.genesyslab.ail.WebServicesFactory"
assembly="AilLibrary">
<option name="Url" value="http://[Server Address]:[Server Port]/gis" />
<option name="gis.username" value="default" />
<option name="gis.password" value="password" />
< !—- OPTIONAL
<option name="gis.sessionId" value="1234567"/>
<option name="notification.HTTPport" type="int" value="10000"/>
<option name="notification.createHTTPchannel" type="bool" value="true"/>
<option name="notification.objectURI" value="NotifLoad"/>
<option name="gis.checkSessionInterval" type="int" value="900"/>
<option name="service-point-manager.defaultConnectionLimit" type="int" value="10"/>
<option name="notification.reachableURL" value="http://localhost:8080/ail"/>
<option name="service-point-manager.maxServicePointIdleTime" type="int" value="90000"/>
END OPTIONAL — >
</factory>
</configuration>
```

HTTP Redirections

By default, redirections are disabled at startup. To enable redirections, you must set the AllowAutoRedirect property to true as follows:

```
WebServicesFactory wsf = mAilServiceFactory.ServiceFactoryImpl as WebServicesFactory;
```

```
wsf.gisSessionService.AllowAutoRedirect = true;
```

This option is dynamic and can be modified at runtime or during the compilation.
The following table lists the supported HTTP codes in this configuration.

Redirection Codes tested with the .NET Proxy

HTTP Code	Supported	Description
300	Yes	POST, then GET to the new URL.
301	Yes	POST, then GET to the new URL.
302	Yes	POST, then GET to the new URL.
303	Yes	POST, then GET to the new URL.
304	Yes	No redirect.
307	Yes	POST, then POST to the new location.
308	No	No redirect.

Using the Java Proxy

You can use the provided Java proxy to minimize session management tasks and to simplify service creation. This proxy is built from the Apache Axis toolkit, version 1.3, and is available in the `tools/` directory on the GIS Product CD.
This section presents how to connect to GIS, and how to use XML and options for instantiating this connection.

Service Factory

The `com.genesyslab.soa.client.ServiceFactory` class is the entry point of the proxy. You must create a `ServiceFactory` object in order to connect. The connection can be synchonous or asynchronous, according to the method called.
Except for the default configuration file name, the process and the method to be called are identical to those described in Service Factory.

> **Important**
>
> The default XML configuration filename is `proxy-configuration.xml`. For further details, see XML Configuration File for Java.

Access Services

To access the available services, you create them by calling the `createService()` method of your instantiated factory, as detailed in Access Services.

XML Configuration File for Java

In your XML configuration file, or in the default `proxy-configuration.xml` file, you must specify for the `factory` tag one of the following two attributes with their `url` option, according to the protocol used to communicate with GIS:

- SOAP
  - `AilWebServicesFactory`—The factory name.
  - `Url` option—The value is `http://[Server Address]:[Server Port]/gis`.

Your application reads the XML configuration file—by default, `proxy-configuration.xml` —to determine which protocols and options should be used for instanciating its connection to GIS. The following sections present the optional attributes, according to proxy type, attached to the mandatory attributes specified above.

XML Optional Attributes

The following table shows all the attributes that you can define.

Optional Attributes

Name	Description
`Username`	Username pour basic authentification.
`Password`	Password pour basic authentification.
`backupUrls`	List of backup connection urls to be used in case of disconnection.

Name	Description
`MaintainSession`	Indicates whether or not the HTTP session must be maintained. By default, it is set to `false`.
`DocumentMode`	Indicates the document mode, `false` for `rpc/encoding`, otherwise `true` for `document/literal`. The default value is `false`.
`NbRetriesOnFailure`	The maximum number of reconnection attempts when calling a service method. The default value is `0`.
`RetryPeriodOnFailure`	The period in milliseconds between two reconnection attempts.
`Connection.Timeout`	The timeout interval for an XML web service client that waits for a synchronous XML web service request to complete, in milliseconds. The default value is `100000` milliseconds.
gis.asynchronousConnectionInterval	Specifies the time period, in seconds (`30` seconds by default), between two connection attempts. This option is used if your application connects asynchonously.
`gis.checkSessionInterval`	The check session interval, in seconds. A value of `0` means no check.
`gis.username`	The GIS user name to log in the factory. Refer to Configuration Layer documentation for details.
`gis.password`	The GIS password to log in the factory. Refer to Configuration Layer documentation for details.
`gis.tenant`	The GIS tenant to use with the factory. Refer to Configuration Layer documentation for details.
`gis.sessionId`	The GIS session identity to use with the factory. If you use this option, do not use `gis.username`, `gis.password`, and `gis.tenant`.
`notification.HTTPport`	The notification HTTP port. The default value is `0`, in which case the remote system chooses an open port on your behalf.
`notification.reachableURL`	The reachable URI from the server. http://[client host]:[client port]
`http.proxyHost`	The name for the proxy host.
`http.proxyPort`	The port of the proxy host.

Name	Description
`http.proxyUser`	The username for the proxy host.
`http.proxyPassword`	The password for the proxy host.

XML Configuration File Example for the Java Proxy

The following code snippet presents a `proxy-configuration.xml` file to be used with the Agent Interaction Services Proxy Library for Java:

```
<?xml version="1.0" ?>
<configuration default-factory="AilWebServicesFactory" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance" >
<factory name="AilWebServicesFactory"
classname="com.genesyslab.ail.ws.client.AilWebServicesFactory" >
<option name="Url" value="http://localhost:8080/soa"/>
<option name="gis.username" value="default"/>
<option name="gis.password" value="password"/>
<option name="gis.tenant" value=""/>
<option name="MaintainSession" value="false"/>
<option name="DocumentMode" value="false"/>
<option name="Username" value=""/> // username pour basic authentification
<option name="Password" value=""/> // password for basic authentification
<option name="http.proxyHost" value=""/> // proxy host
<option name="http.proxyPort" value=""/> // proxy port
<option name="http.proxyUser" value=""/> // proxy user
<option name="http.proxyPassword" value=""/> // proxy password
<option name="ConnectionTimeout" value="60"/> // timeout request response in s
<option name="gis.asynchronousConnectionInterval" value="30"/>
<option name="gis.checkSessionInterval" value="900"/>
<option name="gis.sessionId" value="1234567"/>
<option name="notification.HTTPport" value="0"/>
<option name="notification.reachableURL" value="http://[client host]:[client port]"/>
</factory>
</configuration>
```

GIS License

SOAP

The `GIS_INTERACTIONSERVICE` license is checked out when your application needs to call the `ServiceFactory.createFactory()` method. To check the license in, your application calls the `ServiceFactory.releaseFactory()` method.
In a scenario where the agent logs out and the application properly terminates, your application should release the ServiceFactory instance when the application ends; this frees the agent's license. In an application crash scenario, the license is checked in when the GIS session ends, as is also the case with the `GIS_CONFIGURATION_SERVICE` and `GIS_STATSERVICE` licenses.

HTTP Redirections

Unlike with the .NET proxy, you cannot enable or disable redirections at compilation or runtime. In Java, the Axis client handles the HTTP connection through a library, which is defined in the `wsdd` client file, and relies on the v3.0.1 Jakarta HTTP client (see HttpClient Home.)
For instance, to enable redirections, you should disable the `HttpCommonsSender` library in the Java client that implements the proxy library:

```
context.setProperty("EnableHttpCommonsSender", "false");
```

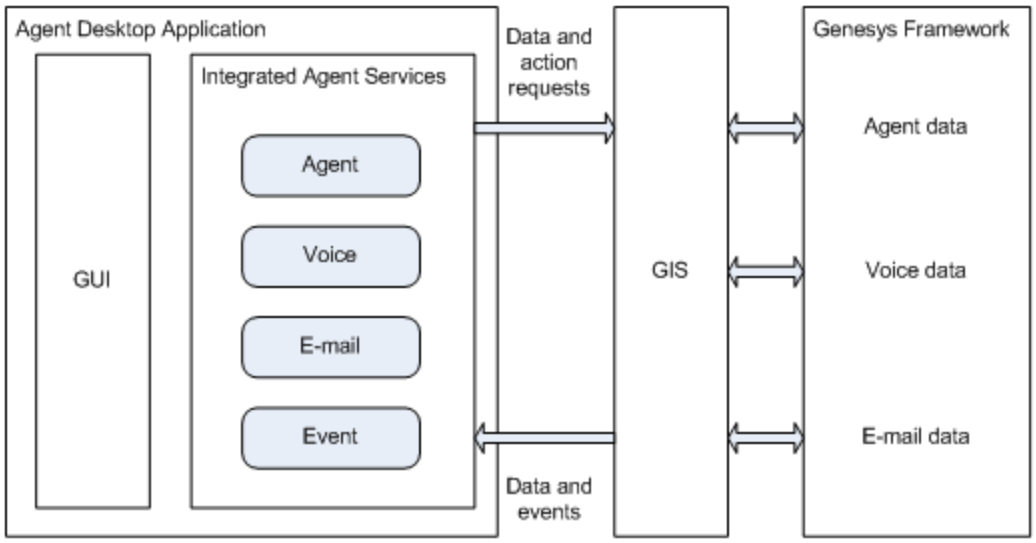
Then, you can provide your own `client-config.wsdd` file which defines the transport layer, as shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment name="defaultClientConfig"
xmlns="http://xml.apache.org/axis/wsdd/"
xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
<globalConfiguration>
<parameter name="disablePrettyXML" value="true"/>
<parameter name="enableNamespacePrefixOptimization" value="false"/>
<parameter name="sendMultiRefs" value="false"/>
</globalConfiguration>
<transport name="http" pivot="java:com.genesyslab.soa.impl.channel.axis.handler.HTTPSender" />
<transport name="https" pivot="java:org.apache.axis.transport.http.CommonsHTTPSender"/>
<transport name="local" pivot="java:org.apache.axis.transport.local.LocalSender"/>
</deployment>
```

To develop a transport layer, you should extend the `org.apache.axis.handlers.BasicHandler` class. Refer to the official Axis documentation for further information.

API Overview

The Agent Interaction Services API works with the mirroring services available in GIS. Typically, your application is a client application, integrating Agent Interaction Services to perform agent actions and to communicate data and events with the Genesys Framework, as shown in the figure below.



Services Integrated in an Agent Application

Your application deals with services that transparently hide GIS and the Genesys Framework, which handles information and CTI objects. These services provide the following features:

- handling agent activity, such as login and logout.
- handling voice interactions such as answering, calling, and callback.
- handling e-mail interactions.
- handling outbound campaigns.
- using the Standard Response Library.
- handling contacts and their histories.

One particularity of the Agent Interaction Services API is its *service approach design* This means that a service deals with dedicated information concerning a set of remote objects. For example, the agent service deals with agent data only.

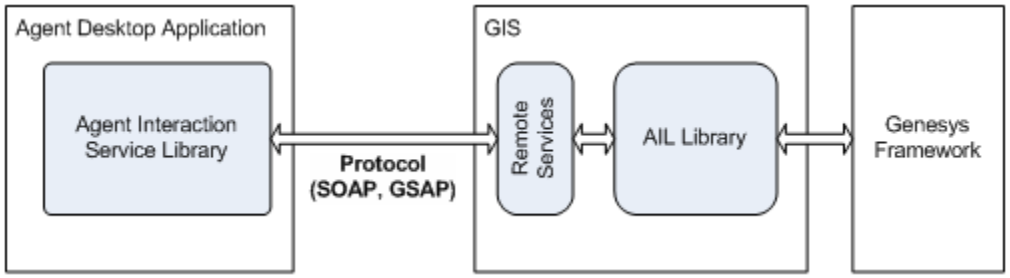
Building an Application Using Services

Take the following steps (a general strategy) to build your application on the Agent Interaction Services API:
Connect your application. See Opening a Session.

1. Get an event service. The Event Service.
2. Get an agent service. The Agent Service.
3. Subscribe to events.
4. Get the services required to implement your application features.
5. Update your application any time you retrieve an event according to possible actions or status changes.

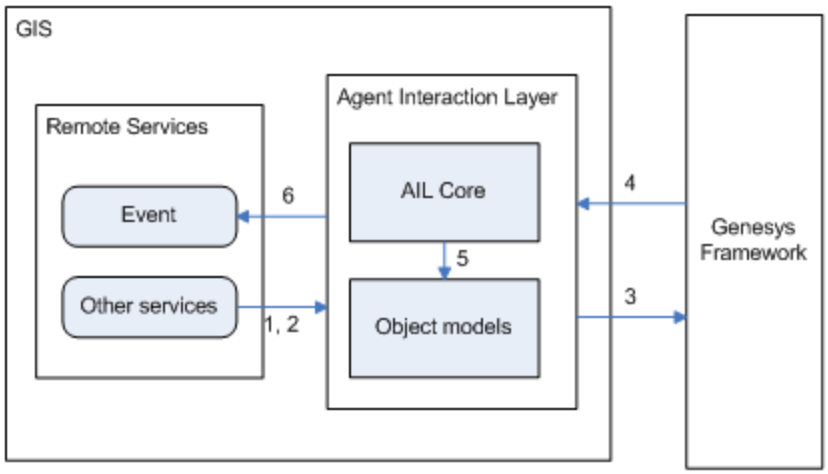
The Remote Services

When using the Agent Interaction Services API, you are dealing with remote services integrating the Agent Interaction Layer library. This library is part of the Genesys Agent Interaction (Java API). See the Agent Interaction SDK Java documentation for full details of AIL features.
The AIL library internally implements models of Genesys products, such as Framework voice calls, e-mail, outbound campaigns, and so on.
The remote services are exposed in GIS, as shown in the following figure.



The Remote Services and the Genesys Interface Server

On the GIS side, the remote services implement the AIL library, which internally handles models of the Genesys Framework. On the agent-desktop-application side, the integrated services hide the interactions with the remote services which encapsulate the AIL library.
The AIL library maintains models of the Genesys objects used by your application. These objects might include, for example, DNs, a `Places`, agents, or interactions. The native AIL library API offers interfaces to perform actions on these objects, and the library core internally handles the state models, as illustrated in the following figure.



The Remote Services and the AIL Internal Core

The numbered labels in the above figure describe the following actions:

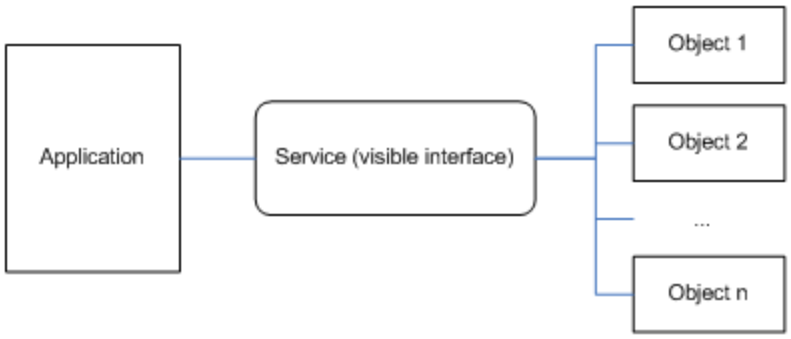
1. Implementation of an AIL Interface dealing with a core object
2. Calling an AIL object method
3. AIL Sending a set of requests
4. AIL Receiving Events
5. AIL core updating core object models
6. AIL sending events to the Event Service

As shown in the above figure, the AIL core notifies the remote services with events when the states of objects or data change. For example, if your service requests an agent login on a media type, once the agent is logged in, the AIL core notifies the event service that the agent status on the media is now `READY` or `NOT_READY`. If your application has registered to listen to events on the agent, it receives the event and can inspect the data.

> **Important**
>
> Genesys recommends that you base your agent desktop application on the state models provided by the Agent Interaction Services API.

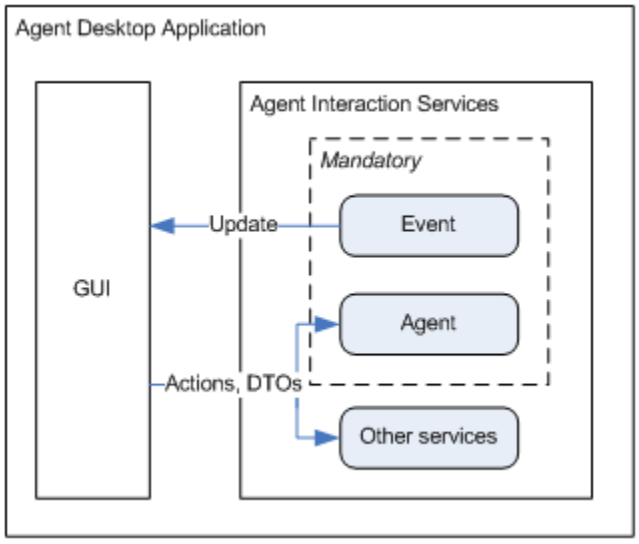
Using the Services

The Agent Interaction Services API provides a set of services. A service is an interface dealing with a group of objects. For example, an interaction service gives access to interactions' data. A single request can apply to a group of several interactions.



Service Concept

Services are agent-oriented; that is, services are designed to fulfill requests of agent type and should be used for this purpose.
The Agent Interaction Services API represents services suitable to perform agent actions and to retrieve any data required by an agent-oriented application. How you make use of the various services is a matter of your application design.
Your application integrates at least two services: the agent service and the event service, as shown in the following figure.



Using the Services in Your Agent Desktop Application

To properly use other services, your application requires:

- An agent service—This service manages agent actions and data; without an agent logged in with this service, some services cannot fulfill actions and data requests.
- An event service—Your application must update according to the data propagated in events which are

published in this service.

Possible Actions

The Agent Interaction Service API is designed to implement agent desktop applications. Therefore, services have been designed to perform agent actions that affect the components managed by the Genesys Framework.
For example, getting ready, logging in, and logging out on media are actions handled by the `IAgentService` interface, and performed by calls to the corresponding methods.
However, a given service's actions might not all be available at a given point in time. An obvious example is an agent not being allowed to become `READY` on media where he or she has not yet logged in.
Thus, to perform a particular action, your application must first check to see if this action is possible:

- Actions are identified in the `<service_name>Action` enumeration.
- Possible actions are provided with attributes of the `I<service_name>Service` interface attributes.
- Possible actions are updated and propagated with events.

A good use of possible actions is enabling or disabling graphical components that the agent uses to perform actions.

Statuses

The services interact with objects and provide you with their statuses during runtime.

- Statuses are identified in the `*Status` enumeration of service namespaces.
- Status access is provided with attributes of the `I<service_name>Service` interface.
- Status changes are propagated in events.

You should base your application design on these statuses. For example, in a GUI context, if a call status is `IDLE` (terminated), the associated GUI components have no need to be visible anymore.

> **Important**
> To determine which actions are available in the current status of the application, rely on the provided possible actions.

Data Transfer Object

This chapter discusses the use of Data Transfer Objects (DTO).

Introduction

The purpose of this section is to introduce the general DTO concepts.
In a client application, a transaction may require multiple server requests to complete. This going back and forth takes up significant amount of time to complete the transaction.
To improve the performance of a set of requests, the solution is to package all the required data into a Data Transfer Object (DTO) that can be sent with a single call.
A DTO is a generic container for a key-value list of data associated with several distinct remote objects. You specify in the list only the keys you are interested in.
You use this list to retrieve or modify attributes' values according to their properties.

DTOs in the Service API

The Agent Interaction Service API makes use of the DTO Pattern for services attributes that can be retrieved, set or published. DTOs are involved in published events, in services methods calls, and so on.
DTOs carry key-value attributes of several services. They are handled with dedicated methods and classes as presented in the following sections.

Dedicated Classes

Each service namespace includes classes gathering attributes for DTO handling. You can find several types of DTO classes:

- `XxxDTO` —This class manages the DTO related to `Xxx`; for example, `InteractionDTO` is a class managing the DTO of an interaction.
- `XxxListDTO` —This class manages an array of `XxxYyyDTO`; for example `AgentListDTO` manages an array of `PersonDTO`.
- `XxxSummaryDTO` —This class manages the DTO of an `Xxx` summary; for example `AgentSummaryDTO` is a class managing the DTO of an agent summary.

The attributes list of a DTO is a `KeyValue` array. The `KeyValue` class is a very simple container having two fields:

- `KeyValue.key` —The attribute name
- `KeyValue.value` —The object corresponding to the attribute value.

Attributes

Each service handles a set of objects and proposes several domains defining accessible attributes to deal with objects data.

> **Important**
>
> To determine what are the available attributes of a service, see the service interface description in the*Agent Interaction SDK 7.6 Services API Reference*.

For example, the `IAgentService` interface is a service dealing with a set of agents. Agents have two defined domains:

- `person` defines common data associated with persons (even if the person is not an agent) as for example: `person:lastname` and `person:firstname`.
- `agent` defines specific agent data as for example `agent`:defaultPlaceId (default place for the agent) or `agent:currentPlaceId` (current place for the agent).

Notation

For each service, the domain attributes used in DTO are defined in accordance with the following rule:

```
domain[[.subdomain]...]:attributeName
```

For example, the following attributes exist:

- `interaction:interactionId`
- `interaction.voice:phonenumber`
- `interaction.mail.out:invitations`

> **Warning**
>
> When you use an attribute, you have to use the complete attribute name, including the domain and sub-domains.

Properties

An attribute can have the properties defined, as shown in the following figure.

Attribute Properties

Attribute	Properties
`read`	The `IXxxService` attribute is readable and can be retrieved with a `IXxxService.getXxxDTO()` method.
`read-default`	The `IXxxService` attribute is likely to be often read, so it is part of the default attributes.
`write`	The `IXxxService` attribute is writable using a `IXxxService.setXxxDTO()` method.
`event`	The attribute can be published via the event service.
`event-default`	The attribute is likely to be often published via the event service, so it is part of the default attributes.

> **Important**
>
> To determine an attribute's properties, see the attribute description in the service interface definition in the *Agent Interaction SDK 7.6 Services API Reference*.

DTOs Handling

Agent desktop applications use DTOs to read and write service attributes. DTOs also play an essential role in event handling.

> **Important**
>
> Genesys recommends that you avoid the use of the * wildcard in DTOs since they cause longer processing times for transactions.

This section shows how to read and set DTO attributes values and introduces the use of DTOs in events.

Reading DTOs

Reading a DTO consists in reading a list of attributes identified with a `read` property in the service domain. You first define the list of attribute names, then use the appropriate `get*DTO()` method. For example, if you want to read the person:firstname and person:lastname attributes of the `IAgentService` interface, first define an array of these attribute names:

```
string[] myAttributeNames = new string[]{“person:firstname”, “person:lastname” };
```

Retrieve the corresponding values using the `IAgentService.getPersonsDTO()` method as illustrated in the following code snippet:

```
/// Defining the list of agents you are interested in:
string[] myAgentIds = new string[]{ “agent0”, “agent1”};

/// Retrieving for each agent the attributes value /// defined in mAttributeNames
PersonDTO[] myValues = myAgentService.getPersonsDTO(myAgentIds,myAttributeNames);
```

Access to the attribute values in the data field of the `AgentDTO` object.

```
/// Displaying agent0 attributes name and value:
foreach(KeyValue data in myValues[0].data )
{
   System.Console.WriteLine("{0}={1}", data.key, data.value.ToString());
}
```

Setting DTOs

You can set new values for service attributes having the `write` property in the service domain. Create a `*DTO` containing the `KeyValue` pairs of the writable service attributes and their new values. Once you have created the DTO, you call the appropriate `I**Service.set*DTO()` method.
For example, the `IAgentService` interface allows you set an array of `PersonDTO`. Each `PersonDTO` object associates a list of `KeyValue` attributes with an agent ID. You can create a DTO containing the writable `agent:signature` attribute and set it with the `IAgentService.setPersonsDTO()` method, as illustrated in the following code snippet:

```
/// Creating a DTO array
AgentDTO[] aNewDTO = new PersonDTO[1];

/// Creating a new DTO
aNewDTO[0] = new PersonDTO();

/// This DTO is related to agent0 information
aNewDTO[0].personId = “agent0”;
aNewDTO[0].data = new KeyValue[1];

/// The targeted data is the agent0 signature
aNewDTO[0].data[0] = new KeyValue();
aNewDTO[0].data[0].key = "agent:signature";
aNewDTO[0].data[0].value = "This is agent0 signature";

/// Setting the new DTO
PersonError[] errors = myAgentService.setPersonsDTO(aNewDTO);
/// Displaying the errors
foreach(PersonError err in errors)
{
   System.Console.WriteLine("Setting DTO: {0} -", err.ToString());
```

```
}
```

The attributes are updated in the order of the `KeyValue` array. If an error occurs on an attribute, the method returns an error in an array specifying on which attribute the error has occurred.

> **Important**
>
> If an error occurs on the update of an attribute, it does not stop the update process: the remaining attributes are updated.

DTOs and Events

Domain attributes that have the `event` property or the `event-default` property are published in events.
To determine which attributes are published by an event, refer to the *Agent Interaction SDK 7.6 Services API Reference*. The available attributes are listed in the event description (part of the service interface definition).
When you subscribe to events, you specify which attributes are retrieved. Then, the incoming events contain the `KeyValue` array with these attributes and their current value.
For further information about event handling, refer to The Event Service:

- Building TopicsEvent for further details on DTO when subscribing to events.
- Reading DTOs in Events for further details on published DTOs.

DTOs and Wildcards

Agent Interaction SDK 7.6 Services API includes wildcards which simplify the code for getting attributes through DTOs.
Attribute Wildcard lists the possible wildcards to access groups of attributes. However, Genesys recommends that your application makes a limited use of these wildcards.
Retrieving large groups of attribute values causes longer processing times for transactions, and decreases your application's performances. In particular, Genesys recommends that you avoid using the * wildcard.
The `default` wildcard should be your preferred one, because attributes marked as `default` are commonly used by applications deployed above the *Agent Interaction SDK 7.6 Services API*.

Attribute Wildcard

Wildcard	Meaning
*	All the attributes of all the domains and subdomains.
`default`	All the attributes marked default in all domains and subdomains.

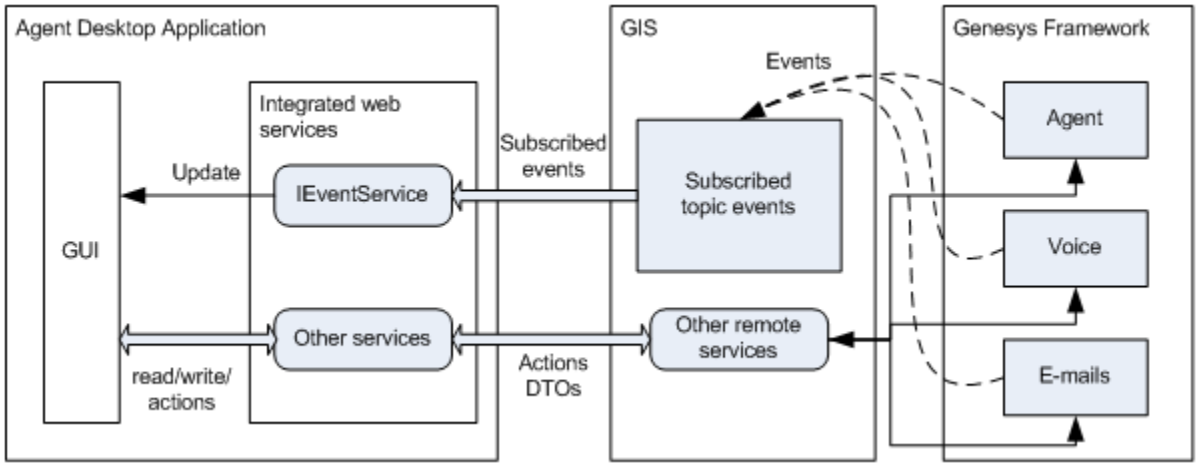
Wildcard	Meaning
`domain:*`	All the attributes of this domain.
`domain:default`	All the attributes marked as `default` in this domain.
`domain.*:*`	All the attributes of the subdomains.
`domain.*:default`	All the attributes marked as `default` in the subdomains.
`domain.subdomain:*`	All the attributes of this `domain.subdomain`.
`domain.subdomain:default`	All the attributes marked as `default` in this `domain.subdomain`.

The Event Service

The event service is the `IEventService` interface defined in the `com.genesyslab.ail.ws._event` namespace. To manage events, your application must integrate this interface and use classes of its namespace to deal with it.

Event Service Overview

Event handling is achieved through the event service and is based on the Subscribe/Publish Pattern. To deal with events, your application integrates the event service, which is in charge of published events.
A `TopicsEvent` associates a topic with a type of event. This class defines which events are available and specifies which data to propagate with this type of event.
To receive events concerning your application, first define `TopicsEvent` s for each service, then subscribe to these `TopicsEvent` s with the `IEventService` interface. Then, the `IEventService` interface can use pull or push mode to retrieve the events published by a service, as shown in the figure below.



The Integrated Event Service

This diagram shows that subscribed topics allow your application to retrieve the correct events. Notice that the other Agent Interaction Services do not provide any management related to events. Incoming events reflect changes in the Genesys Framework—for example, the contact e-mail address is modified or an e-mail is properly sent.
Events are specialized. For example, a `VoiceMediaEvent` is an agent event on voice media and strictly involves the agent service. If your agent service requests a login on a DN for the `agent0` agent, your event service receives a `VoiceMediaEvent` as soon as `agent0`' s login is successful.
For each service, the associated event names are listed in the interface description. For each type of event, you can see the list of available attributes to retrieve with the received event. You define the

attributes to propagate with the event in the same TopicsEvent that specifies the event to which to subscribe.

Understanding the Event Service

The event service is designed to optimize the network activity. Once you have subscribed to the events of a set of services, you get all the events in a single request, in either push or pull mode. The following subsections introduce principal concepts of the event service, and of the classes of the com.genesyslab.ail.ws._event namespace, that you should take into account in your application design.

Events Associated with Services

As presented in Event Service Overview, the event service receives all of your application's events. The other services integrated into your application do not deal directly with events.
However, these services are interfaces for a set of objects. Events can occur on the objects hidden by a service. Therefore, each service has its own set of events, which are designed to be appropriate to activites for that service. A few services, such as the SRL and resource services, have no events, because their use is restricted to simple data access.
To find the list of events for any particular service in the *Agent Interaction SDK 7.6 Services API Reference*, open its service interface. For example, under com.genesyslab.ail.ws.agent, open the IAgentService interface, scroll past its list of attributes (in domain:attribute notation) to find the available types of events:

- VoiceMediaEvent
- MediaEvent
- PlaceChangedEvent

For each service, the attributes that have an event property are likely to be published in the service events. Event descriptions in the *Agent Interaction SDK 7.6 Services API Reference* list all the attributes published by each event.

Understanding TopicsEvents and Events

To receive events, you define TopicsEvents for each event type to which you want to subscribe. TopicsEvent is a class of the com.genesyslab.ail.ws._event namespace that has the following attributes:

- eventName—The string type of the targeted events (for example, MediaEvent).
- filters—An array of key-value pairs defined to filter this type of event.
- triggers—An array of one or more key-value pairs defined to select events occurring on specific Genesys objects.
- attributes—A string array specifying keys for the attribute list of the event.

TopicsEvents use:

- Triggers and filters to define which specific events you want to receive.

- List of attribute keys to retrieve values for service attributes propagated with the event.

The following subsections explain these aspects of event handling.

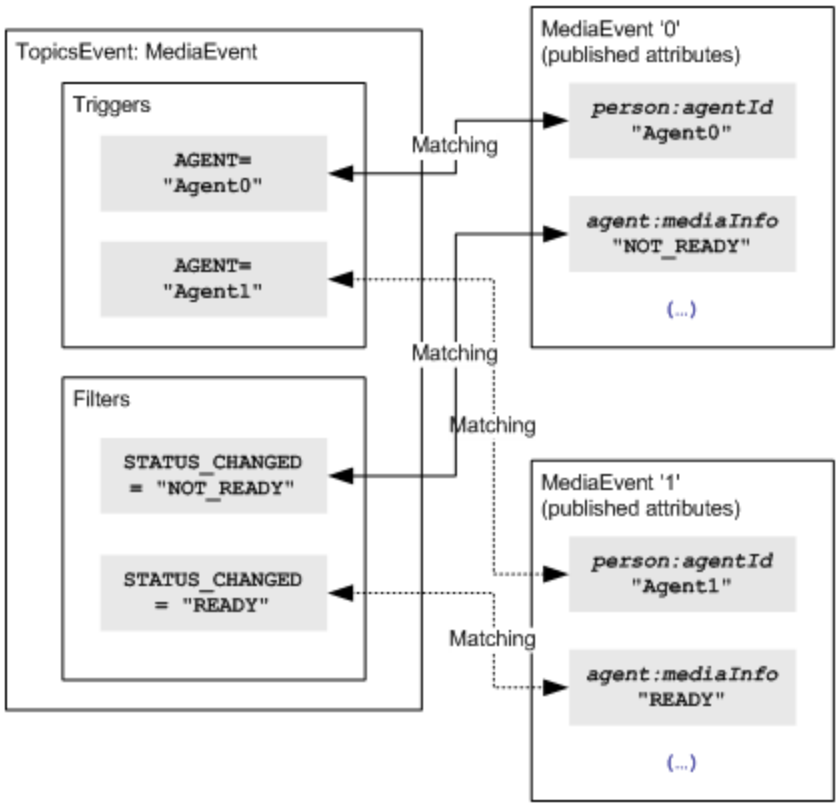
Understanding Triggers and Filters

Triggers identify the Genesys objects involved in an event. For example, if your application uses the agent service to perform agent actions on e-mail media, your application can subscribe to `MediaEvents`. Your application specifies a trigger, in this case, which agent to monitor—for example, `agent0`—so as to receive any `MediaEvents` involving `agent0`.

Filters identify specific values of some attributes published with events. If your application sets no filters, it receives any event that matches a trigger. If your application set some filters, it receives events that match one of the filter values.

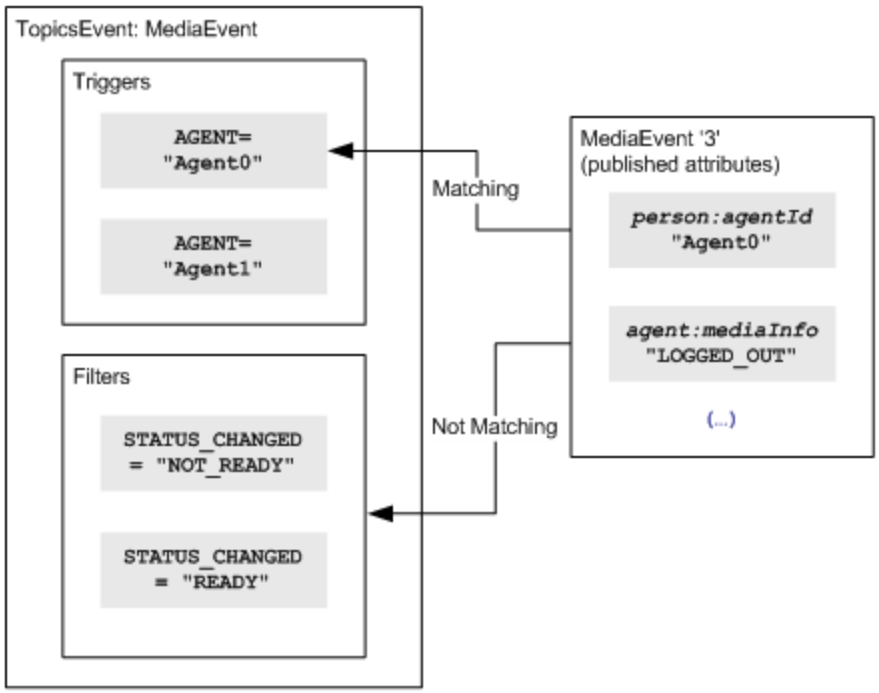
For example, your application can define a filter so as to receive `MediaEvents` only for a specific status change in the media. If `agent0` performs a successful login on certain media, your application might receive a `MediaEvent` due to a status change and associated with the `NOT_READY` agent media status. Your application can choose to receive only these events.

Triggers and filters are values or fields of some published attributes. An event matches a `TopicsEvent` if its published attributes match one of the triggers and one of the filters. If no filter is defined, then the event just has to match the trigger.
The following figures below present the general matching process for triggers and filters.



MediaEvents Matching a TopicsEvent

The first figure shows an example of what happens on the server-application side when an `IEventService` has subscribed to a `TopicsEvent` for a `MediaEvent`. When the server-side application receives a `MediaEvent`, it checks with the `TopicsEvent` to determine whether one of the triggers and one of the filters match. If so, the `IEventService` can retrieve an `Event` object corresponding to the `MediaEvent`.



MediaEvent Not Matching a TopicsEvent

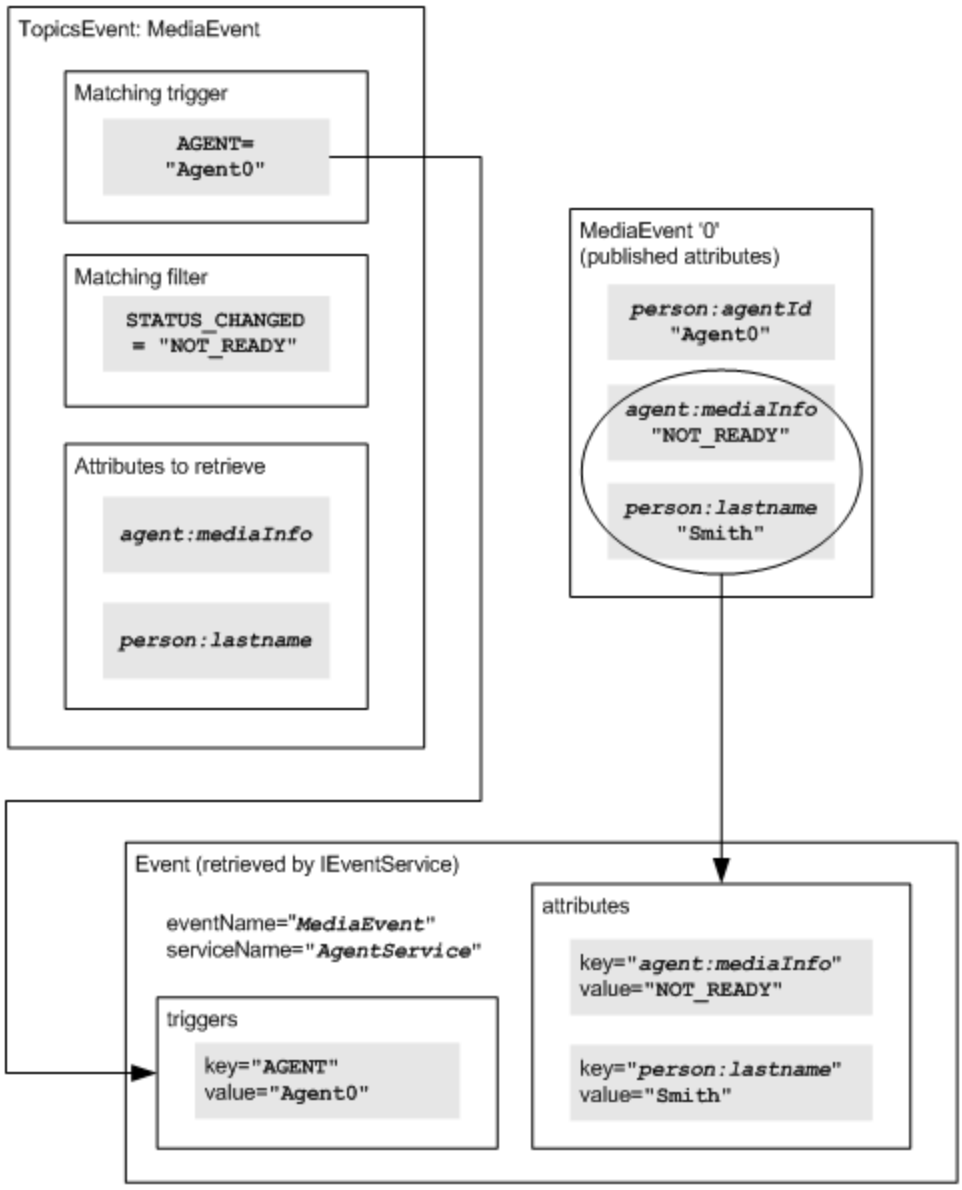
The second figure shows a `MediaEvent` that does not match a `TopicsEvent` defined for `MediaEvent`. Although the event matches the `Agent0` trigger, no filter corresponds.

Retrieved Events and TopicsEvents

Whatever type of event is received on the server-side application, the `IEventService` interface retrieves only `Event` objects.
The `Event` class is part of the `com.genesyslab.ail.ws._event` namespace. Its attributes include the following:

- `eventName`—A string representing the event type.
- `serviceName`—A string representing the service name involved in the event.
- `triggers`—A key-value array of the triggers matched by the event.
- `attributes`—A key-value array of the published attributes propagated with the event.

In each event description in the *Agent Interaction SDK 7.6 Services API Reference*, the published attributes are listed. Only these attributes can be propagated in the `Event.attributes` field.
The `TopicsEvent` class lets your application specify the keys of the published attributes to retrieve with an `Event`.
The following figure illustrates the relationship between the attributes keys of a `TopicsEvent`, the published attributes of an event, and the key-value pairs propagated with an `Event` object.



TopicsEvent and Event Relationship

As shown in above, the attribute keys specified in the `TopicsEvent` determine which attributes are propagated in the `Event` object retrieved by the `IEventService`.

> **Important**
>
> There is no need to propagate filters in order to use them. Filters are independent from the published attribute values.

Understanding TopicsServices

The `TopicsService` class lets your application subscribe to the `IEventService` interface. A `TopicsService` associates a set of `TopicsEvent` with a service, as presented in the following diagram.

The TopicsService Class Diagram

Your application should subscribe to general `TopicsService` objects for every service that your application integrates. The `IEventService` interface offers a set of features to dynamically remove, add, or modify these objects, according to your application needs, as presented in the following sections.

Handling Topics Objects

According to your requirements, your application must deal with services' events. Therefore, your application must subscribe to `TopicsService and TopicsEvents` to define the set of events to retrieve.
These classes are part of the `com.genesyslab.ail.ws._event` namespace, as detailed in the following subsections.

Building TopicsEvent

The `TopicsEvent` class is used to define the events to which to subscribe.

- Your application can specify triggers and filters to determine the list of events to receive.
- Your application can specify the attributes to retrieve in a DTO (Data Transfer Object) when the targeted events occur.

Each `TopicsEvent` is dedicated to a single type of event. It defines, for example, which `MediaEvent` to receive for the agent service.

```
/// Defining a topic event for MediaEvent
TopicsEvent myTopicsEvent = new TopicsEvent() ;
myTopicsEvent.eventName = "MediaEvent" ;
```

Defining Triggers and Filters

Triggers and filters are (respectively) `TopicsEvent.triggers` and `TopicsEvent.filters` attributes. The filter is related to the event occurrence and the trigger to the identifier of the monitored object.

See Understanding Triggers and Filters for further explanation.

> **Important**
> The `trigger` attribute is mandatory when filling in a `TopicsEvent` object.

Triggers and filters are both `Topic` objects (see The TopicsService Class Diagram).
The `Topic` class is a simple container for a key-value pair. For example, the following code snippet shows how to set some triggers and filters for a `MediaEvent`.

```
/// Defining the filter for the TopicsEvent.
myTopicsEvent.filters = new Topic[1];

/// Defining a filter for a specific media status
myTopicsEvent.filters[0] = new Topic();
myTopicsEvent.filters[0].key = "STATUS_CHANGED";
myTopicsEvent.filters[0].value = "NOT_READY";

/// Defining the trigger agent0.
myTopicsEvent.triggers = new Topic[1] ;

/// Specifying the targeted agent
myTopicsEvent.triggers[0]= new Topic();
myTopicsEvent.triggers[0].key = "AGENT" ;
myTopicsEvent.triggers[0].value = “agent0”;
```

The above code snippet specifies retrieval conditions for each `MediaEvent` occurring on `agent0` with a `NOT_READY` agent media status, as follows:

- If your application does not define any other trigger and filter for the `MediaEvent`, it only retrieves events with these characteristics.
- If your application sets a `null` value for the `TriggerFilter.filter` attribute, it retrieves any `MediaEvent` occurring on `agent0`.

For further information about the existing key-value pairs for triggers and filters, refer to the events description in the *Agent Interaction SDK 7.6 Services API Reference.*

Propagated Attributes

The TopicsEvent.attributes field defines the published attributes to retrieve for the events that match a trigger and a filter (if filters are defined). See Retrieved Events and TopicsEvents for further details. Your application can only retrieve attributes that have an event property (as specified in their description that appears in services’ attribute lists, in the API reference.)
The following code snippet sets a list of `MediaEvent` attributes to retrieve.

```
/// Defining a topic event for MediaEvent
TopicsEvent myTopicsEvent = new TopicsEvent() ;
myTopicsEvent.eventName = "MediaEvent" ;

/// Setting the filters and triggers
///...
/// Setting the key list of attributes to retrieve in the Event
myTopicsEvent.attributes = new String[] { "agent:mediaInfo", "agent:mediaAgentStatus",
```

```
"agent:mediasActionsPossible"} ;
```

Wildcards

Your application can employ wildcards when setting the TopicsEvent.attributes field. Genesys recommends that your application rather uses the `default` wildcard than the * wildcard. Default attributes are the most commonly used attributes in applications based on this SDK, and they should provide your application with most values it needs, without increasing significantly the activity on the network. At the contrary, the usage of the * wildcard could disturb the network traffic and reduce your application's performances.
In the following code snippet, the `default` wildcard specifies that the default attributes in the agent domain having an event property are propagated. For further information about wildcards, see Data Transfer Object.

```
myTopicsEvent = new TopicsEvent() ;
myTopicsEvent.eventName = "MediaEvent" ;
/// Retrieving all the agent attributes
myTopicsEvent.attributes = new String[] {"agent:default"};
/// ...
```

Building TopicsServices

The `TopicsService` class associates a specific service with an array of `TopicsEvent` to which to subscribe (see the TopicsService Class Diagram).
The `TopicsService.TopicsEvents` array must contain `TopicsEvent` objects for events occurring for the`TopicsService.serviceName` service.
For example, `MediaEvent`, `VoiceMediaEvent`, and`PlaceChangedEvent` might occur if your application uses the agent service. They can be specified in the `TopicsEvent` objects of a `TopicsService` object dedicated to the agent service.
The following code snippet defines a `TopicsService` object for the agent service. Its `TopicsEvents` lets your application subscribe to `MediaEvent` and `VoiceMediaEvent` only.

```
/// Creating a TopicsService for the Agent Service
TopicsService myTopicsServices = new TopicsService() ;
myTopicsServices.serviceName = "AgentService" ;

/// Creating Topics Events for the Agent Service
TopicsEvent[] myTopicsEvents = new TopicsEvent[2] ;

/// Defining a topic event for MediaEvent
myTopicsEvents[0] = new TopicsEvent() ;
myTopicsEvents[0].eventName = "MediaEvent" ;
/// ...

/// Defining a topic event for VoiceMediaEvent
myTopicsEvents[1] = new TopicsEvent() ;
myTopicsEvents[1].eventName = "VoiceMediaEvent" ;
/// ...
/// Adding the previous TopicsEvents to the TopicsService object
myTopicsServices.topicsEvents = myTopicsEvents ;
```

> **Important**
> Refer to the *Agent Interaction SDK 7.6 Services API Reference* for more information about available events: See services' interface descriptions.

Subscribing to the Events of a Service

Your application can subscribe to several topics' services. To do so, it must:
Get an event service.

1. Build an array of `TopicsService`.
2. Create a subscriber.
3. Subscribe to the topics.

Initial Subscription

Next, create a `TopicsServices` array that includes `TopicsEvents` to which your application must subscribe, as illustrated in the following code snippet:

```
/// Creating the array of topics
TopicsService[] myTopicsServices = new TopicsService[2] ;
myTopicsServices[0] = new TopicsService() ;
myTopicsServices[0].serviceName = "AgentService" ;
/// ....
myTopicsServices[1] = new TopicsService() ;
myTopicsServices[1].serviceName = "InteractionService" ;
/// ....
```

For further information on `TopicsServices`, see Building TopicsServices.
Once the array is filled, create a subscriber:

```
/// Creating a Subscriber
SubscriberResult mySubscriber = myEventService.createSubscriber(null,myTopicsServices) ;
```

> **Important**
> Use this `SubscriberResult` for your further subscribing and unsubscribing operations. This ensures the use of a single subscriber for your application.

Further Subscriptions

During runtime, your application's needs for event-propagated data can change. Your application can define new `TopicsService` objects and use the `IEventService.subscribeTopics()` method to subscribe to them, as presented in the following code snippet:

```
/// Creating the array of new topics
TopicsService[] newTopicsServices = new TopicsService[2] ;
```

```
///...
/// Subscribing
myEventService.subscribeTopics( mySubscriber.subscriberId, newTopicsServices);
```

> **Warning**
> When your application subscribes to a `TopicService` using a `TopicsEvent` with a trigger that has already been subscribed, filters and attributes are all replaced by new ones.

Remove Subscriber

Before your application logs out from GIS, first it must remove its subscriber, as shown in the following code snippet.

```
myEventService.removeSubscriber(mySubscriber.subscriberId);
```

Unsubscribing from Topics

Your application may unsubscribe from `TopicsServices`, or modify `TopicsEvents`' content, during application runtime to fulfill your application's needs. The following subsections detail the corresponding `IEventService` features.

Removing All the Topics Events

Your application can remove all the `TopicsEvents` for all the services. Use the `IEventService.unsubscribeAllTopics()` method.
The following code snippet unsubscribe from all the topics objects defined for your application subscriber:

```
myEventService.unsubscribeAllTopics(mySubscriber.subscriberId);
```

All the `TopicsEvents` previously defined with a `TopicsService` are removed. Your application receives no further events.

Removing Specific Topics for a Service

The process of removing a specific topic for a service is similar to the subscription process. Instead of subscribing to a `TopicsService` array, your application unsubscribes using a `TopicsServiceRemove` array.
A `TopicServiceRemove` object is dedicated to a service and includes the `TopicsEventRemove` objects that list the removed events for this service. The removed events are associated with a trigger.
The following code snippet removes the trigger `agent0` of the `MediaEvent` for the agent service:

```
/// Defining the trigger
Topic myTriggerToRemove = new Topic();
myTriggerToRemove.key = "AGENT";
myTriggerToRemove.value = "agent0";

/// Creating the array of event to remove
```

```
TopicsEventRemove[] myTopicsEventToRemove = new TopicsEventRemove[1];
myTopicsEventToRemove[0] = new TopicsEventRemove();

/// Setting the trigger for the MediaEvent
myTopicsEventToRemove[0].eventName = "MediaEvent";
myTopicsEventToRemove[0].triggers = new Topic[1];
myTopicsEventToRemove[0].triggers[0] = new Topic();
myTopicsEventToRemove[0].triggers[0] = myTriggerToRemove;

/// Creating the array of TopicsServiceRemove
TopicsServiceRemove[] myTopicsServiceToRemove =
new TopicsServiceRemove[1];

/// Creating a TopicsServiceRemove for the Agent Service myTopicsServiceToRemove[0] = new
TopicsServiceRemove();
myTopicsServiceToRemove[0].serviceName="AgentService";

/// Associating the previous topics with the Agent Service
myTopicsServiceToRemove[0].topicsEventsRemove = myTopicsEventToRemove;
/// Unsubscribing
myEventService.unsubscribeTopics( mySubscriber.subscriberId, myTopicsServiceToRemove);
```

The above code snippet ensures that subsequent `MediaEvents` retrieved with the mySubscriber.subscriberId no longer involves events for `agent0`.

Handling Subscription Errors

When your application subscribes or unsubscribes, the topics objects are processed sequentially: If an error occurs for one topic, the remaining topics are processed. The errors are returned in an array of `TopicServiceError` objects, as shown in the following code snippet:

```
/// subscribing to topics
TopicServiceError[] myTopicsServiceErrors = myEventService.subscribeTopics(
mySubscriber.subscriberId, myTopicsServices);
/// Displaying the topics errors
foreach(TopicServiceError err in myTopicsServiceErrors)
{ System.Console.WriteLine("Subcr. error for event {0}: key = {1} val = {2}", err.eventName,
err.filter.key,
err.filter.value.ToString());
}
```

In the above code snippet, the event service processes a subscription and errors are displayed in the console.

Getting Events

There are two available modes to get events:

- Pull mode—your application retrieves the events.
- Push mode—your application is notified of the events.

Pull Mode

In pull mode, your application must periodically retrieve events; it is not notified when an event

occurs. The server-side application waits for the client-side application request to deliver the subscribed events.

Retrieving Events

To retrieve events, your application defines topics for the services, then subscribes to these topics. See Subscribing to the Events of a Service.
Once your application has subscribed, it can retrieve events associated with the `SubscriberResult.subscriberId` identifier by calling the `IEventService.getEvents()` method. The following code snippet is an example of a `getEvents()` call:

```
/// Retrieving the last occurred events /// timeout in seconds is set to 1
Event[] events = myEventService.getEvents(mySubscriber.subscriberId, 1);

/// Displaying the events
foreach(Event evt in events)
{
   System.Console.WriteLine("Occurred {0}",evt.ToString());
}
```

> **Warning**
> If you set a non-zero value for the timeout parameter of the `IEventService.getEvents()` method, this method does not return until either an event occurs or the timeout is reached.

Specifics

In pull mode, the subscriber must be sure to retrieve the events before the server-side timeout is reached.

> **Warning**
> The default timeout is 10 minutes. If no event has been retrieved within 10 minutes, the subscriber is removed.

Push Mode

In push mode, your application is notified of events as they occur. Your application must:

1. Implement the notifyEvents() method of a class inheriting the INotifyService interface.
2. Subscribe to the event service.

Then, during runtime, whenever events occur, the `notifyEvents()` method is called and its code content is executed.

Using the INotifyService Interface

Your application must create a class inheriting the `com.genesyslab.ail.ws._event.INotifyService` class. This inherited class must implement the `INotifyService.notifyEvents()` method.
The following code snippet presents a short implementation of an inherited class. This class' `notifyEvents()` method displays, in the console, the content of reported events.

```
public class NotificationImpl : com.genesyslab.ail.ws._event.INotifyService
{
  public void notifyEvents(string subscriberId, com.genesyslab.ail.ws._event.Event[] events) {
    if (events == null)
    {
      System.Console.WriteLine("notifyEvents - null \n"); return ;
    }
    System.Console.WriteLine( "notifyEvents getEvents : " + events.Length + "\n") ;
    foreach( Event evt in events)
    {
       System.Console.WriteLine( "Service :"+ evt.serviceName + "Event: "+ evt.eventName +
                                 "timeStamp:"+ evt.timeStamp +"\n");
    }
  }
}
```

Subscribing

Use an instance of your inherited `INotifyService` class to fill the notif.notificationEndpoint field.

```
Notification notif = new Notification();
notif.notificationEndpoint = new NotificationImpl();
```

Then, subscribe to the `Notification` instance:

```
SubscriberResult result = myEventService.createSubscriber(notif,myTopicsServices) ;
```

Reading DTOs in Events

When your application subscribes to events, it specifies a set of published attributes to retrieve with the events (see Building TopicsEvent).
The attributes can be accessed with the `Event.attributes` attribute, which is a `KeyValue` array. The following code snippet is a pull-mode example:

```
/// Retrieving the last occurred events
Event[] events = myEventService.getEvents(mySubscriber.subscriberId, 1);
foreach(Event evt in events)
{
   KeyValue[] attributes = evt.attributes ;
   foreach( KeyValue attr in attributes)
   {
       System.Console.WriteLine( "Service: {0}\tKey: {1} value: {1}", evt.serviceName,
attr.key, attr.value) ;
   }
}
```

The above code snippet displays the attribute key-value pairs retrieved with the events.

Event Notification in Java

This section describes how to use Interaction SDK (Web Services) Notification with Java. Several solutions are available to use unsolicited events in Java with GIS.
In this section, we use the Apache Axis SOAP toolkit, version 1.1, to implement a client-side notification mechanism in a simple notification server.
This example supposes that we have GIS running on host <GIS_HOST> and port <GIS_PORT>. All the following subsections are related to this example.

Notification Classes Generation

To generate classes used in notification events, we will use WSDL2java, a tool provided by Apache Axis. Replace the italicized placeholders when typing the following command line:

```
java org.apache.axis.wsdl.WSDL2Java
-o ''output''
–server-side ''http://<GIS_HOST>:<GIS_PORT>/gis/services/AIL_NotifyService?wsdl''
```

The required classes will be generated in the directory specified by output. These classes must be added to your source path. The WSDL2java tool generates a mapping file that maps the SOAP types to Java classes.
The tool generates the classes for each type from WSDL, using a type-mapping file (deploy.wsdd). It also generates the following server implementation class:

```
com/genesyslab/www/services/ail/wsdl/event/NotifyServiceSoapBindingImpl.java
```

This class has a method notifyEvents(String subscriberId, Event[]events), which is called on each notification event, as shown in the following example:

```
public void notifyEvents(String subscriberId, Event[] events) throws
java.rmi.RemoteException,    com.genesyslab.www.services.ail.wsdl.event.WServiceException
{
   // Put action to process for each event received here
}
```

Simple Notification Server

This subsection introduces the implementation of a simple notification server for your client application. To achieve this, you can use a little server provided by the Axis toolkit and identified as the following class:
org.apache.axis.transport.http.SimpleAxisServer
To provide it with all the deployment information included in the deploy.wsdd file, start it as shown in the following code snippet:

```
org.apache.axis.client.AdminClient adminClient = new org.apache.axis.client.AdminClient();

String[] argsDeploy = {"deploy.wsdd", "-p", Integer.toString(<CLIENT_PORT>)};

adminClient.process(argsDeploy);
```

Once the server is started, you can browse `Notify Service` on the client side at: `http://client_host:client_port/axis/services/NotifyService?wsdl` When creating a subscriber in your application for the event service, you must define the notification location by setting the following fields to:

- `notificationEndPoint—http://<CLIENT_HOST>:<CLIENT_PORT>/axis/services/NotifyService`
- `notificationType—SOAP_HTTP`

The Agent Service

The agent service is the `IAgentService` interface defined in the `com.genesyslab.ail.ws.agent` namespace. To be able to manage agent features, your application has to integrate this interface and uses classes.

Introduction

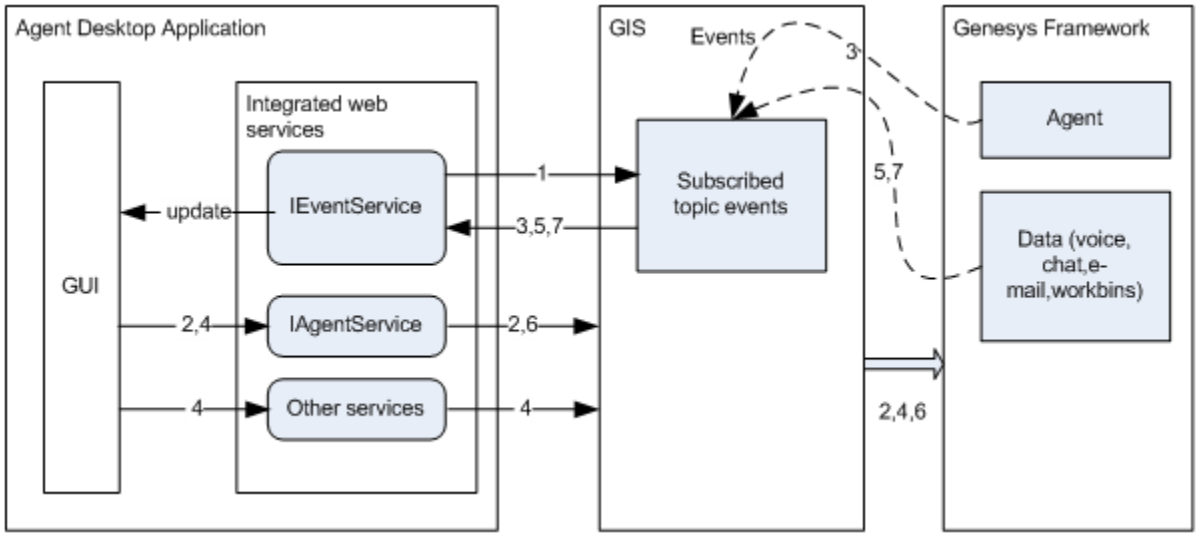
The agent service is used to access the agent features managed on the server-side application. This service is required for most of the applications developed with the Agent Interaction Service API. Without it, you cannot access most of the features with others services that require an identified agent.

For example, if your application has to make a call, it requires an `IInteractionVoiceService` to access to a DN performing the call (see Voice Interactions). To access to this DN, your application needs to login an agent on the DN. Therefore your application needs to integrate an agent service.

To use correctly the agent service, your application must take into account integrated services events including agent services events when subscribing `TopicsEvent` with the `IEventService` interface. (See The Event Service).

Then, your application needs to log in an agent with the `IAgentService` interface. If the login is successful, your application receives the corresponding events and is able to use other services features, including agent service features. Services actions—corresponding to the Agent Interaction Service features—generate events that are taken into account according to the current `TopicsEvent` subscribed by your `IEventService`. Once your agent service logs out agents, services that depend on a logged in agent do not fulfill action requests.

The figure below demonstrates a typical request and event flow.



The Integrated Agent Service

Subscribing to TopicsEvent for integrated services, including agent service.

1. Requesting an agent login with the agent service login action.
2. Receiving an agent event for a successful agent login.
3. Requesting services actions depending on the logged agent, including agent services actions (ready, not ready, and so on).
4. Receiving events due to services actions.
5. Requesting an agent logout with the agent service logout action.
6. Receiving an agent event for logout.

The Integrated Agent Service shows the general agent service handling in an application integrating services depending on logged agents. As you can see, the agent service plays a predominant and determinant role for services requiring an agent.
The agent service is designed to:

- Access all the data related to a set of agents.
- Access agents status and possible agents actions for any logged agent.
- Perform agent actions on media, such as login, logout, ready, not ready, or aftercallwork actions.

A characteristic of this agent service is that voice media have dedicated classes to handle voice media data, whereas other media–chat and e-mail–use a set of common media classes.

Moreover, some agent actions require a filled form handling action data.

Agent Service Essentials

This section introduces essential aspects for using features of the agent service needed in most agent desktop applications developed in the Agent Interaction Service API.

Agent and Statuses

The agent status is not a general status of an agent. Rather, an agent status is related to a medium, that is an agent status is the status of an identified agent for a particular medium.

Using an Agent Status

The agent media status is modified:

- Each time an action requested by the agent service is performed on the media.
- When the media is impacted by a performed action that another service requested.
- When the media is affected by an internal change (for example, a system issue).

In terms of development requirements, take into account agent media statuses so as to:

- Inform your agent of its current status on a media and of any status change.
- Base on statuses your application design.

Agent Status for Voice Media

Voice media involve underlying switches and imply a dedicated management taken into account by the voice state model.

The Existing Voice Media Statuses

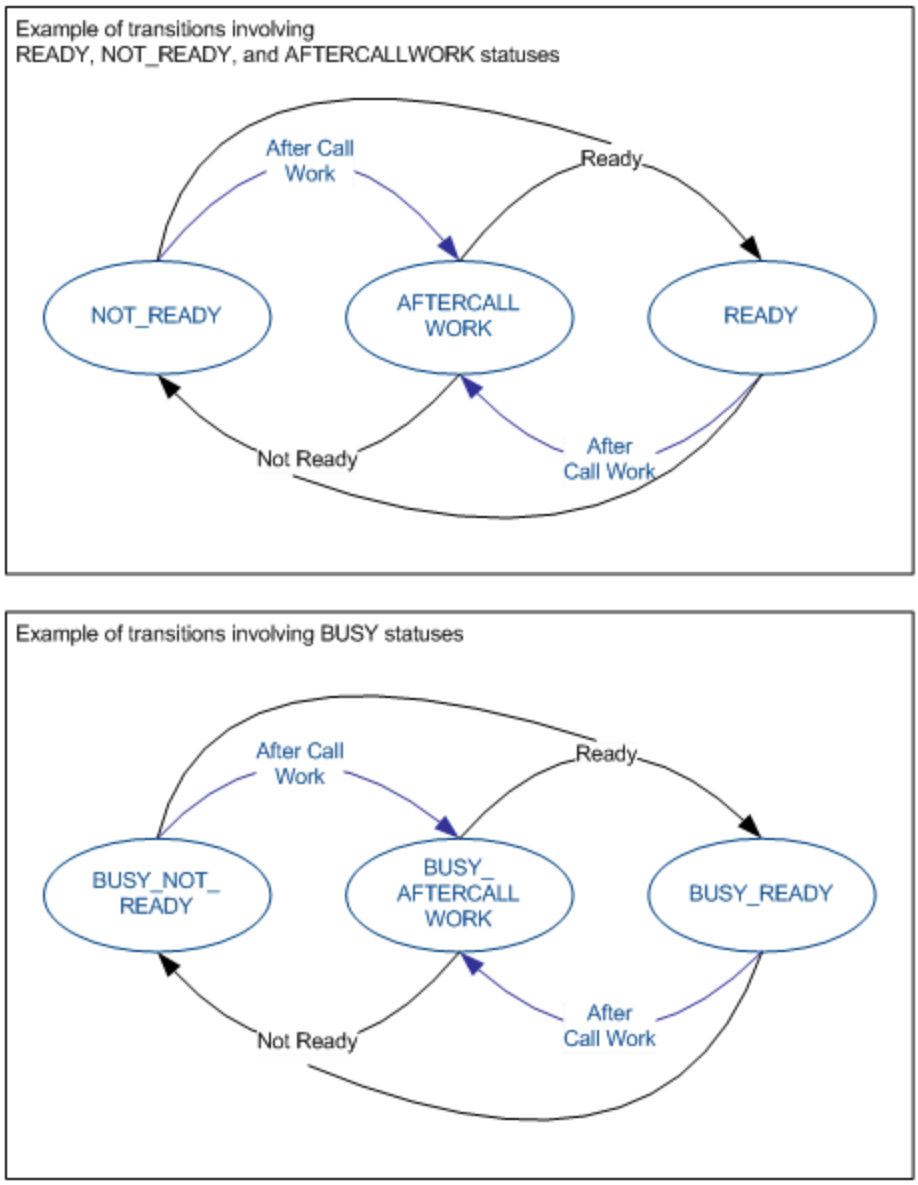
The `VoiceMediaStatus` type is an enumerated type describing the agent status on a DN:

- `LOGGED_OUT`—the agent is logged out.
- `READY`—The agent is logged and ready to work on the DN.
- `BUSY_READY`—The agent is busy.
- `NOT_READY`—The agent is logged and not ready for working.
- `BUSY_NOT_READY`—The agent is busy and not ready.
- `BUSY_LOGGED_OUT`—The agent is busy and logged out.
- `AFTERCALLWORK`—The agent is in After Call Work mode.
- `BUSY_AFTERCALLWORK`—The agent is busy and in After Call Work mode.
- `OUT_OF_SERVICE`—The DN is out of service.

> **Important**
>
> The `VoiceMediaStatus` type is defined in the `com.genesyslab.ail.ws.place` namespace as it is a media information. See Place, DNs, and Media for further details.

> **Warning**
>
> For voice media, some statuses depend on the switch capability. For example, the `AFTERCALLWORK` status is usable only if the switch offers this feature.

The following figure shows examples of the state transition that can occur due to calls to agent service features.



Examples of State Diagrams for Voice Media

> ## Warning
> This figure is provided as an informative example. It does not include all the existing transitions.

The transition from one state to another is not always due to a performed agent service action. State changes can occur due to other reasons, as for example, a T-Server event, or internal management

on the server-side application.

> **Warning**
> Do not assume any status sequence in your application design. Design your application always to update with the provided possible agent actions and current agent status.

Dealing with Voice Media Statuses

The statuses are handled in the `VoiceMediaInfo` class of the`com.genesyslab.ail.ws.place` namespace. This class associates the status `VoiceMediaInfo`.status with a DN identifier `VoiceMediaInfo`.dnId.
You can retrieve `VoiceMediaInfo` objects with the following attributes of the `IAgentService` interface:

- `agent:dns`—`VoiceMediaInfo` array containing all the DNs of the current place of the agent.
- `agent:loggedDns`— `VoiceMediaInfo` array containing all the DNs where the agent is logged.

Getting these attributes values is done with agent DTO classes of the `com.genesyslab.ail.ws.agent` namespace as shown in the following code snippet:

```
/// Retrieving the DTOs
PersonDTO[] agentDTO = myAgentService.getPersonsDTO(new string[]{ "agent0" }, new string[]{
"agent:loggedDns"});
///Getting the VoiceMediaInfo array
VoiceMediaInfo[] arrayInfo = (VoiceMediaInfo[]) agentDTO[0].data[0].value ;

/// Displaying the status for each DN
foreach(VoiceMediaInfo info in arrayInfo)
{
   System.Console.WriteLine("agent0 status on "+ info.dnId +" is
"+info.status.ToString()+"\n");
}
```

> **Important**
> Attributes related to statuses are published in events. You should update your agent desktop application with the statuses of the agent when a `VoiceMediaEvent` occurs with a matching `STATUS_CHANGED` filter.

Agent Status for Other Media

Chat and e-mail are standard media and their agent media statuses management is similar to agent voice media statuses.

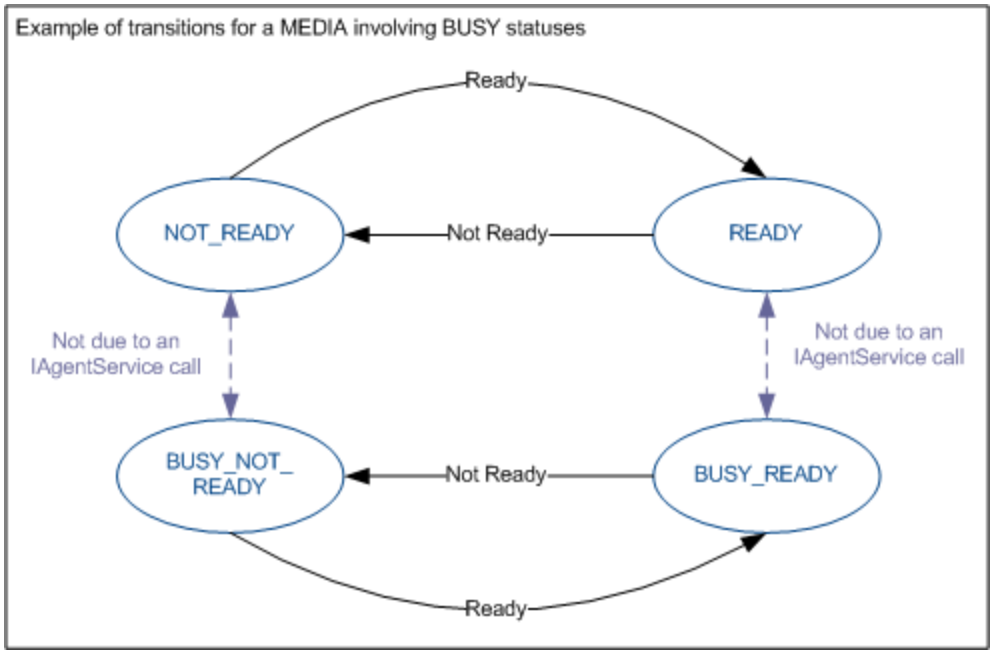
The Existing Media Statuses

The`MediaStatus` type is an enumerated type describing the agent status on a medium:

- `LOGGED_OUT`—The agent is logged out.
- `READY`—The agent is logged and ready to work with the media.
- `BUSY_READY`—The agent is busy.
- `NOT_READY`—The agent is logged and not ready for working.
- `BUSY_NOT_READY`—The agent is busy and not ready.
- `OUT_OF_SERVICE`—The medium is out of service.

> **Important**
>
> The MediaStatus type is defined in the com.genesyslab.ail.ws.place namespace as it is a media information. See Place, DNs, and Media.

The diagram example below shows examples of state transitions that can occur due to an `IAgentService` feature call.



An Example of Agent Media Status Diagram

> ### Warning
> This figure is provided as an informative example. It does not include all the existing statuses and transitions.

The transition from one state to another is not always due to a performed agent service action. State changes can occur due to other reasons, as for example, a T-Server event, or internal management on the server-side application.

> ### Warning
> Do not assume any status sequence in your application design. Design your application always to update with the provided possible agent actions and current agent status.

Dealing with Media Statuses

The `MediaInfo` class of the`com.genesyslab.ail.ws.place` namespace associates the status `MediaInfo.status` with a media name `MediaInfo.name`.

> **Important**
> Possible values for the MediaInfo.name attribute are chat or email.

You can retrieve MediaInfo objects with the following attributes of the IAgentServiceinterface:

- agent:availableMedias—array of the mediaType available on the current place of the agent.
- agent:loggedMedias—MediaInfo array containing all the media on which the agent is logged.

Dealing with MediaInfo objects is performed with agent DTO classes of the com.genesyslab.ail.ws.agent namespace as shown in the following code snippet:

```
/// Retrieving the DTOs
PersonDTO[] agentDTO = myAgentService.getPersonsDTO(new string[]{ “agent0” }, new string[]{
"agent:loggedMedias"});
///Getting the MediaInfo array
MediaInfo[] arrayInfo = (MediaInfo[]) agentDTO[0].data[0].value ;

/// Displaying the status for each media
foreach(MediaInfo info in arrayInfo)
{
  System.Console.WriteLine("agent0 status on "+ info.name +" is "+
info.status.ToString()+"\n");
}
```

> **Important**
> The IAgentService attributes are published in events. You should update your agent desktop application with the statuses of the agent when a MediaEvent occurs with a matching STATUS_CHANGED filter.

Agent and Possible Actions

In the agent service, voice media actions are differentiated from other media actions. There are two enumerated types defining the existing actions for an agent:

- AgentDnActions —actions that the IAgentService interface can perform on a voice medium, that is, a DN.
- AgentMediaActions—actions that the IAgentService interface can perform on media such as chat and e-mail.

Each action defined in these enumerated types is associated with a method of the IAgentService interface. For example, AgentDnAction.LOGIN represents the IAgentService.login() method that is able to perform the login of an agent on a set of media.
The following sub-sections details possible agent action use with the IAgentService interface.

Understanding Agent Possible Action

The `IAgentService` interface lets your application retrieve the possible actions associated with a media. You can use this information to enable or disable your desktop application's features that use the corresponding calls to your service.
For example, if at a certain point in time a ready action is not possible for a given medium, your application should disable its ready button if the user selects this medium.
The possible actions take into account:

- The current state of the media concerned by the action.
- The capability of the underlying genesys solution to realize the action.

You retrieve these possible actions with DTOs as explained in the following subsections.

> **Warning**
> It is not possible to retrieve the attribute values corresponding to possible actions on media on which no agent is logged.

Voice Media Possible Actions

The possible actions for a voice media are accessed with the `agent:dnsActionsPossible` attribute of the `IAgentService` interface. This attribute contains an array of `DnActionsPossible` associating the current available actions with DN identifiers.
The following code snippet shows how to display the voice available actions for `agent0` using DTO classes of `com.genesyslab.ail.ws.agent` namespace:

```
/// Retrieving the DTOs
PersonDTO[] agentDTO = myAgentService.getPersonsDTO(new string[]{ "agent0" }, new
string[]{"agent:dnsActionsPossible"});

DnActionsPossible[] myPossibleActionsOnDN = (DnActionsPossible[]) agentDTO[0].data[0].value ;

/// Displaying the Possibilities for each DN.
foreach(DnActionsPossible myPossibleActions in myPossibleActionsOnDN)
{
   System.Console.WriteLine("agent0 possible actions on " + myPossibleActions.dnId +" are:");
   foreach(AgentDnAction action in myPossibleActions.agentActions)
   {
      System.Console.Write("\t"+action.ToString());
   }
   System.Console.WriteLine("\n");
}
```

Other Media Possible Actions

The possible actions for other media are accessed with the `agent:mediaActionsPossible` attribute of the `IAgentService` interface. This attribute contains an array of `MediaActionsPossible` associating the current available actions with media names.

> **Important**
> The authorized values for media names are `chat` and `email`.

The following code snippet shows how to display the available actions for `agent0` using DTO classes of `com.genesyslab.ail.ws.agent` namespace:

```
/// Retrieving the DTOs
PersonDTO[] agentDTO = myAgentService.getPersonsDTO(new string[]{ "agent0" }, new
string[]{"agent:mediaActionsPossible"});

MediaActionsPossible[] myPossibleActionsOnMedia = (MediaActionsPossible[])
agentDTO[0].data[0].value ;

/// Displaying the possible actions for each media.
foreach(MediaActionsPossible myPossibleActions in myPossibleActionsOnMedia)
{
   System.Console.WriteLine("agent0 possible actions on " + myPossibleActions.mediaType +"
are:");
   foreach(AgentMediaAction action in myPossibleActions.agentActions)
   {
      System.Console.Write("\t"+action.ToString());
   }
   System.Console.WriteLine("\n");
}
```

Agent and Events

There are two types of agent event related to media:

- `VoiceMediaEvent`—agent events occurring on voice media.
- `MediaEvent`—agent events occurring on chat or e-mail media.

These events enable application update based on agent, place, voice, and media events. The available attributes through these events let your application:

- Update the agent media statuses with:
  - `agent:mediaInfo` for a media
  - `agent:voiceMediaInfo` for a DN.
- Update the possible agent actions for a medium with:
  - `agent:dnActionsPossible` for a single DN
  - `agent:mediaActionsPossible` for another medium such as e-mail.

  The `PlaceChangedEvent` occurs when an agent changes place. This lets your application update GUI components displaying information related to the place.

> **Important**
>
> The place is an object associating an agent with several media. See Place, DNs, and Media

Forms and Agent Actions

The Agent Interaction Service API includes form classes to handle the data related to a medium. Voice is managed differently from other media, such as chat and e-mail.
For most agent actions on media, your application has to define a form containing the parameters required for perform the actions.
The following subsections detail the existing forms and show their use in agent service features.

Forms for Voice Media

The forms used for voice media are classes of `com.genesyslab.ail.ws.agent` namespace. There are several form objects that may be defined in a voice context:

- `LoginVoiceForm`—This class is used to login on voice DNs.
- `LogoutVoiceForm`—This class is used to logout on voice DNs.
- `ReadyVoiceForm`—This class is used to get ready or not ready on voice DNs.
- `AfterCallWorkVoiceForm`—This class is used when getting in an AfterCallWork status on voice DNs.

Each form is dedicated to an action. However, the data are characteristics of the voice media. For example, the LoginVoiceForm class is used to fill forms required for a login and includes the whole list of the data involved in voice forms. The following code snippet shows how to fill this form.

```
// Creating the voice form for the login of the agent0
LoginVoiceForm myLoginVoiceForm = new LoginVoiceForm();

// Defining the set of DNs concerned with the login
myLoginVoiceForm.dnIds = new string[]{ "DN0" };

// Setting the login ID of the agent0
myLoginVoiceForm.loginId = "login0";

// Setting the agent password
myLoginVoiceForm.password = "Password0";

// Setting the queue parameter
myLoginVoiceForm.queue = "myQueue";

// Optional: you can set reasons otherwise null
myLoginVoiceForm.reasons = null;

// You can tune your application with switch specific data
myLoginVoiceForm.TExtensions = null;
```

```
// You can specify the workmode for your login.
myLoginVoiceForm.workmode = WorkmodeType.MANUAL_IN;
```

For further information on each form attributes, see the *Agent Interaction SDK 7.6 Services API Reference.*

T-Extensions

T-Extensions are data used by the underlying switches. T-Extensions are switch specific, you can fine-tune your application with these data for a specific switch type.
Use a KeyValue array to specify T-Extension keys-values as shown in the following code snippet:

```
// Specifying a trunk for the G3 Lucent switch
KeyValue[] myTExtensions = new KeyValue[1];
myTExtensions[0]=new KeyValue();
myTExtensions[0].key = “Trunk”;
myTExtensions[0].value = “myTrunk”;
```

> **Important**
>
> For further information about key-value T-Extensions, refer to the T-Server documentation associated with your switch. See also T-Extensions and T-Reasons.

Workmode

Workmode is switch specific. Switches don’t always provide all the workmodes defined by the `com.genesyslab.ail.ws.agent.WorkmodeType` enumeration. Refer to your T-Server documentation for further information.

> **Important**
>
> The workmode can affect the state sequence of the agent. See also Workmode.

Forms for Other Media

Other media -e-mail or chat- require the `MediaForm` class of the `com.genesyslab.ail.ws.agent` namespace to perform agent service actions, such as login, logout, ready, and so on.
This form is a container which specify the media types concerned by the agent service actions.
The following code snippet shows how to define this form for chat and e-mail.

```
// Creating an instance of MediaForm
MediaForm myMediaForm = new MediaForm();
// Defining the types of concerned media
myMediaForm.mediaTypes = new string[]{ "email", "chat" };
```

Agent Login

The `IAgentService` interface has a `login()` method to let agents login on a place for a set of

media defined in their corresponding forms. See Forms for Voice Media and Forms for Other Media.
The following code snippet shows a login of the agent0 agent performed on an authorized place, using the myLoginVoiceForm and myMediaForm that were defined in the previous sections:

```
MediaInfoError[] errors = myAgentService.login( "agent0", "placeForAgent0",
myLoginVoiceForm, myMediaForm);
```

The login is performed on the set of DNs defined in the LoginVoiceForm and on the media that are defined in the place and specified in the types of the MediaForm.
You retrieve in MediaInfoErrors objects both errors and useful information about the login attempt.

Agent Logout

The IAgentService interface has a logout() method to let agents logout a place. The agent performs a logout only for the set of media defined in their corresponding forms.
The following code snippet is an example of a logout performed by the agent0 agent.

```
// Creating a instance of LogoutVoiceForm
LogoutVoiceForm myLogoutVoiceForm = new LogoutVoiceForm();

// Defining the set of IDs concerned by the logout
myLogoutVoiceForm.dnIds = new string[]{ "DN0" };

// Setting the queue and the reason attributes
myLogoutVoiceForm.queue = “myQueue”;
myLogoutVoiceForm.reasons = null;

// Logout action of the agent0 performed on DN0 // and the media defined in myMediaForm
MediaInfoError[] errors= myAgentService.logout( "agent0", myLogoutVoiceForm,
myMediaForm);
```

Getting Ready or Not Ready

The IAgentService interface has ready() and notReady() methods to let agents get ready or not ready on media defined in a place. The agent performs the ready and not ready actions on the set of media defined in their corresponding forms.
The following code snippet is an example of a ready action performed by the agent0 agent.

```
// Creating a instance of ReadyVoiceForm
ReadyVoiceForm myReadyVoiceForm = new ReadyVoiceForm();
// Defining the set of IDs concerned by the ready (or not ready)
myReadyVoiceForm.dnIds = new string[]{ "DN0" };

// Setting the other attributes
myReadyVoiceForm.queue = "myQueue";
myReadyVoiceForm.reasons = null;
myReadyVoiceForm.TExtensions=null;
myReadyVoiceForm.workmode = WorkmodeType.MANUAL_IN;

// Agent0 getting ready on DN0 // and the media defined in myMediaForm
MediaInfoError[] errors =
myAgentService.ready("agent0", myReadyVoiceForm, myMediaForm);
```

The notReady() call is equivalent to the ready() call as illustrated in the following code snippet:

```
MediaInfoError[] errors = myAgentService.notReady("agent0", myReadyVoiceForm,
 myMediaForm);
```

Managing Agent Skills

In 7.6.6, the Agent Service allows to manage Agent skills (add, update, or remove) using the following method:

```
void updateAgentSkills(
   string agentId,
   SkillDTO[] addedSkills,
   SkillDTO[] updatedSkills,
   int[] deletedSkills);
```

> **Important**
>
> To manage skills, the skills must exist in the Configuration Server.

The following code sample shows how you should proceed:

```
//get all skills available in the configuration server
 Skill[] skills = resourceService.getAvailableSkills();
 //Create a Skill DTO
 SkillDTO dto = new SkillDTO();
 // Set the skill level
 dto.level = 11;
 // Get the skill's name in the Configuration Server
 dto.name = skills[0].name;
 // Get the skill's config DB ID
 dto.skillId = Convert.ToInt32(skills[0].id);
 //Now assign a new skill to the given agent
 agentService.updateAgentSkills(anAgent1, new SkillDTO[] { dto }, null, null);
```

Place, DNs, and Media

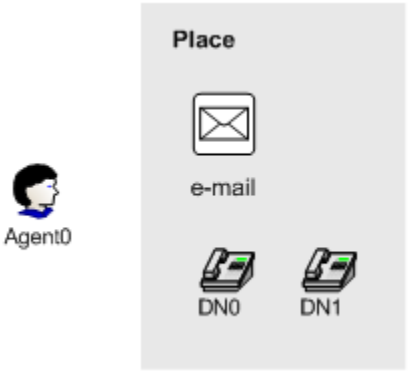
Media are resources that services use to perform actions such as making a call, sending an e-mail, and so on. This chapter covers how media work in an agent's place with the Agent Interaction Service API, how to monitor them, and how to deal with specific voice media, that is DNs.

Introduction

Places, media, and DNs are objects defined in the Configuration Layer. These objects offer important information that your application might have to take into account.

What Are Places?

A place defines a set of media and DNs that can all be used together by a single agent at a given time. Media types are `EMAIL` and `CHAT`, whereas DNs are voice media resources that offer phone-call features. The following figure presents an example of a place defined for the agent `Agent0`.



A Place for Agent0

A place can be associated with a single chat media type and a single e-mail media type but with several DNs.
The place service is designed to provide your application with information about the places defined in the Configuration Layer, and also with information about media and DNs associated with these places.
The place service provides the following features:

- List a place's DN identifiers and media types with their status.
- Retrieve the identifier of an agent logged into a place.
- Retrieve the possible workmodes for a place's DNs.

- Events for place modification:
  - Media added or removed.
  - DNs added or removed.
  - Changes of place identifier.

Place and Other Services

The place has an important role in other services. The agent has first to log into a place, and this logged-in agent can use interaction services only if the requested media are logged in on this place.
Once this requirements are met, the place can help your application perform some management. For example, the `IInteractionService` interface lets your application retrieve all current interactions on a place. Another example is the `IInteractionMailService` interface, which uses the place to pull interactions.

DNs-Voice Specifics

DNs are resources that handle voice interactions. Depending on the underlying switches, DNs have some specific features, for example, the `Do Not Disturb` feature.
The DN service, `IDnService`, deals with all DNs configured in the Configuration Layer. A DN has a string identifier `dnId`, and a callable number that is readable with the `dn:callableNumber` attribute of the DN service. DN identifiers are mandatory in any call to the DN service's methods.
The DN service's features are the following:

- Activate/Deactivate the `Do Not Disturb` feature.
- Activate/Deactivate the `Forward` feature.
- Retrieve the preceding features' statuses (activated or deactivated).
- Retrieve the possible workmodes of a DN.
- Retrieve DN information with DTOs.
- Fine-tune your application with switch specific information.

> **Important**
> To use the DN service features on a DN, your application must have an agent logged into the DN.

Understanding Place, DNs, and Media

The place service does not let your application perform actions on places. The place service is designed to provide your application with information about places' media and DNs.
Your application may use the place service in scenarios like the following:

- Display whether a place is used by an agent.

- Display the available media and DN for a logged place.
- Inform the agent when new media are added to his or her place.

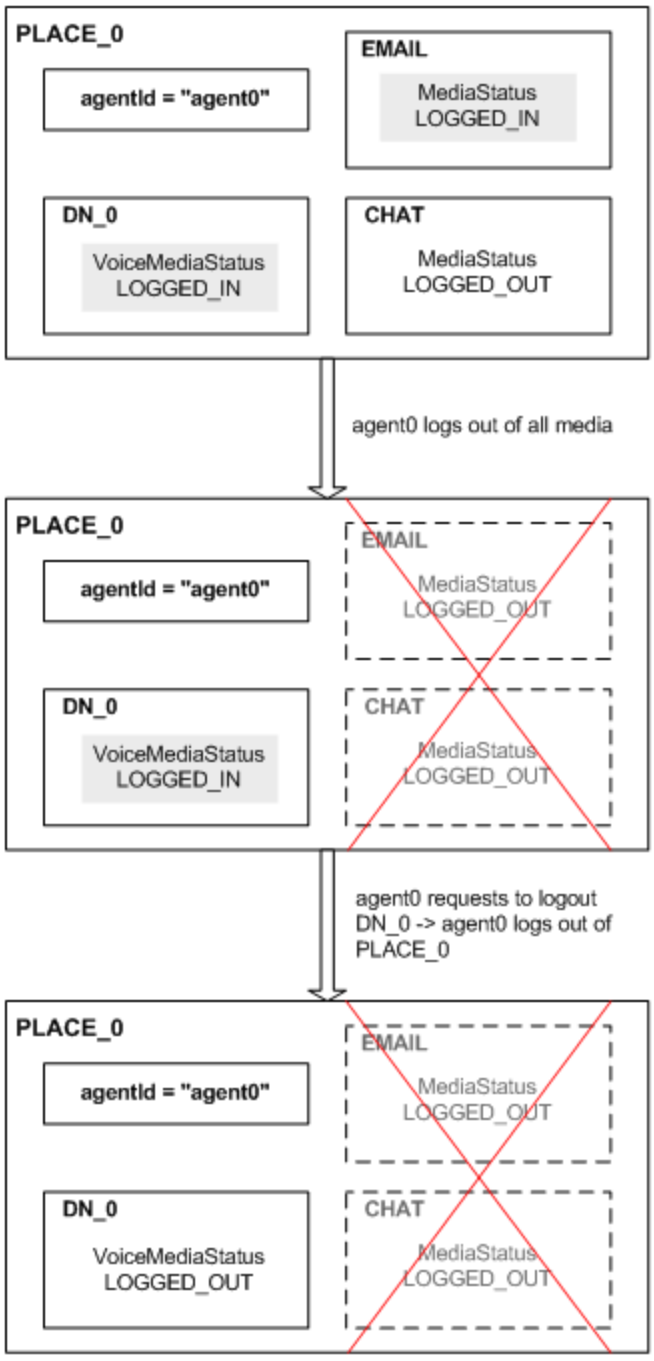
The following subsections specify the relationships amongst place, media, and DN.

Place, Agent, and Statuses

The place service can provide the statuses of places' logged media and DNs. These statuses of media and voice media (DNs) are the statuses of an agent on these media.

Place and Login

When your application logs in an agent with the agent service, the application specifies the place and media on which the agent will log in. For details, see Agent Login.
While the agent is logged into one of the place's media types (or DNs), the place and its media are associated with the agent, as shown in A Place for Agent0.
To start working, the agent must log into the place's media types or DNs which are defined in the Configuration Layer. As long as the agent is logged into one of the place's media types or DNs, the agent is also logged into the place, and the place, its DNs, and its media are associated with the agent.
To get the place's media types, your application takes a login action on at least one of the place's media types. When your application requests a login action on media types, media types are added to the place, and the ones specified in the request are logged in. If the agent logs out of the media types, your application no longer accesses them, as shown in the figure below.



Agent Actions on the Media of a Place

The above figure shows the place `PLACE_0` where agent `agent0` has first logged in on a DN `DN_0` and `e-mail` media of the place.
At runtime, the media only exist in the Interaction Server. To get the media of a place, your application must call the `IAgentService.login()` method and pass in argument a `MediaForm`

instance which contains at least one media type.
As a result, the available media are added to the place, and the ones specified in the request are logged in. If your application logs out media per media, the Interaction Server considers that the place is still logged in.
To definitly logout of the place, your application must call the `IAgentService.logoutMultimedia()` method, that will logout all medias at once, including in the Interaction Server, if you set the `MediaForm.mediaNames` field to `null` for the `mediaForm` passed in parameters, of if you pass an empty `MediaForm` instance as shown in the following code snippet.

```
myAgentService.logout("agent0", "place0", null, new MediaForm());
```

> **Important**
>
> When an agent is logged into one of the place's media types, the agent is logged into the place.

VoiceMediaStatus and MediaStatus

The `VoiceMediaStatus` and `MediaStatus` enumerated types are part of the `com.genesyslab.ail.ws.place` namespace.

VoiceMediaStatus

The `VoiceMediaStatus` type is an enumerated type describing the agent status on a DN when an agent is logged in.
When no agent is logged into the DN, the voice media status is `LOGGED_OUT`, or else `OUT_OF_SERVICE` if there is an issue with the DN. For details, see Agent Status for Voice Media.

MediaStatus

The `MediaStatus` type is an enumerated type describing the agent status on a media type when an agent is logged into a place.
When the agent is not logged into any media type of a place, no media type is available on this place, and your application cannot get any media status for this place.
When the agent is logged into at least one of the place's media types, all media types are available on this place, and your application can get media status for any media type. For logged-out media types, the media status is `LOGGED_OUT`, or else `OUT_OF_SERVICE` if there is an issue with the media type. For details, see Agent Status for Other Media.

Media and Voice-Media Information

The `com.genesyslab.ail.ws.place` namespace offers the `VoiceMediaInfo` and the `MediaInfo` classes to deal with essential information about the media associated with a place.
The place service lets your application retrieve this information using two attributes:

- `place:medias`—An array of `VoiceMediaInfo`, containing information about media of type `MediaType`

associated with a place.

- `place:dns`—An array of `MediaInfo`, containing information about the DNs associated with a place.

VoiceMediaInfo

The `VoiceMediaInfo` class is a simple container for the main data of the voice media (DNs) belonging to a place. The attributes of this class are:

- `dnId`—The identifier of the relevant DN.
- `status`—The `VoiceMediaStatus` of the DN.

MediaInfo

The `MediaInfo` class represents an information container for a media type belonging to a place. The attributes of this class are

- `type`—The `MediaType` of the media type.
- `status`—The `MediaStatus` of the media type.
- `name`—The string to display for this media type.

> **Important**
> Each place can contain zero to one media of each type MediaType. Therefore, the type of the media identifies the media.

Place and DNs Consolidation

Switch DNs' types—for example, ACD position or extension—are mostly switch-specific. They affect the use of features provided by the agent service, such as login, logout, and ready actions.
The underlying AIL library provides consolidation of the DN objects in its model. In all cases, the services expose a single DN, even if a place's configuration requires more DNs in the Configuration Layer for compliance with the underlying switch.
To find the required configuration for each switch, and to thus find which DN identifier is visible, refer to the Interaction SDK Java Deployment Guide.

> **Warning**
> You must respect this DN consolidation model for your applications to handle DNs properly.

For example, to work in regular mode with an A4400 switch, the server-side application:

- Registers the ACDPosition and Extension.
- Exposes a single Dn object.

The Agent Interaction Service API presents a single DN, and transparently manages the requests to hidden ACDPosition and Extensions. The DN ID exposed in the API is the ACDPosition.
To work in substitute mode on an A4400 switch, your application must manage an Extension's activities on a place, as well as an ACDPosition's activities on that place.
When an agent logs in successfully, the event service receives two place events:

- dnRemoved carries notification that the Extension is no longer visible.
- dnAdded carries notification that a DN of type ACDPosition is now visible.

Successful agent login generates a voice media event. All subsequent voice media events and request activities occur with respect to the ACDPosition DN, not the Extension.
When the agent logs out, logout is performed through the ACDPosition DN, and upon successful logout, the event service receives four place events:

- An event carrying notification of successful logout.
- A dnRemoved event carrying notification that the ACDPosition DN is no longer visible.
- A dnAdded event with notification that the Extension is now visible.
- An event carrying notification of the Extension's status.

While no agent is logged in, the extension is visible and the ACDPosition DN is not visible. While an agent is logged in, the ACDPosition DN is visible in the place.

DN Callable Number

A DN has an identifier (or ID) and a callable number. The ID is used to access the corresponding DN's data through the services. To avoid any ambiguity and ensure unique DN IDs, a DN ID is now defined in accordance with the following rule:<DN_CME_Name>@<switch_name>
The callable number is the number that your application must use to reach a DN. Depending on the switch, the DN number declared in the configuration layer might not be the number to dial in call operations, such as make call, transfer, or conference. For example, you must remove some leading numbers to reach a Nortel DMS 100's DN.
The number to dial can also depend on the DN status. For example, if an agent is logged into the DN, you have to dial the agent ID.
Finally, because of the switch abstraction, some DNs can be hidden, but still be the real ones to dial instead of their visible counterpart. Once again, the callable number of a visible DN properly returns the hidden's one.
It is recommended to use callable numbers to access CTI features.

Using the Place Service

The place service is designed to provide your application with information about media and DNs associated with a place. The place service does not offer any direct actions on places.
The following subsections detail how to use the place service.

Getting Place DTOs

To retrieve the attribute values of the place service, use the `PlaceDTO` class of the namespace and the `getPlacesDTO()` method of the `IPlaceService` interface, as shown in the following code snippet.

```
/// Retrieving the DTOs
PlaceDTO[] myPlacesDTO = myPlaceService.getPlacesDTO(new string[]{"Place0"}, //Place ids new
string[]{ "place:medias" }); //Attributes to retrieve

/// displaying the DTO content for each Place
foreach(PlaceDTO myPlace in myPlacesDTO)
{
   // Displaying the id of the place associated with the DTO content
System.Console.WriteLine("Place : "+myPlace.placeId+"\n");

   // Displaying the DTO content
   foreach(KeyValue data in myPlace.data)
   {
       //Displaying media info only
       if(data.key == "place:medias")
       {
          MediaInfo[] myMedias = (MediaInfo[]) data.value as MediaInfo[];
          foreach(MediaInfo media in myMedias)
          {
              System.Console.WriteLine("Medium name: "+media.name + " Type: " +
media.type.ToString() + " Status:" + media.status.ToString()+"\n");
          }
       }
    }
}
```

The above code snippet retrieves the `place:medias` and `place:dns` attributes of a place and displays the content of the corresponding information objects.

PlaceEvents

The place service defines `PlaceEvent`, which is triggered when modifications on a place occur. To handle `PlaceEvent`, two important attributes are propagated:

- `place:placeId`—String identifier of the place concerned by the `PlaceEvent`.
- `place:eventReason`—Reason for this `PlaceEvent`.

The following table lists the existing `PlaceEventReasons` and the associated attributes to take into consideration.

PlaceEventReason and Associated Attributes

PlaceEventReason	Associated Attribute	Attribute Value
DN_ADDED	place:dn_added	Identifier of the added DN

PlaceEventReason	Associated Attribute	Attribute Value
DN_REMOVED	place:dn_removed	Identifier of the removed DN
MEDIA_ADDED	place:media_added	Identifier of the added media type
MEDIA_REMOVED	place:media_removed	Identifier of the removed media type

For further information on events and subscription, see The Event Service.

Using the DN Service

The DN service is specific to voice media management. A DN is an access point for phone calls identified by a callable number, which is the dn:callableNumber attribute of the IDnService interface.
The following subsections detail how to use the DN service.

Features and Possible Actions

Like many other services, the DN service can provide your application with the possible actions associated with features at a certain point in time. The com.genesyslab.ail.ws.dn.DnAction enumeration lists the DN service's actions.
The dn:actionsPossible attribute allows your application to retrieve the list of available DnAction for a DN at a certain time. DnEvent propagates this attribute when the possible DN actions are modified due to DN actions already performed.
The following table shows correspondence between DN actions and methods of the DN service.

DN Actions and Methods

DnAction	IDnService Method	Feature
SET_FORWARD	setForward()	Activate the Forward to another number.
CANCEL_FORWARD	cancelForward()	Cancel the Forward feature.
SET_DND_ON	setDNDOn()	Activate theDo Not Disturb feature.
SET_DND_OFF	setDNDOff()	Deactivate theDo Not Disturb feature.

The following subsections details how to implement the Forward and Do Not Disturb features.

Do Not Disturb (DND)

While the `Do Not Disturb` feature is activated, the DN does not accept calls.
Use the`dn:dndStatus` attribute value of the DN service to test whether the DND feature is activated:

- `DnStatus.OFF`—The DND feature is not activated.
- `DnStatus.ON`—The DND feature is activated.
- `DnStatus.UNSPECIFIED`—The switch cannot provide information.

Test the `dn:actionsPossible` attribute to determine which DND actions are possible.
The following code snippet shows how to activate and deactivate this feature for a DN identified by the string `myDnId`.

```
// Activating the Do Not Disturb feature for myDnId
myDnService.setDNDOn(myDnId,
   null, // tReasons
   null); // tExtensions
//.....
// Deactivating the Do Not Disturb feature for myDnId
myDnService.setDNDOff(myDnId, null, // tReasons
                              null); // tExtensions
```

> **Important**
>
> TExtensions and TReasons are switch-specific. Refer to your T-Server documentation for further information. See also Switch-Specific.

Forward
When the `Forward` feature is activated, the DN transfers any call received to a number that is specified when the feature is activated.
Use the `dn:forwardStatus` attribute value of the DN service to test whether the feature is activated:

- `DnStatus.OFF`—The `Forward` feature is not activated; `dn:forwardNumber` is null.
- `DnStatus.ON`—The `Forward` feature is activated; `dn:forwardNumber` contains the number receiving the forwarded calls.
- `DnStatus.UNSPECIFIED`—The switch can not provide information.

Test the `dn:actionsPossible` attribute to determine which `Forward` actions are possible.
The following code snippet shows how to activate and deactivate this feature for a DN identified by the string `myDnId`.

```
// Forwarding calls of myDnId to myDestinationNumber
myDnService.setForward(myDnId,
            myDestinationNumber,//string number receiving the transfer
            null,// tReasons
            null);// tExtensions
///....
// Cancelling the forward for myDnId
myDnService.cancelForward(myDnId,
           null, // tReasons
           null); //tExtensions
```

> **Important**
> TExtensions and TReasons are switch-specific. Refer to your T-Server documentation for further information. See also Switch-Specific.

Events of the DN Service

The DN service provides your application with two types of events, `DnEvent` and `DnUserEvent`, which are detailed below.

DnEvent

The DN service defines `DnEvent` for common events occurring on DN. To handle `PlaceEvent`, two important attributes are propagated:

- `dn:dnId`—String identifier of the DN involved.
- `dn:eventReason`—`DnEventReason` specifying the reason for the `DnEvent`.

The following table lists the existing `DnEventReason` and the associated attributes of the `IDnService` interface to take into consideration.

DnEventReason and Associated Attributes

DnEventReason	Associated Attribute	Attribute values
`STATUS_CHANGED`	`dn:status`	The current `VoiceMediaStatus` of the DN.
`INFO_CHANGED`	`dn:TEventExtensions` `dn:TEventReasons`	Switch specific-TEvent information.
`DND_ON`	`dn:dndStatus`	The DND feature has been turned on
`DND_OFF`	`dn:dndStatus`	The DND feature has been turned off.
`ON_HOOK`	`dn:hookStatus`	Telephone headset has been placed on-hook.
`OFF_HOOK`	`dn:hookStatus`	Telephone headset has been taken off-hook.
`FORWARD_SET`	`dn:forwardStatus dn:forwardNumber`	Updated status and number for the

DnEventReason	Associated Attribute	Attribute values
		Forward feature.
FORWARD_CANCELED	dn:forwardStatus dn:forwardNumber	Updated status and number for the Forward feature.

DnUserEvent

The IDnService interface provides you with a sendUserEvent() method that enables your application to directly send information to the user registered on a DN.
The DN is used as a pipe, and the user application registered on the DN receives a DnUserEvent. The propagated dn.user-data:userData attribute contains the information in a key-value array.
The following code snippet sends a DnUserEvent to the dnId0; the information sent is contained in the myUserData key-value array.

```
// Writing information in a KeyValue array
KeyValue[] myUserData = new KeyValue[1];
myUserData[0]=new KeyValue();
myUserData[0].key= "myKeyString";
myUserData[0].value= "my message to send via dnId0";
// Sending the dn user event
myDnService.sendUserEvent(dnId0,myUserData);
```

Switch-Specific

DNs depend on the underlying switch. Your application might require some switch-specific features that are detailed below.

> **Warning**
> Managing some switch-specific features implies a dependency on this particular switch.

Workmodes

Agent actions use workmodes to provide more detailed information about the agent's current state.

Workmode Types

The com.genesyslab.ail.ws.agent.WorkmodeType enumeration lists the available workmodes. Refer to the *Agent Interaction SDK 7.6 Services API Reference* to get this list. For example, the WorkmodeType.MANUAL_IN workmode indicates that the agent has to validate the action manually on the phone, and the WorkmodeType.AFTERCALLWORK workmode indicates the agent is still working on the last call.
Workmodes can be specified in forms that the agent service uses to perform some of the following

agent actions: `login`, `logout`, `ready`, and `notready`. Forms for Voice Media.

Supported Workmodes

Switches do not always provide all the workmodes defined by the `com.genesyslab.ail.ws.agent.WorkmodeType` enumeration. See Switch Information.
The DN Service offers the `dn:workmodesPossible` attribute to retrieve the list of available workmodes for a DN at a certain time. The possible workmodes in this list take into account the DN status and the switch capability. During runtime, changes in DN status may affect this list.

> **Important**
>
> Use the `dn:workmodesPossible` attribute to enable and disable workmodes, and to refresh your application when a `DnEvent` occurs.

The following code snippet shows how to read the workmode for the `dnId0` DN.

```
// Retrieving possible workmodes in a DTO for dnId0
DnDTO[] myDnDTOs = myDnService.getDnsDTO( new string[]{dnId0}, new
string[]{"dn:workmodesPossible"});

// Displaying the DTO Content
foreach(DnDTO dto in myDnDTOs)
{
   System.Console.WriteLine(" Possible workmodes for "+dto.dnId);
   // Reading attributes retrieved in the DTO
   foreach(KeyValue pair in dto.data)
   {
      // Displaying the possible workmodes
      if(pair.key == "dn:workmodesPossible")
      {
         WorkmodeType[] myPossibleWorkmodes = (WorkmodeType[]) pair.value as WorkmodeType [];
         foreach(WorkmodeType type in myPossibleWorkmodes)
         {
            System.Console.WriteLine((type.ToString());
         }
      }
   }
}
```

T-Extensions and T-Reasons

T-Extensions and T-Reasons are switch-specific key-value pairs that you can use to fine-tune your application. Your application can pass T-Extensions and T-Reasons in arguments to method calls or get them from T-Server Events (if any).

Methods Call

The server-side application maps agent and DN services' methods that take T-Extensions and T-Reasons in parameters with the T-Library. For T-Library users, the mapping of the T-Library features is straightforward, because the naming convention is similar to T-Library functions.
To determine which T-Extensions and T-Reasons to use, refer to your T-Server documentation.

T-Server Events

The server-side application can propagate T-Events with `DnEvent`. T-Extensions and T-Reasons can be optional in T-Events, so the server-side application copies them in the published `dn:TEventExtensions` and `dn:TEventReasons` attributes of the DN service only if they exist. Refer to your T-Server documentation for further information.

Switch Information

The `dn:switchInfo` attribute of the `IDnService` interface lets your application retrieve a `SwitchInfo` object.
The `SwitchInfo` class provides information about the switch associated with a DN:

- `SwitchInfo.name`—The identifier of the switch associated with this DN.
- `SwitchInfo.switchType`—The switch type.
- `SwitchInfo.workmodeCapables`—Array of workmode types that this switch supports.
- `SwitchInfo.actionCapables`—Array of `InteractionVoiceAction` that this switch supports.

> **Important**
> InteractionVoiceActions are actions of the voice interaction service applied to phone calls. See Voice Interactions.

The `XxxCapables` attributes list the switch's supported features. These feature lists do not change during runtime. For example, if a workmode does not appear in the `SwitchInfo.actionCapables` attribute, the workmode is not supported. Therefore, this workmode will never appear in the list of possible workmodes and should not be visible for the agent using the DN.

> **Important**
> The list of DN's possible workmodes (`dn:workmodesPossible`) takes into consideration the current DN's status and the DN's capable workmodes.

The Interaction Service

The interaction service is defined by the `IInteractionService` interface defined in the`com.genesyslab.ail.ws.interaction` namespace. Its features are designed to manage characteristics common to all interactions, no matter their media.

Introduction

This section introduces the interaction concepts and the interaction service covered in this chapter.

What Is an Interaction?

An interaction is the representation of the communication between two persons or more in the Genesys world. It can represent, for example, a phone call between an agent and a customer, an agent and an incoming e-mail, or a chat session between agents and a customer.
An interaction exists:

- From the moment that a contact gets in touch with the agent (a call or an e-mail is ringing on the agent desktop) or from the moment that the agent initiates getting in touch with a contact or another agent.
- Till the interaction is done: the agent hangs up (or closes the e-mail) and marks the interaction as done.

Interactions are associated with a particular medium:

- Voice interactions (or phone calls) are associated with DNs.
- E-mail (or mail) interactions are associated with the `EMAIL` media.
- Chat interactions are associated with the `CHAT` media.

What Is the Interaction Service?

Although the underlying media are different, interactions share some common characteristics and similar data management.
The interaction service is an interface designed to read and write DTOs containing:

- Common attributes defined in the `IInteractionService` interface.
- Interaction-specific attributes defined in other interactions interfaces such as `IInteractionVoiceService` and `IInteractionMailService` .

The `IInteractionService` interface provides dedicated DTO methods using DTO classes of the `com.genesyslab.ail.ws.interaction` namespace. See also Data Transfer Object for details.
The following are examples of common interaction attributes defined in the interaction service:

- `interaction:interactionId`—the system interaction identifier.
- `interaction:contactId`—the contact identifier.

- `interaction:status`—the interaction status defined with the `InteractionStatus` enumeration.
- `interaction:interactionType`—the interaction type defined with the `InteractionType` enumeration.
- `interaction:eventReason`—the `InteractionEventReason` value published with interaction events.
- `interaction:outboundChainId`—if an interaction arrives, and it is in an outbound context, this attribute is filled with the corresponding outbound chain ID. See The Outbound Service for details.

The interaction service does not let you perform any actions on an interaction, such as making a call or sending an e-mail. You may have noticed, that there is no existing interaction action enumeration in the `com.genesyslab.ail.ws.interaction` namespace. However, the interaction service has methods for adding, modifying, and deleting attached data of any existing interaction (see Attached Data).
The interaction service also offers `InteractionEvent`, that are generic interaction events used by specialized interaction services such as `IInteractionVoiceService` and `IInteractionMailService`.

Specific Interaction Services

The Agent Interaction Service API includes other interaction services and their associated namespaces:

- `IInteractionVoiceService` (`com.genesyslab.ail.ws.interaction.voice`)—this service is dedicated to voice interactions occurring on the DNs of a Place. This service requests actions on voice interactions.
- `IInteractionMailService` (`com.genesyslab.ail.ws.interaction.mail`)—this service is dedicated to e-mail interactions occurring on the `EMAIL` media of a Place. This service requests actions on e-mail interactions.
- `IInteractionChatService` (`com.genesyslab.ail.ws.interaction.chat`)—this service is dedicated to chat interactions occurring on the `CHAT` media of a Place. This service requests actions on chat interactions.

These services deal with specialized interaction actions, but to properly use them, your application must rely on the interaction service for interaction data handling and integrate `InteractionEvent` events. The following sections present details.

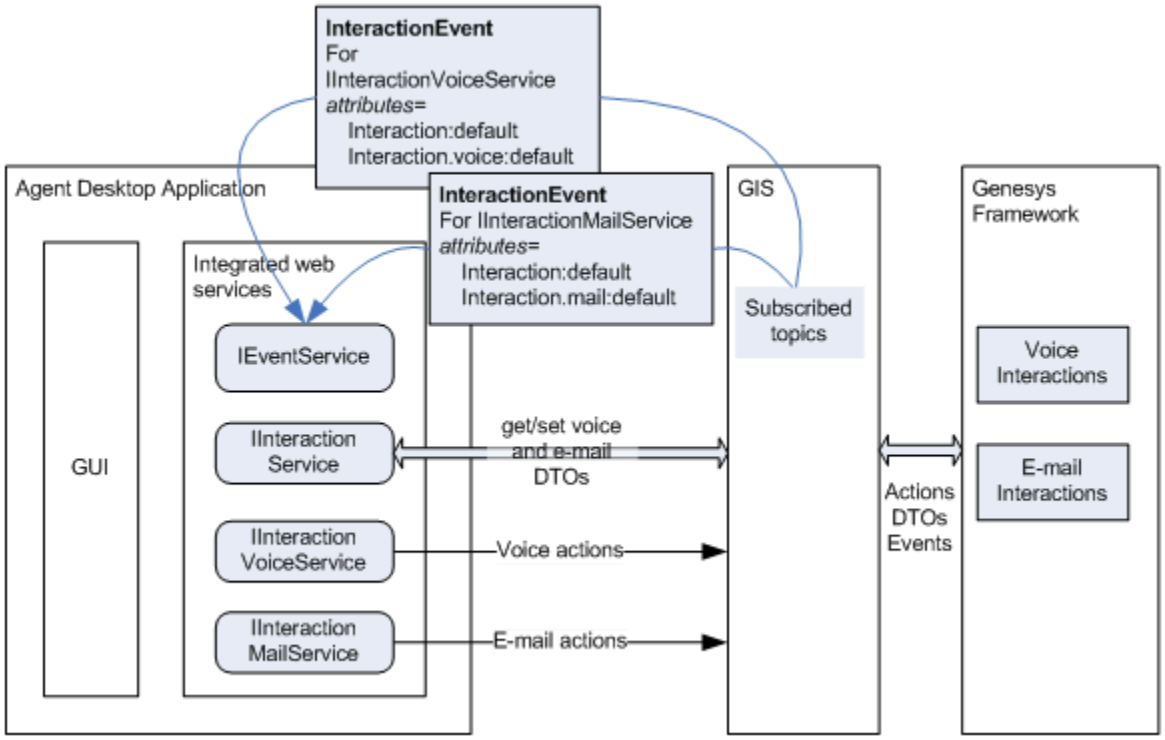
Using IInteractionService

The `IInteractionService` interface is used to manage interactions' data that correspond to attributes defined in the other interactions services interfaces. Your application is likely to use it any time it needs to access and modify interactions' data.
Moreover, the `IInteractionService`interface offers `InteractionEvent` which are generic events publishing the following attributes having theevent property:

- `interaction:*`—`IInteractionService` attributes
- `interaction.*:*`—attributes for e-mail, chat, and voice interaction services.

`InteractionEvent` events are the most common events received by your application after the media-specialized interaction services have successfully performed actions.
To properly deal with interactions and specialized services, your application has to subscribe to these

events. The following figure illustrates relationships between services, interactions, and `InteractionEvent` received by the `IEventService`.



Interactions and Services

This figure shows that the `IInteractionService` interface manages all data transactions and that specialized services—e-mail and voice services—are used to perform dedicated actions on the corresponding interactions.
The `IEventService` has subscribed to `InteractionEvent` with `TopicsEvent` having a trigger on the agent identifier (or place identifier). The propagated attributes include some `IInteractionService`, `IInteractionVoiceService`, and `IInteractionMailService` attributes.
When an `InteractionEvent` occurs:

- If the `Event` object retrieved is related to a voice interaction, it propagates `IInteractionService` and `IInteractionVoiceService` attributes.
- If the `Event` object retrieved is related to an e-mail interaction, it propagates `IInteractionService` and `IInteractionMailService` attributes.

> **Important**
>
> The media-specialized services interfaces may offer other specific events. For further information, see the Agent Interaction SDK 7.6 Services API Reference.

Handling Interaction DTOs

The `IInteractionService` interface has two general methods to deal with DTOs:

- `IInteractionService.getInteractionsDTO()`—This method retrieves the DTOs of a set of interactions for a set of attributes having the `read` property.
- `IInteractionService.setInteractionsDTO()`—This method uses DTOs to set the values of a set of interactions for a set of attributes having the `write` property.

> **Important**
> These methods offer standard DTO handling. See Data Transfer Object

After your application gets an instance of the `IInteractionService` interface, it can use the above methods or specific DTO methods detailed in the following subsections.

Specific Getting DTO Methods

The `IInteractionService` interface retrieves attributes of any interaction type with interactions DTOs according to their associated agents, place identifiers, or DN identifiers.

- The `IInteractionService.getInteractionsDTOFromAgent()` method retrieves DTOs for voice, chat, or e-mail interactions associated with agents.
- The `IInteractionService.getInteractionsDTOFromPlace()` method retrieves DTOs for voice, chat, or e-mail interactions associated with a place if an agent is logged on this place.
- The `IInteractionService.getInteractionsDTOFromDN()` method retrieves DTOs for voice, chat, or e-mail interactions associated with a DN if an agent is logged on this DN.

> **Important**
> You can also use the IInteractionService.getInteractionsDTO() method to retrieve the DTOs of a set of interactions.

The following code snippet shows how to use `IInteractionService.getInteractionsDTOFromAgent()` method to retrieve the interaction identifiers and status available for `Agent0`.

```
InteractionAgentDTO[] ixnAgentDTOs = myInteractionService.getInteractionsDTOFromAgent( new
string[]{"agent0"},
new string[]{ "interaction:interactionId", "interaction:status"});

foreach(InteractionAgentDTO agentDTO in ixnAgentDTOs)
{
   System.Console.WriteLine("Agent: "+ agentDTO.agentId+"\n");
   foreach(InteractionDTO dto in agentDTO.interactionsDTO)
   {
```

```
      System.Console.WriteLine(dto.interactionId+" ");
      foreach(KeyValue attributes in dto.data)
      {
         System.Console.WriteLine(attributes.key+": " +attributes.value.ToString()+"\n");
      }
   }
}
```

Opening a Workbin Interaction

The `IInteractionService` interface has two general methods to opens a workbin interaction:

- `IInteractionService.openInteractionForAgentDTO()`—This method opens a workbin interaction for an agent and a set of attributes.
- `IInteractionService.openInteractionForPlaceDTO()`—This method opens a workbin interaction for a place and a set of attributes. Once the interaction is opened, it goes onto the specified place for treatment.

The following code snippet shows how to use `IInteractionService.openInteractionForAgentDTO()` method to open a workbin interaction for `Agent0`.

```
InteractionDTO InteractionAttributes =
   myInteractionService.openInteractionForAgentDTO(
      "agent0", // the agent Id
      myInteractionId, // the interaction ID
      new string[]{
      "interaction:interactionId",
      "interaction:status"
   } // the set of the interaction's attributes
);
```

Attached Data

Attached Data—also known as User Data—is a set of key-value pairs attached to an interaction and stored with the interaction in the contact history. Attached Data can be collected by the Genesys Solution and modified by the agent.
An attached data can be any data useful to your application's design. However, it can also include the following specific attached data:

- Interaction attribute values.
- Custom attached data's values.

The Configuration Layer defines keys and information for these attached data, available through the `IResourceService` interface. See Interaction Information for further details.

Attached Data DTOs

Attached Data are handled with the following attributes of the `IInteractionService` interface:

- `interaction:attachedData`—This attribute is used to read or write the attached data of the interaction.

> **Important**
> Writing with these DTOs replaces the previous attached data by the ones written.

- `interaction:addAttachedData`—This attribute is used to write the added attached data in the interaction.
- `interaction:removeAttachedData`—This attribute is used to remove all the attached data of a set of interactions.

The `com.genesyslab.ail.ws.interaction` namespace includes the `AttachedData` container class to deal with DTOs. An `AttachedData` object represents a single key-value pair attached to an interaction.
Use the `IInteractionService.setInteractionsDTO()` to write DTOs and the `IInteractionService.getInteractionsDTO*()` to read them.
In the following code snippet, a new attached data is added to the `Interaction0` interaction:

```
/// Creating a AttachedData
AttachedData myData = new AttachedData();
myData.key = "Key0";
myData.value = "This is the value of Key0";

/// Creating an array of InteractionDTO
InteractionDTO[] ixnDTOs = new InteractionDTO[1];

/// Filling the first DTO
ixnDTOs[0] = new InteractionDTO();

/// Specifying the concerned Interaction
ixnDTOs[0].interactionId = “Interaction0”;

/// Creating an array of DTOs
ixnDTOs[0].data = new KeyValue[1];
ixnDTOs[0].data[0] = new KeyValue();

/// Giving the DTO content
ixnDTOs[0].data[0].key = "interaction:addAttachedData";
ixnDTOs[0].data[0].value = new AttachedData[]{ myData };

/// Writing the DTOs with the IInteractionService
myInteractionService.setInteractionsDTO(ixnDTOs);
```

Attached Data and Event

When the attached data of an interaction are modified, an `InteractionEvent` is sent with an`InteractionEventReason.INFO_CHANGED` reason for the `interaction:eventReason` attribute. The modified attached data are available in the `interaction:modifiedAttachedData` attribute.

Voice Interactions

The voice interaction service is the `IInteractionVoiceService` interface defined in the `com.genesyslab.ail.ws.interaction.voice` namespace. To use this service, your application works with classes and enumerations of this namespace, and with the classes and interface of the `com.genesyslab.ail.ws.interaction` namespace.

Introduction

The voice interaction service is designed around a set of actions for managing voice interactions. Voice interactions are specific objects representing telephone calls, that is, interactions using the voice media (DNs). Voice interaction features may vary, one from another, depending on the capabilities of the underlying switch.
The voice interaction service manages the following tasks:

- Creating and dialing a call.
- Answering a call.
- Transferring a call.
- Holding and retrieving a call.
- Beginning a conference.
- Activating or deactivating the mute function during a call.

The voice interaction service only performs actions on voice interactions. The voice interaction service depends on the following other services:

- The event service to subscribe and receive events.
- The agent service.
  - Before working with voice interactions, your application first must use the agent service to log in an agent on a DN.
  - While the agent is logged on a DN, your application can use the `IInteractionVoiceService` interface to perform actions on voice interactions associated with the DN.

> **Tip**
> See the Agent Service for further details.

- The interaction service:
  - To access `IInteractionVoiceService` attributes, your application uses the `IInteractionService` DTO methods.

- There is no event associated with the `IInteractionVoiceService` interface: its attributes are published in events of type `InteractionEvent` which is described in the `IInteractionService` interface.

> **Important**
> See The Interaction Service.

Voice Interaction Essentials

The `IInteractionVoiceService` interface exposes methods and pertinent attributes to let your application manage multiple simultaneous voice interactions, each identified by a unique interaction ID.

Each voice interaction follows a sequence of states, for example, beginning in a `NEW` state, transitioning to a `DIALING` state, through other states until its state is finally `MARKED_DONE`.

For any particular state, the voice interaction service permits use of only a small subset of its possible actions (available to your application as method calls). For any one voice interaction, your application may apply only one action at the same time.

After the voice interaction service successfully applies an action with respect to a particular voice interaction, the event service receives an `InteractionEvent` that carries the `interactionId` identifying the corresponding voice interaction along with a variety of attributes reflecting new state and other data. To receive `InteractionEvent` events, your application must subscribe to them.

For each incoming `InteractionEvent` event, your application should test various attributes of the `IInteractionVoiceService` interface, including especially the`interaction.voice:actionsPossible` attribute (to determine which actions have become possible to apply).

The following sections present the details behind the above general description.

Voice Attributes

For each voice interaction, the `IInteractionVoiceService` interface defines a set of attributes which are characteristic of phone call data. The following list is representative (but not exhaustive):

- `interaction.voice:ANI`—The Automatic Number Identification parameter associated with a voice interaction.
- `interaction.voice:DNIS`—The Dialed Number Identification Service parameter associated with a voice interaction.
- `interaction.voice:phoneNumber`—The phone number(s) to which this interaction has been connected.
- `interaction.voice:duration`—The call duration in seconds.

For each voice interaction, the attributes of the `IInteractionService` interface are also available. Your application may often use the following:

- `interaction:interactionId`—The system interaction identifier, required in calls to the methods of the voice interaction service.
- `interaction:status`—The interaction status defined with the `InteractionStatus` enumeration.
- `interaction:eventReason`—The `InteractionEventReason` value published when an `InteractionEvent` event occurs for a voice interaction.

The `IInteractionVoiceService` has no methods to read these attributes. Your application must call one of the `IInteractionService.getInteractionDTO*()` methods. See also Handling Interaction DTOs.

See the *Agent Interaction SDK 7.6 Services API Reference* for further details.

Voice Actions

The `InteractionVoiceAction` enumeration defines voice actions of the `IInteractionVoiceService` interface. Constants each correspond to one voice interaction service method. For example, the `ANSWER_CALL` constant corresponds to the `IInteractionVoiceService.answerCall()` method. Each method call performs an action on one voice interaction. The method call does not apply do a set of interactions.

> ### Warning
> It is recommended to use callable numbers to access CTI features. See DN Callable Number.

Possible actions for voice interactions can be accessed by reading the value of the `interaction.voice:actionsPossible` attribute of the `IInteractionVoiceService`.

The `IInteractionVoiceService` has no methods to read attributes. Your application must use one of the `InteractionService.getInteractionsDTO*()` methods. See also Handling Interaction DTOs.

Voice Interaction Status

The current state of a voice interaction is available as the value of the `interaction:status` attribute, which is defined in the `IInteractionService`. For a voice interaction, possible status values are listed in the `InteractionStatus` enumeration of the `com.genesyslab.ws.interaction` namespace.
The `IInteractionVoiceService` has no methods to read attributes. Your application must use one of the `InteractionService.getInteractionsDTO*()` methods. See also Handling Interaction DTOs.

Status Change

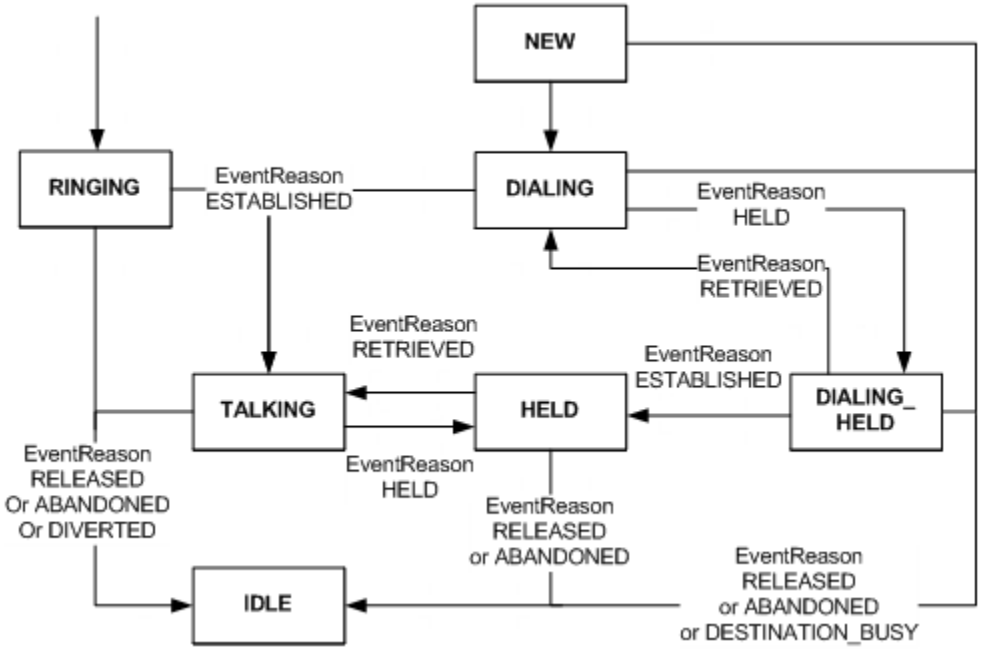
The status of a voice interaction changes if:

- A successful action is confirmed by an event sent to the server-side application; for example, the held

action has been performed on the call so the voice interaction status changes to `InteractionStatus.HELD`.

- A CTI event changed it; for example, if a call is no longer dialing but now ringing, the voice interaction status changes to `InteractionStatus.RINGING`.

The following example diagram shows some transitions between interaction voice statuses.



Generalized Example of a Voice State Diagram (Incomplete)

> ## Warning
> This figure is provided as an informative example. It does not include all possible statuses and transitions.

Switch-Specific

Some statuses are switch-specific and not reachable for those switches for which the feature associated with this status is not available.
For example, if the held feature is not available on a particular switch, the`InteractionVoiceAction.HELD` action is not available impacting on `InteractionStatus.HELD` and `InteractionStatus.DIALING_HELD` status:

- If `InteractionVoiceAction.HELD` is unavailable, the `InteractionStatus.HELD` and `InteractionStatus.DIALING_HELD` status are not reachable.
- Some switches have the`InteractionStatus.HELD` feature but do not allow its use during the call dialing, in which case the`InteractionStatus.DIALING_HELD` status is not reachable.

As the possible statuses, transitions and event workflow differ from one switch to another.

> **Warning**
>
> Do not assume any particular transition sequence. You should base your application design on the status and not on the associated `InteractionEventReason` of the `InteractionEvent` received.

Voice Events

The `IInteractionVoiceService` has no defined events. All events occurring on a voice interact ion are propagated in `InteractionEvent` only. Published attributes are `IInteractionService` attributes as well as those `IInteractionVoiceService` attributes that have the `event` property.

Your application must subscribe to a `TopicsService` defined for the `IInteractionService`. This `TopicsService` has to specify in its `TopicsEvents` the particular voice interaction DTOs to retrieve. Moreover, they must include a trigger on the agent or on the place where the agent is logged in.

> **Important**
>
> For further details on the `InteractionEvent` mechanism, see Using IInteractionService.

The following code snippet shows how to receive `InteractionEvent` occurring on any interaction belonging to `agent0` and how to be sure to have the voice interaction DTOs propagated for a voice interaction event.

```
TopicsService[] mytopicsServices = new TopicsService[1] ;

/// Defining a Topic Services
mytopicsServices[0] = new TopicsService() ;
mytopicsServices[0].serviceName = "InteractionService" ;

TopicsEvent[] mytopicsEvents = new TopicsEvent[1] ;
mytopicsEvents[0] = new TopicsEvent() ;

/// the targeted events are InteractionEvent
mytopicsEvents[0].eventName = "InteractionEvent" ;
mytopicsEvents[0].attributes = new String[]{ // interaction commons attributes to retrieve
"interaction:interactionID", "interaction:status",
"interaction:interactionType", // the entire set of voice interaction attributes is retrieved
"interaction.voice:*"};

/// The InteractionEvent has to concern agent0 interactions
mytopicsEvents[0].triggers = new Topic[1];
mytopicsEvents[0].triggers[0] = new Topic();
mytopicsEvents[0].triggers[0].key = "AGENT";
mytopicsEvents[0].triggers[0].value = “agent0”;


/// ... Subscribe to the event service
```

See also The Event Service.

TEvent DTOs

An `InteractionEvent` event received for a voice interaction may correspond to an underlying TEvent coming from the T-Server side. The specific content of these TEvents may be copied in the `InteractionEvent`.
In this case you can access to T-Event specific attributes (if there are some):

- `interaction.voice:TEventExtensions`: KeyValue pairs for TExtensions.
- `interaction.voice:TEventReasons`: the reason for this TEvent.

> **Important**
> TEvents attributes are switch-specific. For further information on TEvent, refer to your T-Server documentation.

Making and Answering Voice Calls

The most commonly used actions of the `IInteractionVoiceService` are presented in the following table.

Standard Features for a Voice Interaction

Action	InteractionVoiceAction	IInteractionVoiceService
Make a call	`MAKE_CALL`	`makeCall()`
Answer a call	`ANSWER_CALL`	`answerCall()`
Release a call	`RELEASE_CALL`	`releaseCall()`
Mark done a call	`MARK_DONE`	`markDone()`

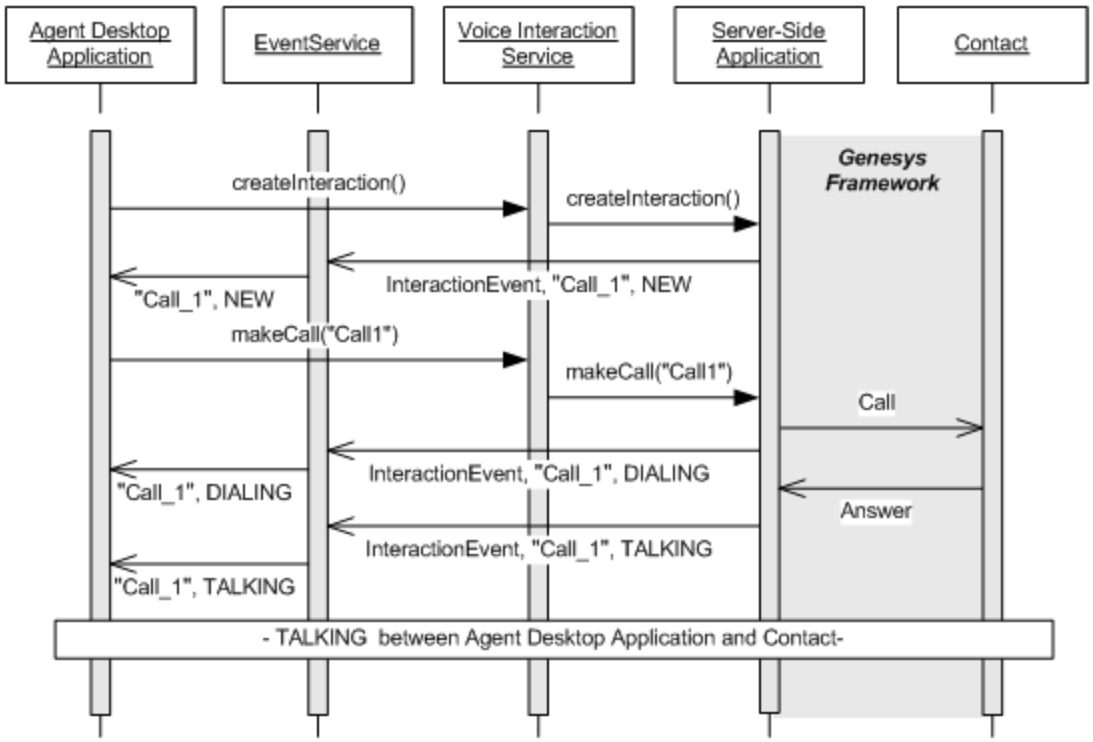
Your application uses the `IInteractionVoiceService` features to send requests to the server-side application to take some actions. After the server-side application (working with the Genesys framework and possibly other Genesys solutions) has performed a corresponding action, the event

service receives `InteractionEvents` if it has subscribed with the correct topics.
An `InteractionEvent` event propagates status changes for a voice interaction, identified by the `interaction:interactionId` attribute. Your application should test the `interaction.voice:actionsPossible` attribute to determine which actions are currently possible to use on this interaction.
The following sections detail commonly used voice interaction actions along with sequence diagrams showing action sequences.

> **Warning**
> Do not use the sequence diagrams to make assumptions of possible actions. Your application always should always use the `InteractionVoiceAction` possible actions provided in DTOs.

Making a Call

To make a call, your application needs to create a new voice interaction in order to dial the call. The following sequence diagram presents the creation of a new voice interaction and how to dial a call.



Making a Call

All the `InteractionEvent` events received by the event service have a `STATUS_CHANGED` reason, and their labels indicate the current status of the associated voice interaction.

Creating a Voice Interaction

`IInteractionVoiceService` has two available methods for creating a voice interaction:

- `createInteractionFromPlaceDTO()`—creates a voice interaction with a place identifier if:
  - An agent is logged on the place.
  - The place has an available DN to create the interaction.
- `createInteractionFromDnDTO()`—creates a voice interaction with a DN identifier if an agent is logged on the DN.

> **Important**
>
> To test if a voice interaction can be created for an agent on a DN, use the `agent:dnsActionsPossible` attribute of the IAgentService interface to determine if the `AgentDnAction.CREATE_INTERACTION` action is possible at this time for this

> DN.

The following code snippet shows a voice interaction created from a DN:

```
// Setting attributes to retrieve in InteractionVoiceErrorDTO
String[] atts = new String[]{ "interaction:agentId", //the id of the agent owning //the
interaction
"interaction:status"}; // the status of the interaction //once created

// Creating the interaction
InteractionVoiceErrorDTO myVoiceErr = myInteractionVoiceService.createInteractionFromDnDTO(
myDNID, // DN identifier
  atts, // attributes from the creation to retrieve in a DTO
  null, // destination number
  null, // T-Server to use
  null, // call type
  null, // attached data
  null, // reasons defined by the user
 null); // tExtensions
```

> **Important**
>
> If you specify the destination number of a call in the `IInteractionVoiceService.createInteractionFrom*()` method parameters, the call is dialed as soon the voice interaction is created. If you do not specify a destination number, the method does not take into account the following parameters: location, call type, attached data, reasons, and T-Extensions.

The `createInteractionFromDnDTO()` method returns an `InteractionVoiceErrorDTO` object containing:

- If the interaction is created, the `InteractionVoiceErrorDTO.interactionDTO` field contains:
  - The interaction ID.
  - The values of the attributes specified in the method call.
- If the interaction creation failed, the corresponding error is available in the `InteractionVoiceErrorDTO.voiceError` field.

For example, you can choose to retrieve the `interaction:agentId` attribute at the time of interaction creation. The following code snippet displays the content of the retrieved `InteractionVoiceErrorDTO` object:

```
string myInteractionId=null;
// If an error occurred, displaying the errors
if(myVoiceErr.voiceError!=null)
{
   // The creation has failed
   System.Console.WriteLine("Creation failed.\n");
   // Displaying the telephony error type
   System.Console.WriteLine("Telephony Error: "
```

```
+myVoiceErr.voiceError.telephonyErrorType.ToString() +"\n");
   // Displaying the corresponding TServer error
   System.Console.WriteLine("TServer Error: " +myVoiceErr.voiceError.TServerError +"\n");
}
else
{
   // The creation is successful
   System.Console.WriteLine("Creation OK.\n");
   // Displaying the interaction id
   myInteractionId = myVoiceErr.interactionDTO.interactionId;
   System.Console.WriteLine("ID of the created interaction: " + myInteractionId+"\n");
}
```

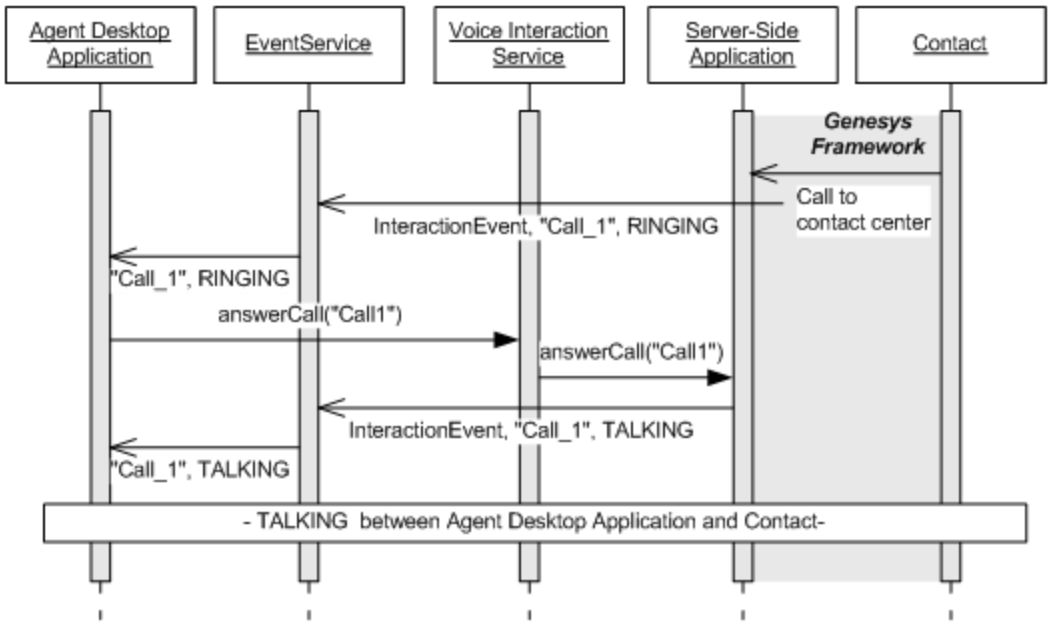
Dialing a Call

To dial a call, use the `IInteractionVoiceService.makeCall()` method and specify the proper voice interaction identifier (`interaction:interactionId`) as shown in the following snippet.

```
// There is no attribute to specify before the method call
VoiceError err = myInteractionVoiceService.makeCall( myInteractionId, // voice interaction
making the call destination, // phone number to call null, // T-Server to use
MakeCallType.REGULAR, null, // attached data null, // reasons defined by the use null); //
tExtensions

// If an error occurred, displaying the error
if(err!=null)
{
   // The creation has failed
   System.Console.WriteLine("Dial failed.\n");
   // Displaying the telephony error type
   System.Console.WriteLine("Telephony Error: " +err.telephonyErrorType.ToString() +"\n");
   // Displaying the corresponding TServer error
   System.Console.WriteLine("TServer Error: " +err.TServerError +"\n");
}
```

Answering a Call

Your application is likely to receive phone calls from external or internal contacts. In such cases, your application receives an `InteractionEvent` associated with a voice interaction that has a `InteractionStatus.RINGING` status as shown in the following diagram.



Answering a Call

If the `interaction.voice:actionsPossible` attribute available through that event includes `InteractionVoiceAction.ANSWER_CALL` as one of its values, your application can call the`IInteractionService.answerCall()` method as presented in the following code snippet.

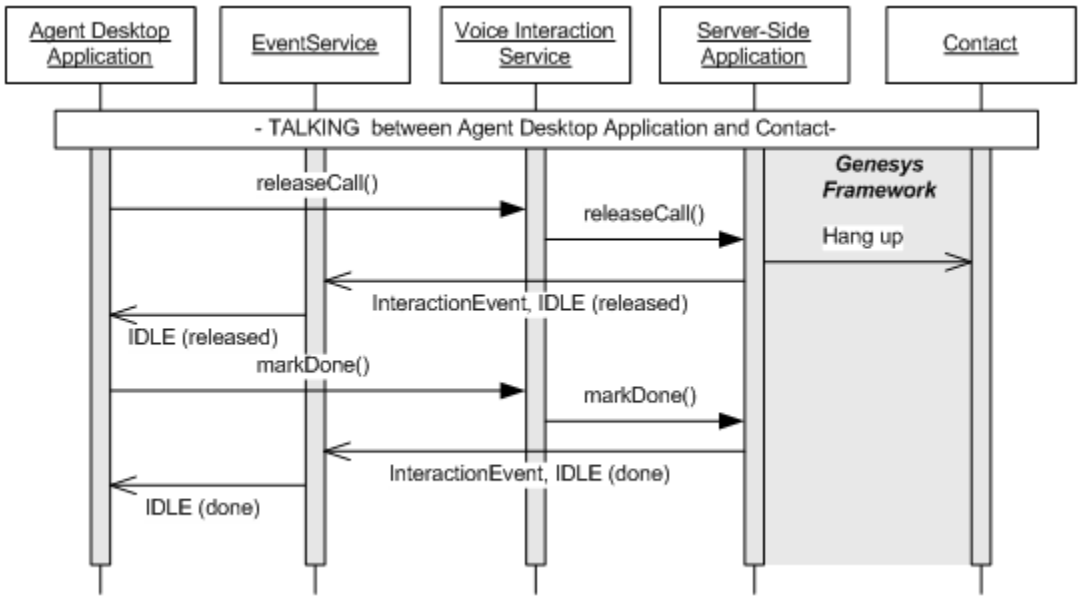
```
// There is no attribute to specify before the method call
VoiceError err = myInteractionVoiceService.answerCall( myInteractionId, // voice interaction
making the call
   null, // reasons defined by the user
   null); // tExtensions

// If an error occurred, displaying the error
if(err!=null)
{
   // The creation has failed
   System.Console.WriteLine("Answer failed.\n");
   // Displaying the telephony error type
   System.Console.WriteLine("Telephony Error: " +err.telephonyErrorType.ToString() +"\n");
   // Displaying the corresponding TServer error
   System.Console.WriteLine("TServer Error: " +err.TServerError +"\n");
}
```

Terminating a Call

To terminate a call, your application must request two actions:

- Release the call—The agent releases or the contact hangs up.
- Mark the call done—The interaction is saved in the contact history and becomes inactive. The `IInteractionService` and `IInteractionVoiceService` can no longer access this interaction.

The following diagram shows the actions and `InteractionEvent` sequence when the agent desktop application hangs up.



Releasing and Marking Done a Call

If it is the contact that hangs up, the agent desktop application does not have to release.

Releasing the Call

To release the voice interaction, use the `IInteractionVoiceService.` releaseCall() method and specify the corresponding voice interaction identifier (`interaction:interactionId`) as shown in the following code snippet.

```
VoiceError err = myInteractionVoiceService.releaseCall( myInteractionId, // voice interaction
to release null, // reasons defined by the use null); // tExtensions

// If an error occured, displaying the errors
if(err!=null)
{
   // The creation has failed
   System.Console.WriteLine("Release failed.\n");
   // Displaying the telephony error type
   System.Console.WriteLine("Telephony Error: " +err.telephonyErrorType.ToString() +"\n");
   // Displaying the corresponding TServer error
   System.Console.WriteLine("TServer Error: " +err.TServerError +"\n");
}
```

Marking Done the Call

To mark a voice interaction as done, use the `IInteractionVoiceService.` markDone() method and

specify the appropriate voice interaction identifier (`interaction:interactionId`) as shown in the following code snippet.

```
// voice interaction to mark done
myInteractionVoiceService.markDone(myInteractionId);
```

Transferring Voice Calls

There are three types of transfer for a voice interaction:

- Single-step transfer—This transfers the call directly to the agent who takes it over. If this type of Transfer is not available, a mute transfer is done instead.
- Dual-step transfer—The agent can contact with the Agent receiving the call before completing it.
- Mute transfer—The transfer is initialized then automatically completed.

These transfers may not be available, depending on the switch in charge of the interaction. Design your application to test the possible actions provided in the `interaction.voice:actionsPossible` attribute to determine which transfers are available.

Single-Step and Mute Transfers

The single-step or mute transfers are direct transfers (shown in Direct Transfers) of a voice interaction, and can be performed in one method call.

Direct Transfers

Type of Transfer	InteractionVoiceAction	IInteractionVoiceService
Single-Step	`SINGLE_STEP_TRANSFER`	`singleStepTransfer()`
Mute	`MUTE_TRANSFER`	`muteTransfer()`

These transfers are performed in a single method call as shown in the following code snippet:

```
VoiceError err = myInteractionVoiceService.singleStepTransfer( string interactionId,
   myDNIdReceivingTheTransfer,
   null, // T-Server to use
   "the agent reasons for this transfer",
   null, //AttachData
   null, // reasons defined by the user
   null); // tExtensions
```

If the transfer—either mute or single-step—succeeds:

- The voice interaction is released and your event service receives an `InteractionEvent` with `InteractionStatus.IDLE`.
- The targeted agent application receives a `InteractionEvent` for a voice interaction with `InteractionStatus.RINGING`. This voice interaction is the transferred call.

The Dual-Step Transfer

The dual-step transfer, described in Steps of the Dual-Step Transfer is performed in two `IInteractionVoiceService` actions:

- The first agent initiates the transfer by calling the second agent to whom the call is to be transferred.
- The first agent completes the transfer, its voice interaction is released, and the call is transferred to the second agent, who is able to talk with the contact.

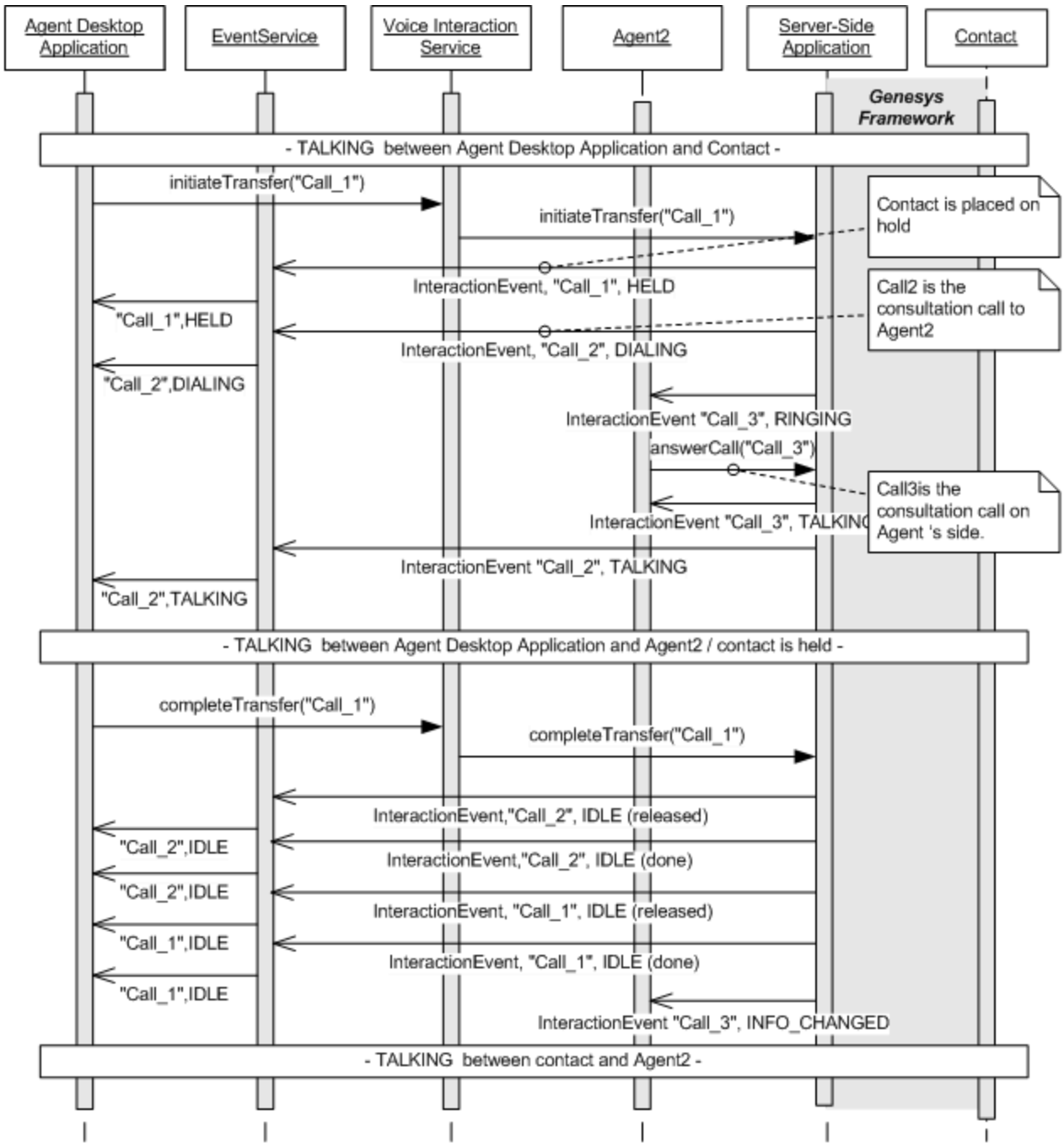
Steps of the Dual-Step Transfer

Steps	InteractionVoiceAction	IInteractionVoiceService
Initiate	`INIT_TRANSFER`	initiateTransfer()
Complete	`COMPLETE_TRANSFER`	completeTransfer()

> **Important**
>
> Steps of the Dual-Step TransferStepsInteractionVoiceActionIInteractionVoiceServiceInitiateINIT_TRANSFERiniti test if the dual-step transfer is available, the InteractionVoiceAction.INIT_TRANSFER has to be available in the interaction.voice:actionsPossible DTO of the voice interaction.

Dual-Step InteractionEvent Sequence

The following figure shows the sequence diagram of method calls and `InteractionEvent` events received during the dual-step transfer.



A Transfer Initiated by the Agent Desktop Application

On Agent1's Side

If `Agent1` initiates the transfer, `Agent1` deals with two interactions to perform the transfer:

- Call_1 is the original interaction to be transferred existing betweenAgent1 and the contact.
- Call_2 is the voice interaction existing between Agent1 and Agent2.

When Agent1 initiates the transfer, Call_1 is held and Call_2 is created so as to enter Phase 1—Agent1 calling Agent2. The two interactions are released whenAgent1 completes the transfer.

> **Important**
>
> To determine if your application can complete the transfer, test if the InteractionVoiceAction.COMPLETE_TRANSFER action is available in the interaction.voice:actionsPossible attribute.

On Agent2's Side

Call_3 is the interaction originally created between Agent2 and Agent1: this interaction receives the transfer once it is completed byAgent1. As a result, Agent2 application receives an InteractionEvent for Call_3:

- The interaction:eventReason attribute is set to InteractionEventReason.INFO_CHANGED.
- The interaction.voice:parties value has changed.
- The interaction:extensions DTO contain extended information about the new party added. See the *Agent Interaction SDK 7.6 Services API Reference* for further information.

Implementation Example

The following code snippet shows the method implementation for dual-step transfer corresponding to the sequence diagram *A Transfer Initiated by the Agent Desktop Application*.

```
/// Agent1 initiates the transfer of Call_1 for Agent2
InteractionVoiceErrorDTO err0 = myInteractionVoiceService.initiateTransferDTO( "Call_1",
   "DN_Agent2", // Dn of agent2 who receives the transfer
   null, // T-Server to use "the agent reasons for this transfer",
   null, // attached data
   null, // reasons defined by the user
   null, // tExtensions
   null); // attributes
//...
/// Agent1 and Agent2 are talking
VoiceError err1 = myInteractionVoiceService.completeTransfer( "Call_1",
    null, // reasons defined by the user
    null); // tExtensions
```

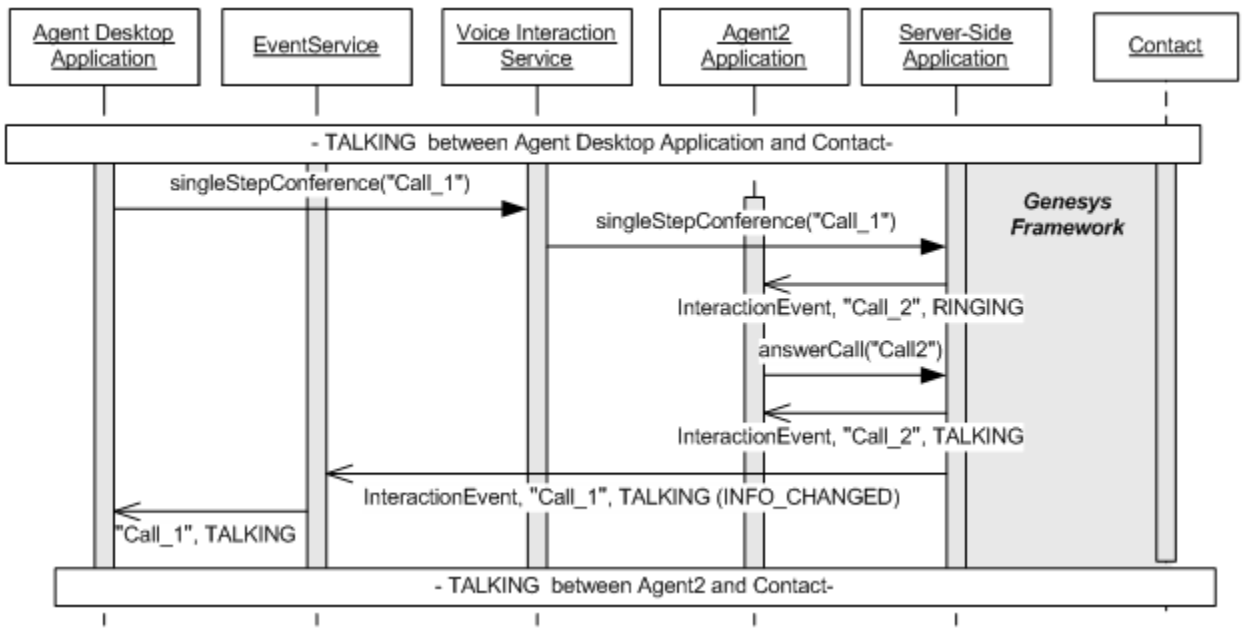
Managing Conference Calls

There are two types of conference voice calls:

- The single-step conference—A single method call performs a conference.
- The dual-step conference—As with dual-step transfers, one agent first talks to the agent who is going to join the conference, then the first agent completes the conference.

Single-Step Conference

The single-step conference is available only if the `InteractionVoiceAction.SINGLE_STEP_CONFERENCE` is available in the `interaction.voice:actionsPossible` attributes corresponding to the voice interaction.
The following diagram shows the sequence of method calls and received `InteractionEvent` events for the single-step conference.



Single-Step Conference Sequence

The single-step conference is easy to deal with: the second agent receives a ringing voice interaction. If the second agent answers the call, the conference is established.
Your application receives an `InteractionEvent` event:

- The `interaction:eventReason` attribute is set to `InteractionEventReason.INFO_CHANGED`.
- The `interaction.voice:parties` value has changed.
- The`interaction:extensions` DTO contains extended information about the new party added. See the *Agent Interaction SDK 7.6 Services API Reference* for further information.

Dual-Step Conference

As with the dual-step transfer, the dual-step conference requires two actions:

- The first agent initiates the conference by calling the second agent to be included in the conference. If the second agent answers the call, the consultation call starts.
- The first agent completes the conference; the consultation call ends and the two agents are in conference with the contact.

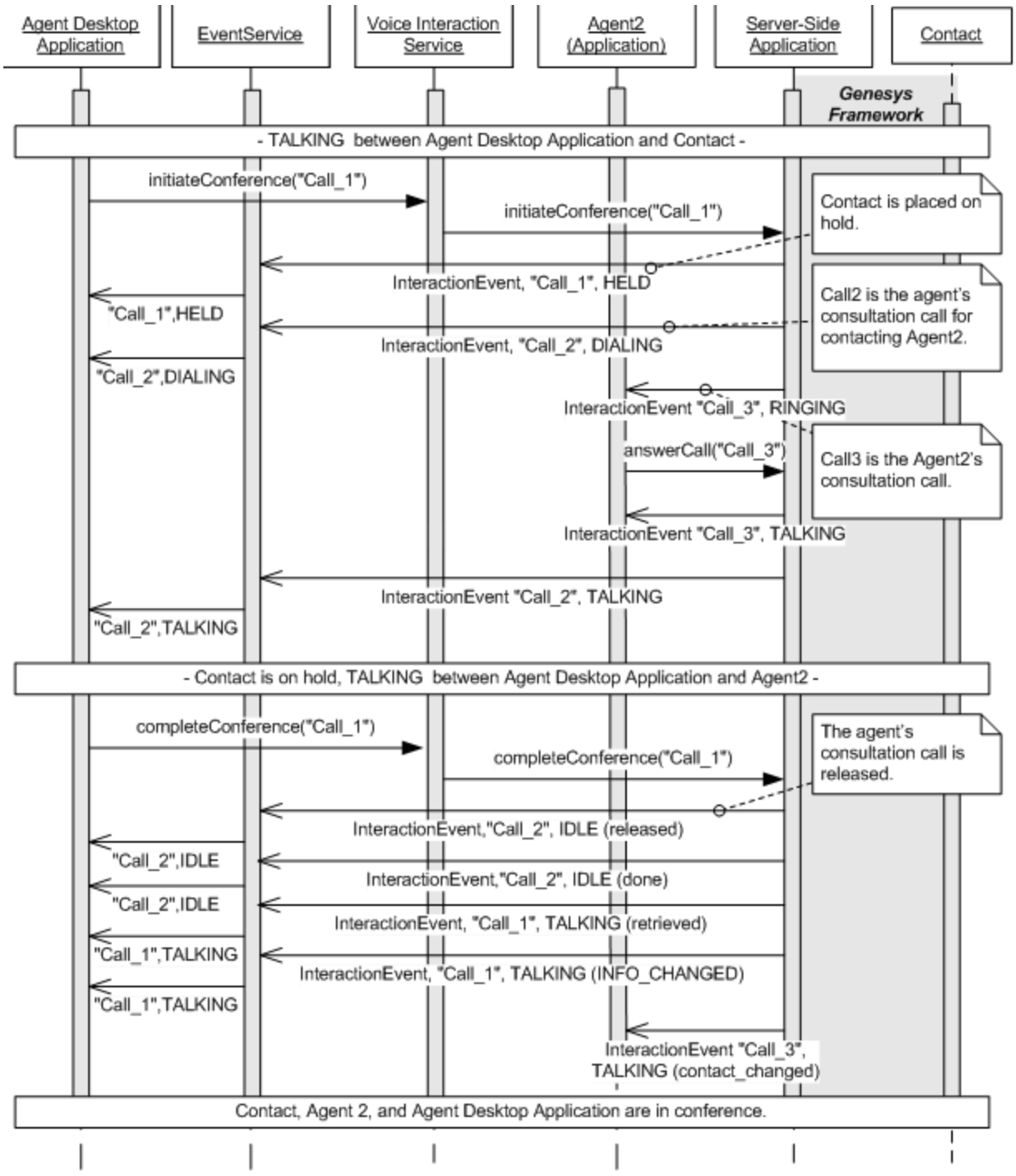
Steps of the Dual-Step Conference

Steps	InteractionVoiceAction	IInteractionVoiceService
Initiate	`INIT_CONFERENCE`	initiateConference()
Complete	`COMPLETE_CONFERENCE`	completeConference()

> **Important**
>
> To know if the dual-step conference action is possible, the `InteractionVoiceAction.INIT_CONFERENCE` has to be available in the `interaction.voice:actionsPossible` attribute of the voice interaction.

Dual-Step InteractionEvent Sequence

The following figure shows the sequence diagram showing method calls and `InteractionEvent` events received during the dual-step conference.



Dual-Step Conference

On Agent1's Side

In Dual-Step Conference, Agent1 deals with two interactions to establish the conference:

- Call_1 is the original interaction existing betweenAgent1 and the contact.
- Call_2 is the voice interaction existing between Agent1 and Agent2 who can be talking before entering in conference mode.

When Agent1 initiates the conference, Call_1 is held and Call_2 is created so as to enter Phase 1—Agent1 calling Agent2. When Agent2 answers Call_2, Agent1 and Agent2 are in Phase 2—they are talking.

WhenAgent1 completes the conference:

- Call_2 is released.
- A first InteractionEvent for Call_1 occurs: Call_1 is retrieved, with InteractionStatus.TALKING.
- The new party is added and the conference is established. An InteractionEvent occurs for Call_1:
  - The interaction:eventReason attribute is set to InteractionEventReason.INFO_CHANGED.
  - The interaction.voice:parties value has changed.
  - The interaction:extensions attributes contains extended information about the new party added. See the *Agent Interaction SDK 7.6 Services API Reference* for further information.

> **Important**
>
> To determine when your application can complete the conference, test if the InteractionVoiceAction.COMPLETE_CONFERENCE is available in the interaction.voice:actionsPossible attributes propagated in InteractionEvent.

On Agent2's side

Call_3 is the interaction originally created between Agent2 and Agent1 to perform the consultation call. This interaction is part of the conference once Agent1 requests a complete action. As a result, Agent2 application receives an InteractionEvent for Call_3:

- The interaction:eventReason attribute is set to InteractionEventReason.INFO_CHANGED.
- The interaction.voice:parties value has changed.
- The interaction:extensions attribute contains extended information about the new parties added to the interaction. See the *Agent Interaction SDK 7.6 Web Services API Reference* for further information.

Implementation Example

The following code snippet shows the method implementation for dual-step conference corresponding with Dual-Step Conference.

```
/// Agent1 initiates the transfer of Call_1 for Agent2
InteractionVoiceErrorDTO err0 = myInteractionVoiceService.initiateConferenceDTO( "Call_1",
   "DN_Agent2", // Dn of agent2 who is concerned by the conference
```

```
   null, // T-Server to use
   null, // attached data
   null, // reasons defined by the user
   null, // tExtensions
   null); // Attributes
//...
/// Agent1 and Agent2 are talking
VoiceError err1 = myInteractionVoiceService.completeConference( "Call_1",
   null, // reasons defined by the user
   null); // tExtensions
```

Leaving the Conference

When a party leaves a conference, the voice interaction for that party is released. Releasing the Call. The other parties each receive an `InteractionEvent`:

- The `interaction:eventReason` attribute is set to `InteractionEventReason.INFO_CHANGED`.
- The `interaction.voice:parties` value has changed.
- The `interaction:extensions` attribute contains extended information about the changed parties. See the *Agent Interaction SDK Services API Reference* for further information.

The following code snippet shows how to use the `IInteractionVoiceService.leaveConference()` method.

```
/// Agent1, Agent2 and Contact are in conference
// Agent1 is leaving
VoiceError err = myInteractionVoiceService.leaveConference( "Call_1",
   null, // reasons defined by the user
   null); // tExtensions
```

E-Mail Interactions

The e-mail interaction service is defined by the `IInteractionMailService` interface of the `com.genesyslab.ail.ws.interaction.mail` namespace. To integrate this service, your application deals with classes and enumerations of this namespace, and also with the classes and interface of the `com.genesyslab.ail.ws.interaction` namespace.

Introduction

The e-mail interaction service performs actions on e-mail interactions and enables your application to benefit from collaboration features. E-mail interactions are specific objects representing e-mails—that is, interactions using the `MediaType.EMAIL` media.
The following subsections provide overviews of e-mail handling and collaboration handling, and identify the services upon which the e-mail service depends.

Common E-Mail Features

The e-mail service is designed to allow your application to provide standard e-mail handling features, such as:

- Creating an e-mail.
- Sending an e-mail.
- Replying to an e-mail.
- Transferring an e-mail.

Collaboration Features

The collaboration features allow your agent desktop application to send invitations to other agents, requesting their assistance.
For example, the agent using your desktop application might be writing an outgoing e-mail replying to a customer's questions. But, this agent needs more information to answer a specific point, and therefore, requires collaboration with other agents.
The agent sends invitations to a set of other agents who might be able to provide assistance. Those invitations contain the agent's question and the problem e-mail. These agents receive invitations; if they have information to help, they reply to the invitation. The e-mail service collects the replies so that the initiating agent can complete the e-mail and send it.
The e-mail interaction service provides the following collaboration features:

- For the agent initiating the collaboration:
  - Creating and sending invitations.
  - Recalling, or reminding other agents about, invitations.
  - Reading collaborative answers.

- For the agent receiving an invitation:
  - Accepting, declining, replying to an invitation.
  - Sending a collaborative reply.
  - Saving, closing, or deleting a collaborative reply.

E-Mail Service Dependencies

The e-mail interaction service only deals with actions on e-mail interactions. When integrating this service, your application has to take into account the events that can occur due to this service. The e-mail interaction service depends on the following other services:

- The event service, to subscribe to and receive events.
- The agent service:
  - To deal with e-mail interactions, your application must log an agent into an `EMAIL` media type with the agent service.
  - While the agent is logged into an `EMAIL` media type, your application can use the `IInteractionMailService` to perform actions on e-mail interactions.

> **Important**
> See The Agent Service.

- The interaction service:
  - To access to `IInteractionMailService` attributes, your application uses the `IInteractionService` DTO methods.
  - Event attributes of the`interaction.mail`

domain are published in `InteractionEvent` defined in the `IInteractionService` interface.

> **Important**
> See also The Interaction Service.

E-Mail Essentials

The `IInteractionMailService` requests a single action on a single e-mail interaction at a time. Your application receives `InteractionEvents` when events occur on e-mail interactions.

E-Mail Attributes

The `IInteractionMailService` has no methods to read attributes. Your application must call one of the `IInteractionService.getInteractionDTO*()` methods. See also Handling Interaction DTOs.

Common Interaction Attributes

For each e-mail interaction, the attributes of the `IInteractionService` interface are also available. Your application should often use the following attributes:

- `interaction:interactionId`—The system interaction identifier, required in calls to the methods of the e-mail interaction service.
- `interaction:status`—The interaction status, defined with the `InteractionStatus` enumeration.
- `interaction:eventReason`—The `InteractionEventReason` value published when an `InteractionEvent` event occurs for a voice interaction.

Common E-Mail Attributes

Attributes common to all types of e-mail interactions are defined in the `IInteractonMailService` interface, and belong to `interaction.mail` domain. The following list is representative (but not exhaustive):

- `interaction.mail:toAddress` is a string containing the e-mail address field for the e-mail's receiver.
- `interaction.mail:fromAddress` is a string containing the sender's e-mail address field.
- `interaction.mail:messageText` is a string containing the text of the e-mail.
- `interaction.mail:ccAddresses` is a string containing the e-mail addresses that are to receive a copy of the e-mail.

The subject of an e-mail is defined in the `interaction:subject` attribute of the `IInteractionService` interface.

Other E-Mail Attributes

The `IInteractionMailService` interface includes two additional subdomains dedicated to common e-mail interactions:

- `interaction.mail.in`—Subdomain dedicated to incoming e-mails.
- `interaction.mail.out`—Subdomain dedicated to outgoing e-mails.

These attributes are available according to the type of the e-mail interaction being processed. Those types are detailed in the E-Mail Types subsection below.
See the *Agent Interaction SDK 7.6 Services API Reference* for further details about the attributes of the `interaction.mail` domain and its subdomains.

E-Mail Types

Your application accesses types for e-mail interactions by reading the

`interaction:interactionType` attribute of the `IInteractionService` interface.
The types of interactions handled by the e-mail service can be divided into two categories:

- `EMAIL_*` for common e-mails interactions.
- `COLLABORATION_*` for collaborative interactions.

The following table presents the types of common e-mail interactions that the e-mail service handles.

Interaction Types for the E-Mail Service

Interactions	InteractionType	Description
Incoming E-mails	`EMAIL_IN`	Interactions for e-mails received by the agent
Outgoing E-mails	`EMAIL_OUT`	Interactions for e-mails sent by the agent
Outgoing Reply E-mails	`EMAIL_OUT_REPLY`	Interactions for e-mail replying to incoming e-mails

The e-mail interaction types are detailed in the following subsections. See Collaboration Interaction Types for further information about collaboration interactions.

Incoming E-Mails

The e-mail interaction type corresponding to incoming e-mails is `InteractionType.EMAIL_IN`. It represents e-mails received by an agent. For this type of e-mail interaction, common e-mail attributes—`interaction.mail:*` such as message, subjects, and addresses—are already filled.

The `interaction.mail.in:currentReplyMailoutId` attribute is specific to incoming e-mails, and is filled with an e-mail interaction identifier if there exists an outgoing e-mail (whether created or sent) replying to this incoming e-mail.

Use the `IInteractionService.getInteractionDTO*()` method to read this attribute value. To deal with incoming e-mails, see Answering an E-Mail and Replying to an E-Mail.

Outgoing E-Mails

The e-mail interaction type corresponding to outgoing e-mails is `InteractionType.EMAIL_OUT`. It represents e-mails interactions that your application creates and sends with the `IInteractionMailService` interface.
For this type of e-mail interaction, most of the writable attributes—such as message, subject and addresses—are empty when the interaction is created. Your application should fill them with `IInteractionService.setInteractionDTO()` method.
Some specific attributes are defined in the `interaction.mail.out` subdomain. interaction. Use also

the IInteractionService.getInteractionDTO*() method to read those attributes.
To deal with outgoing e-mails, see Sending an E-Mail.

Outgoing E-Mails for a Reply

The corresponding e-mail interaction type corresponding to outgoing reply e-mails is InteractionType.EMAIL_OUT_REPLY. It represents outgoing e-mails replying to an incoming e-mail. These interactions are created due to aInteractionMailAction.REPLY action performed with aIInteractionMailService.reply() call.
The attributes available are those for common outgoing e-mails, that is, interaction.mail:* and interaction.mail.out:*. For this type of e-mail interaction, common e-mail attributes—interaction.mail:* (such as subjects and addresses)—are already filled with information extracted from incoming e-mails.
To deal with reply e-mails, see Replying to an E-Mail.

E-Mail Actions

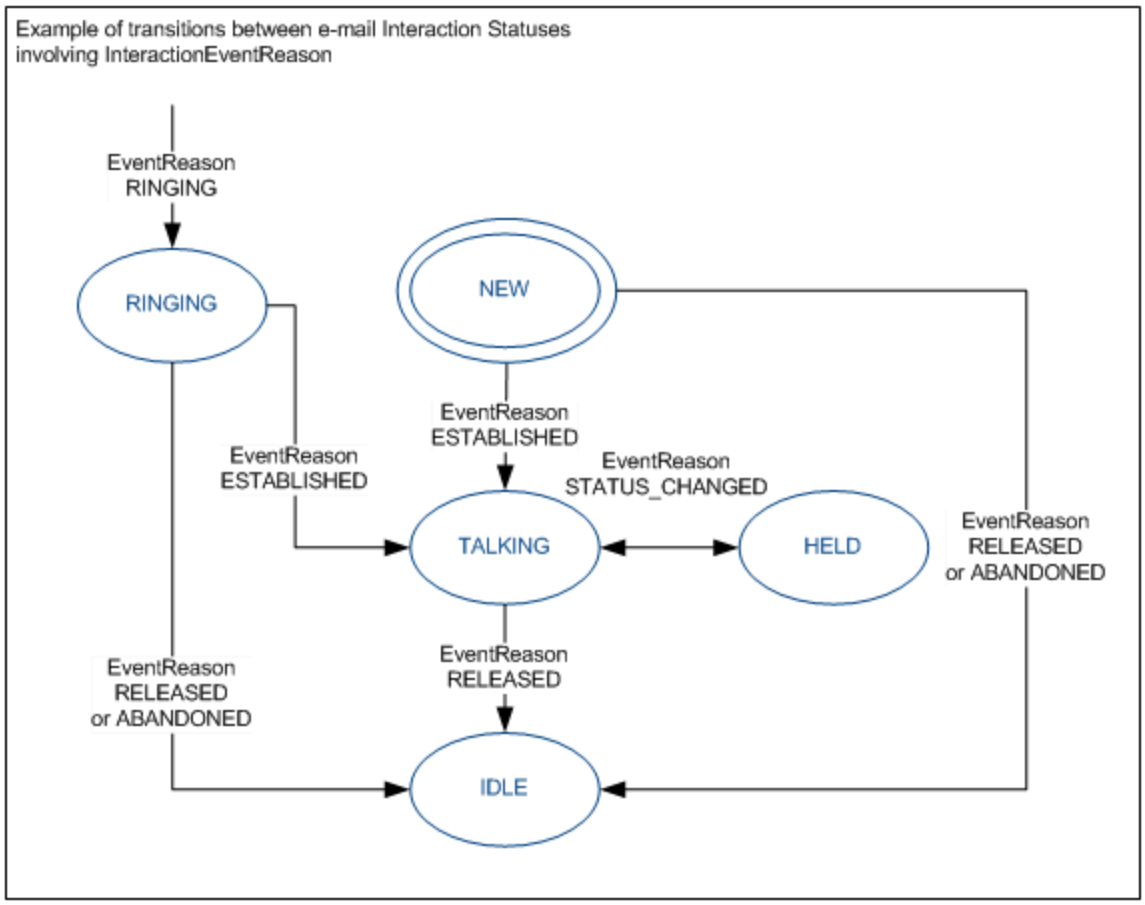
The InteractionMailAction enumeration defines all the available e-mail actions of the IInteractionMailService interface. Constants each correspond to one e-mail interaction service method. For example, the ANSWER constant corresponds to the IInteractionMailService.answer() method.
Your application can access possible actions for e-mail interactions (except for a collaboration interaction) by reading the value of the interaction.mail:actionsPossible attribute of the IInteractionMailService interface.
The IInteractionMailService interface has no methods to read attributes. Your application must use one of the InteractionService.getInteractionsDTO*() methods. See also Handling Interaction DTOs.

E-Mail Statuses

The status of an e-mail interaction is represented by a common interaction attribute defined in the IInteractionService as the interaction:status attribute. Your application cannot read this attribute with an IInteractionMailService method, so you must use one of the IInteractionService.getInteractionDTO*() methods. See also Handling Interaction DTOs.
The possible statuses of an e-mail interaction are parts of the InteractionStatus enumeration of the com.genesyslab.ws.interaction namespace.
The status of an e-mail interaction changes if:

- A successful action is confirmed by an event to the server-side application; for example, the e-mail has been sent and the e-mail interaction status change to InteractionStatus.IDLE.
- A CTI event occurred; for example an error occurred and e-mail interaction status change to InteractionStatus.IDLE.

The following diagram shows non-exhaustive transitions that can occur between e-mail interaction statuses.



Generalized State Diagram for E-Mail Interactions (Incomplete)

> ## Warning
> This figure is provided as an informative example. It is non-exhaustive: it does not include all the possible transitions.

The diagram above shows a sequence of e-mail interaction statuses, with `InteractionEventReason` values as transitions. The `InteractionEventReason` value for a status change is propagated with the `interaction:eventReason` attribute in `InteractionEvent`.

> ## Warning
> Do not assume any status sequence in your application design. Design your application always to update with the possible agent actions provided and the current interaction status.

E-Mail Interactions Events

The `IInteractionMailService` works with the `InteractionEvent` of the `IInteractionService`. Most events that occur on e-mail interactions are `InteractionEvents`. Therefore for these events, published attributes are `IInteractionService` and `IInteractionMailService` attributes that have the event property.
You have to subscribe to a `TopicsService` defined for the `IInteractionService`. This `TopicsService` must specify (in its `TopicsEvents`) the e-mail interaction DTO to retrieve.

> **Important**
>
> For further details on the `InteractionEvent` mechanism, see Using IInteractionService.

The following code snippet shows how to receive `InteractionEvent` occurring on any interaction belonging to `agent0`, and how to ensure the propagation of e-mail interaction DTOs for an e-mail interaction event.

```
/// Defining a Topic Services for interaction service
TopicsService[] myTopicsServices = new TopicsService[1] ;
myTopicsServices[0] = new TopicsService() ;
myTopicsServices[0].serviceName = "InteractionService" ;

TopicsEvent[] myTopicsEvents = new TopicsEvent[1] ;
myTopicsEvents[0] = new TopicsEvent() ;

/// the targeted events are InteractionEvent
myTopicsEvents[0].eventName = "InteractionEvent" ;
myTopicsEvents[0].attributes = new String[]{ "interaction:*", "interaction.mail:*"};

/// The InteractionEvent concern agent0 interactions
myTopicsEvents[0].triggers = new Topic[1];
myTopicsEvents[0].triggers[0] = new Topic();
myTopicsEvents[0].triggers[0].key = "AGENT";
myTopicsEvents[0].triggers[0].value = "agent0";

/// The InteractionEvent must concern some EMAIL types interactions
myTopicsEvents[0].filters = new Topic[3];

myTopicsEvents[0].filters[0] = new Topic();
myTopicsEvents[0].filters[0].key = "INTERACTION_TYPE";
myTopicsEvents[0].filters[0].value = "EMAIL_OUT";

myTopicsEvents[0].filters[1] = new Topic();
myTopicsEvents[0].filters[1].key = "INTERACTION_TYPE";
myTopicsEvents[0].filters[1].value = "EMAIL_OUT_REPLY";

myTopicsEvents[0].filters[2] = new Topic();
myTopicsEvents[0].filters[2].key = "INTERACTION_TYPE";
myTopicsEvents[0].filters[2].value = "EMAIL_IN";

/// ... Subscribe to the event service
```

See also The Event Service

Common E-Mail Management

The common actions of the `IInteractionMailService` on e-mail interactions are presented in Common Features for an E-Mail Interaction.

Common Features for an E-Mail Interaction

Actions	InteractionMail Action	IInteractionMail Service	Relevant InteractionTypes
Send an e-mail	`SEND`	`send()`	`EMAIL_OUT EMAIL_OUT_REPLY`
Answer an e-mail	`ANSWER_CALL`	`answer()`	`EMAIL_IN`
Reply to an e-mail	`REPLY`	`replyDTO()`	`EMAIL_IN`
Delete an e-mail	`DELETE`	`delete()`	`EMAIL_OUT EMAIL_OUT_REPLY EMAIL_IN`
Transfer an e-mail	`TRANSFER`	`transfer()`	`EMAIL_IN`
Release an e-mail	`RELEASE_CALL`	`release()`	`EMAIL_OUT EMAIL_OUT_REPLY EMAIL_IN`
Mark done an e-mail	`MARK_DONE`	`markDone()`	`EMAIL_OUT EMAIL_OUT_REPLY`

Your application uses the `IInteractionMailService` features to send requests to the server-side application. Once the Genesys Solution—that is, the server-side application and the Genesys Framework—has performed the corresponding action, the event service receives `InteractionEvent`s if the application has subscribed to the correct topics.
The `InteractionEvent` propagates the interaction status changes and the new possible actions corresponding to the e-mail interaction identifier (`interaction:interactionId`).
The following sections detail the common e-mail interaction actions and provide you with sequence diagrams to help you to understand the action sequences.

> ### Warning
> You should not use the sequence diagrams to make assumptions about actions' availability. Instead, use the `InteractionMailAction` possible actions provided in DTOs.

Sending an E-Mail

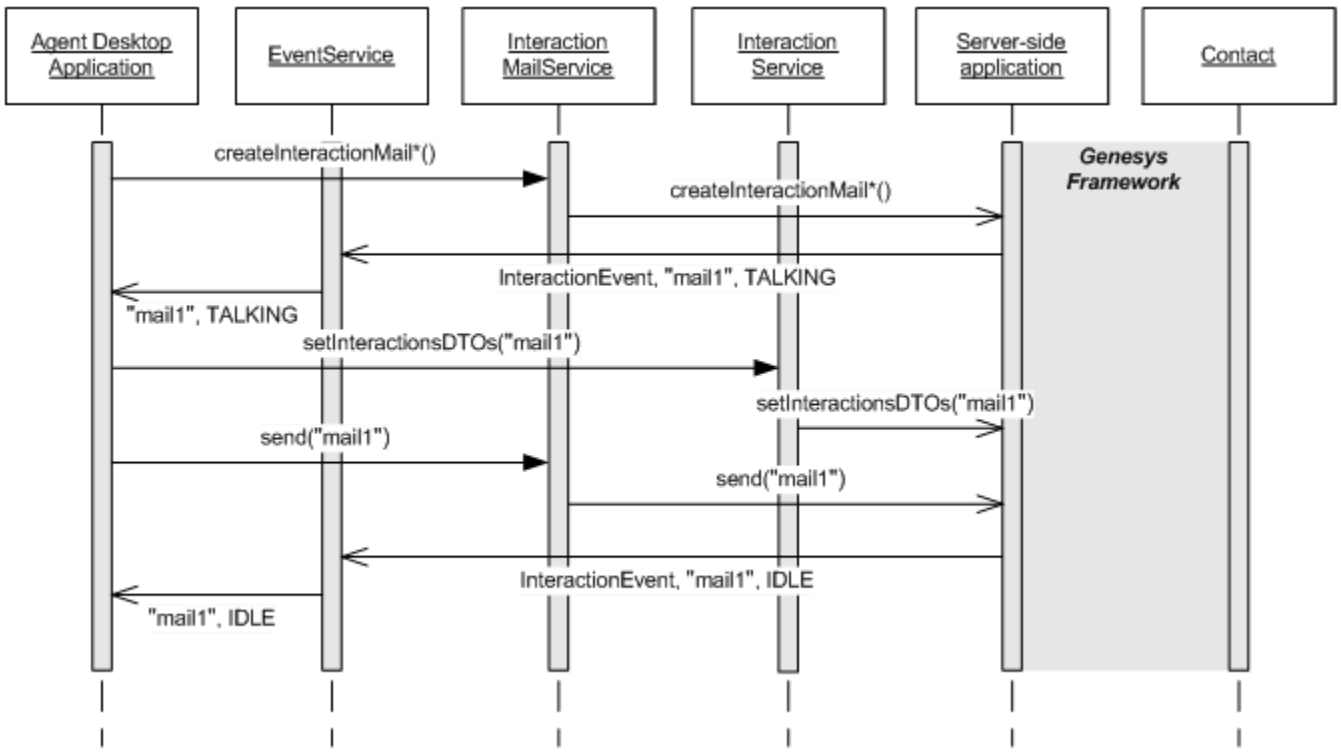
The `IInteractionMailService` interface lets your application send e-mails. There are two scenarios for sending an e-mail:

- Your application has replied to an incoming e-mail and needs to send the reply. See Replying to an E-Mail.
- Your application sends a brand new e-mail as detailed in this section.

This section and its subsections detail how to create and send common outgoing e-mails. The corresponding e-mail interaction type is `InteractionType.EMAIL_OUT`.
When your application needs to send an e-mail, it requires a logged-in agent on an `EMAIL` media type. Your application can test this condition with the `agent:loggedMedias` attribute of the `IAgentService` interface.
Sending the e-mail requires four steps, as follows:

1. Create an outgoing e-mail with one of the IInteractionMailService.createMailInteraction*() methods. See Creating an Outgoing E-Mail Interaction.
2. Fill the outgoing e-mail interaction fields with the `IInteractionService.setInteractionDTO()` method. See Filling an E-Mail Interaction.
3. Send the outgoing e-mail interaction with the `IInteractionMailService.send()` method. See Sending an E-Mail.

The following sequence diagram shows the sequence of actions, requests, and `InteractionEvents` received when sending an outgoing e-mail.



Sequence Diagram for Sending an E-Mail

In Sequence Diagram for Sending an E-Mail, once the outgoing e-mail interaction is created, this interaction status is `InteractionStatus.TALKING` as notified in the received `InteractionEvent`. This status lets your application modify the e-mail data—that is, addresses, text, and so on—with `IInteractionMailService` attributes that have the `write` property. Because the e-mail has just been created, these types of attributes have `null` values.
Then, the `IInteractionMailService.send()` method performs itself a release of the interaction once the outgoing e-mail is sent.

> **Important**
>
> To use `IInteractionMailService` methods, update the possible actions propagated in the received `InteractionEvent`.

The following subsections detail the method calls for creating and sending an e-mail.

Creating an Outgoing E-Mail Interaction

`IInteractionMailService` has two available methods for creating an e-mail interaction:

- `createInteractionFromPlaceDTO()` creates an e-mail interaction with a place identifier if:
  - An agent is logged on the place.
  - An agent is logged on the place's e-mail media type.
- `createInteractionFromAgentDTO()` creates an e-mail interaction with an agent identifier if the agent is logged into an e-mail media type.

> **Important**
>
> Both methods create an outgoing e-mail interaction.

The following code snippet creates an e-mail interaction using the `IInteractionMailService.createInteractionFromAgentDTO()` method:

```
/// Creating the e-mail interaction with the e-mail service
InteractionDTO myEMailDTO = myInteractionMailService.createInteractionMailAgentDTO(
myAgentId, // identifier of the logged agent
  myAgentQueue, // identifier of the agent queue
  null); //an array of key attributes to retrieve in the DTO

/// if the creation succeeded, retrieving the interaction id
if(myEMailDTO!=null)
{
   String myNewEMailId=myEMailDTO.interactionId;
}
```

The above code snippet shows that if the interaction is successfully created, its identifier is available in the `InteractionDTO` object returned by the method.

> **Important**
>
> Upon the outgoing e-mail creation, its writable attributes have `null` values.

Sending the E-Mail Interaction

The following code snippet shows how to send the previously created outgoing e-mail:

```
myInteractionMailService.send(myNewEMailId, // Id of the e-mail interaction to send
  myQueue); // The Queue (should not be null)
```

If the send action is performed, the outgoing e-mail interaction status changes to `IDLE` because the send feature has released the e-mail interaction. Your application receives an `InteractionEvent` with this new status and with the updated possible e-mail actions.

Filling an E-Mail Interaction

The e-mail interactions fields that your application might have to fill are `interaction:*` and `interaction.mail:*` attributes with the `write` property. See the *Agent Interaction SDK 7.6 Services API Reference* for further details.
Filling the e-mail interaction's fields requires the interaction identifier, an `InteractionDTO` object, and an `IInteractionService` instance. The following code snippet fills an empty outgoing e-mail identified with `myNewEMailId`.

```
/// Creating the DTO to fill with attributes key-values
InteractionDTO myEmailDTO = new InteractionDTO();

// Setting the id of the outgoing e-mail to fill
myEmailDTO.interactionId = myNewEMailId;

// Creating a Key-value array
myEmailDTO.data = new KeyValue[3];

// Setting the message text
myEmailDTO.data[0] = new KeyValue();
myEmailDTO.data[0].key= "interaction.mail:messageText";
myEmailDTO.data[0].value= “Text of the e-mail to send”;

// Setting the e-mail addresses
myEmailDTO.data[1] = new KeyValue();
myEmailDTO.data[1].key= "interaction.mail:toAddresses";
myEmailDTO.data[1].value= myContact@company.com;

//Setting a subject for the e-mail
myEmailDTO.data[2] = new KeyValue();
myEmailDTO.data[2].key= "interaction:subject";
myEmailDTO.data[2].value= "Subject of the e-mail";
```

```
// Writing the DTO with the interaction service
myInteractionService.setInteractionsDTO( new InteractionDTO[]{ myEmailDTO });
```

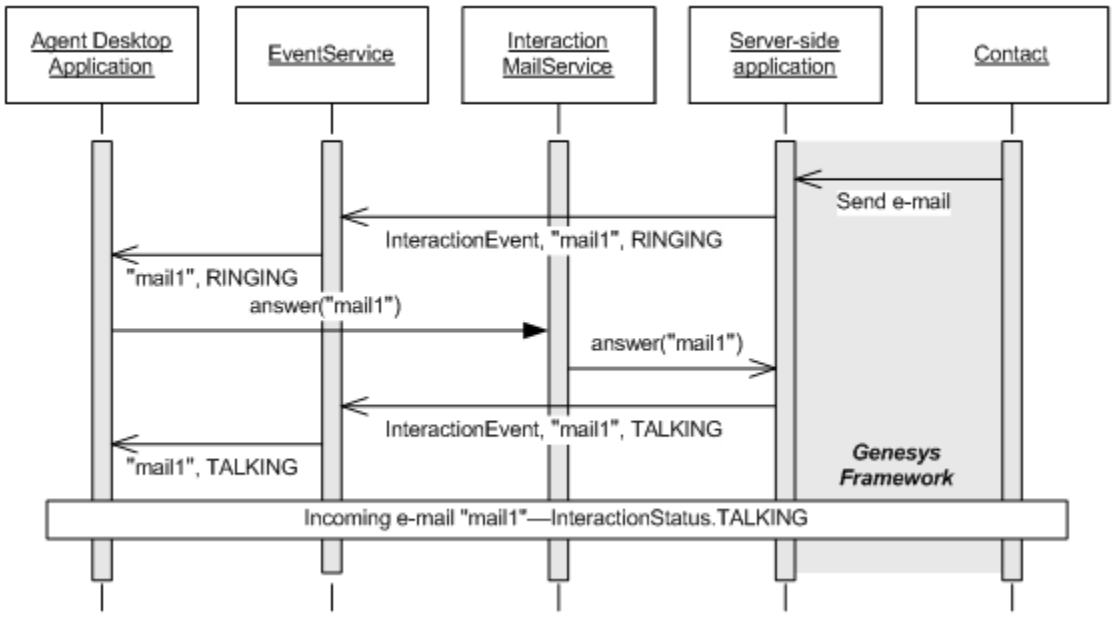
For further detail on `InteractionDTO` and `IInteractionService`, see The Interaction Service.

Answering an E-Mail

The answer feature of the e-mail interaction service is equivalent to the answer feature of the voice interaction service. Your application uses it on an incoming e-mail interaction so as to “accept” the interaction. Once your application has answered the e-mail interaction, the e-mail interaction is assigned to the agent’s place.

For example, your application receives an `InteractionEvent` for an incoming e-mail interaction with the `InteractionStatus.RINGING` status. Your application might then display a dialog box to inform the agent of the new incoming e-mail. If the agent chooses to answer the e-mail, your application can add the incoming e-mail to the agent desktop’s mailbox.

The `IInteractionMailService.answer()` feature works for incoming e-mail interactions. The corresponding e-mail interaction type is `InteractionType.EMAIL_IN`.
The following diagram shows the sequence of actions on incoming e-mail interactions, and the received `InteractionEvents`.



Sequence Diagram for Answering an E-Mail

This diagram shows that an `InteractionEvent` occurs for an incoming interaction e-mail with `InteractionStatus.RINGING` status. Once the `IInteractionMailService` has answered, the new interaction status is `Interactionstatus.TALKING`, which means that the incoming e-mail belongs to the logged-in agent.

The following code snippet illustrates the `IInteractionMailService.answer()` method call:

```
myInteractionMailService.answer( myIncomingMailId,// Id of the incoming e-mail to answer
   null); // KeyValue[] reasons
```

Unlike with a newly created outgoing e-mail, the incoming e-mail interaction attributes do not have `null` values. While the incoming e-mail interaction status is `InteractionStatus.TALKING`, your application might display pertinent information about the incoming e-mail—such as, the sender identity, the subject, and the message itself identified in `IInteractionService` and `IInteractionMailService` attributes.

For further detail about readable attributes of these services, see the *Agent Interaction SDK 7.6 Services API Reference.*

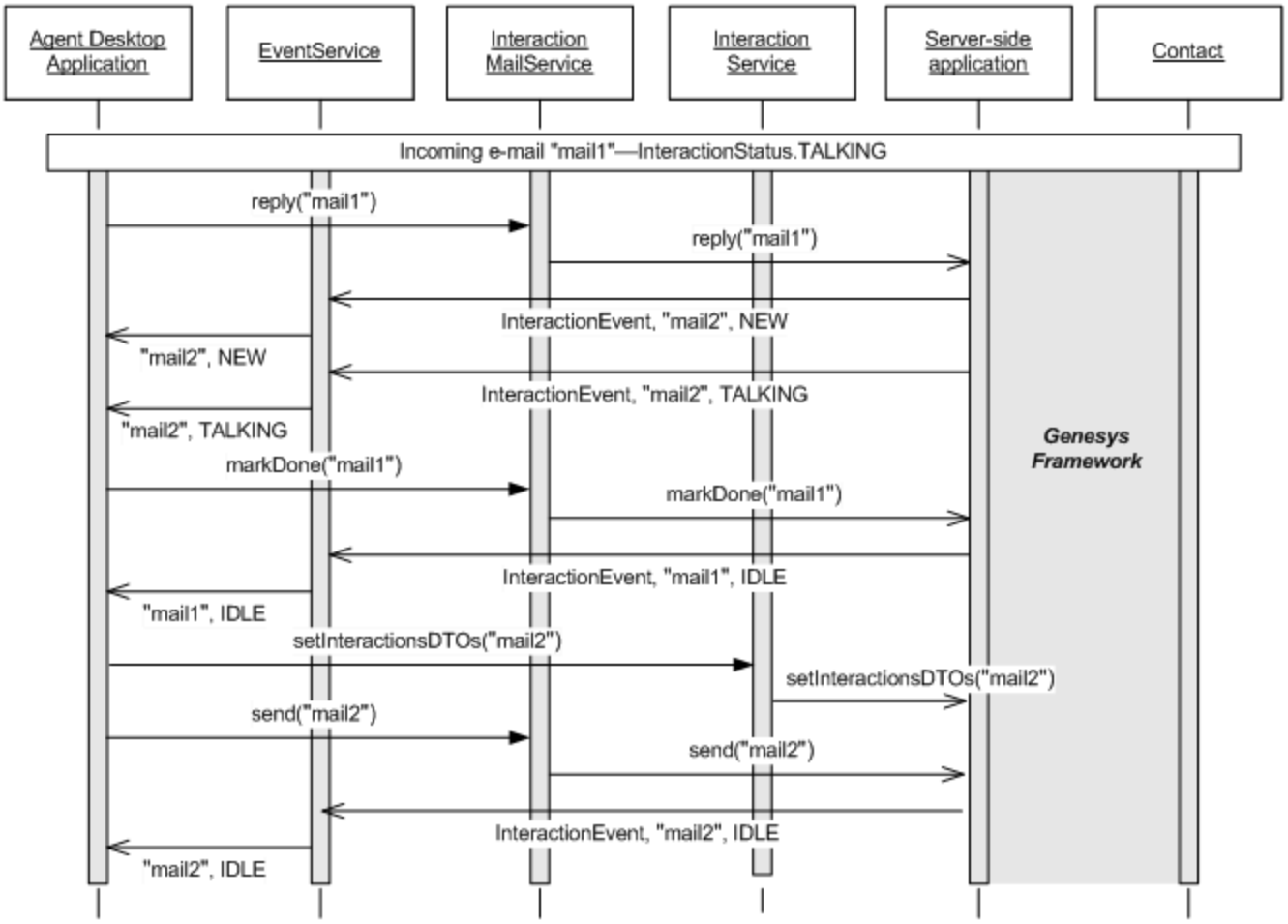
Replying to an E-Mail

Your application can use the `IInteractionMailService` reply feature only with e-mail interactions of type `InteractionType.MAIL.IN`. Moreover, this feature must be available in the possible actions of an incoming e-mail.
When the `IInteractionMailService` interface requests a reply, it creates an outgoing e-mail interaction of type `InteractionType.EMAIL_OUT_REPLY`, which your application must fill and send.
If you use the `IInteractionMailService.replyDTO()` method, replying to an e-mail involves the following steps:

1. Create an outgoing reply e-mail with the IInteractionMailService.replyDTO() method.
2. Mark the incoming e-mail interaction as done when you no longer need it. See Marking Done an E-Mail Interaction.
3. Fill the outgoing reply e-mail interaction fields with the `IInteractionService.setInteractionDTO()` method. See Filling an E-Mail Interaction.
4. Send the outgoing reply e-mail interaction with the `IInteractionMailService.send()` method. See Sending an E-Mail.

> **Important**
>
> If you call the IInteractionMailService.replyExDTO() method and set the auto-mark-done parameter to true, you do not have to mark the incoming e-mail interaction as done.

The following sequence diagram shows the sequence of actions, requests, and `InteractionEvents` sequence for a typical reply to an incoming e-mail, that is, by calling the `IInteractionMailService.replyDTO()` method.



Sequence for Replying to an Incoming E-Mail (no auto-mark done)

In the above diagram, `mail1` is an incoming e-mail interaction identifier. When the `IInteractionMailService` interface has made the reply request, the server-side application creates an outgoing reply e-mail, `mail2`, filled with some information of `mail1`.
The event service receives an `InteractionEvent` for the `mail2` interaction, notifying its `InteractionStatus.TALKING` status. The agent application can mark `mail1` as done and the event service receives the corresponding `InteractionEvent` event for the status change.
The agent desktop application can fill the remaining fields of`mail2` with the `IInteractionService` interface. Once the replied `mail2` interaction is sent, the server-side application releases the `mail2` interaction. The event service receives the corresponding `InteractionEvent` for the status change.
The following code code snippet shows how to implement the `IInteractionMailService.replyDTO()` method to reply to `mail1` and retrieve the corresponding interaction ID.

```
InteractionDTO myReplyDTO= myInteractionMailService.replyDTO("mail1",
   myQueue, // the queue ID
   true,    // Reply to all
   null);   // string keys of the attributes to retrieve
```

```
// Displaying the ID of the created interaction
System.Console.WriteLine("Replying Interaction Id: "+ myReplyDTO.interactionId);
```

> **Important**
> The `InteractionDTO` object can retrieve attributes that have the `read` property.

Marking Done an E-Mail Interaction

Your application should mark done an e-mail interaction when the agent no longer needs it. For instance, if the agent sent several replies to an e-mail and decides that this e-mail no longer requests agent processing, he or she marks it as done.
The following code snippet shows how to mark the previously created outgoing e-mail as done:

```
myInteractionMailService.markDone(myNewEMailId);
```

When the interaction is marked as done, your `IInteractionMailService` and `IInteractionMailService` interfaces can no longer access this interaction or perform your application's requests using its identifier.

Collaboration Essentials

A collaboration session involves several types of interactions. A collaboration interaction is an e-mail interaction that manages additional collaboration data. The e-mail interaction service includes collaboration attributes to access that data, which includes collaboration status.
During a collaboration session, your application can use the e-mail service to:

- Manage the collaboration, if the agent is the initiator.
- Participate in a collaboration session.

When an agent initiates the collaboration, he or she sends invitations to the participants. After a refresh of their applications, the agent can monitor the collaboration activity, and all the participants can access the corresponding invitations. When a participant has replied, the corresponding invitation is fulfilled.
If the agent is a participant, your application only manages interaction events and uses the e-mail service to perform collaboration actions on the collaborative interactions.
The following sections present the details behind this general description.

Collaboration Attributes

Your application can call one of the `IInteractionService.getInteractionDTO*()` methods to read the collaboration attributes. Opening a Workbin Interaction.

Collaborative Interaction Attributes

As the collaborative interactions are e-mail interactions, the following attributes are available for a collaborative interaction:

- `interaction:*`—Common interaction attributes.
- `interaction.mail.*`—Common e-mail attributes.
- `interaction.mail.in.collaboration:*`—Additional collaborative attributes for parent invitations.

Outgoing E-Mail Attributes

An agent initiates a collaboration session when writing an outgoing (reply) e-mail. Your application can monitor this session with some attributes dedicated to the collaboration management and defined in the `interaction.mail.out` domain:

- `interaction.mail.out:invitations`—All the invitations sent by the agent who initiated the collaboration session.
- `interaction.mail.out.invitationSentId`—The system identifier of an outgoing e-mail whose invitations were successfully sent to participants.

Your application can use the following `IInteractionMailService` DTO methods to get interaction data:

- `getSentInvitationsDTO()`—Retrieves the interaction data of each sent invitation.
- `getCollaborativeReplyDTO()` —Retrieves the interaction data of a received reply. See Retrieving a Collaborative Reply.

Collaboration Interaction Types

Your application accesses types for collaborative e-mail interactions using the `interaction:interactionType` attribute of the `IInteractionService` interface. The following diagram presents the types of collaborative e-mail interactions that the e-mail service handles.

Types of Collaborative E-Mail Interactions

Interactions	InteractionType	Description
Inbound invitation	`COLLABORATION_INVIT_IN`	Interactions for inbound invitations that the participant (or child) receives or that the agent (or parent) sent in a collaboration..
Reply to invitation	`COLLABORATION_REPLY_OUT`	Interactions for a collaborative reply sent

Interactions	InteractionType	Description
		by a participant (or child).

Incoming Invitation

The parent can see and manage the invitation interactions of the participants:

- Child invitation (invitation from the child point of view):
  - Each participant in the collaboration session receives an incoming child invitation.
  - This interaction informs the participant of the collaboration request.
  - For information about managing child invitation, see Participating in a Collaboration Session.
- Parent invitation (invitation from the parent point of view):
  - For each invitation sent to a participant, the agent who initiates the collaboration can access the corresponding interaction.
  - The agent uses parent invitations to monitor the collaboration and the participants'replies.
  - For information about managing parent invitations, see Managing a Collaboration Session.

Collaborative Reply

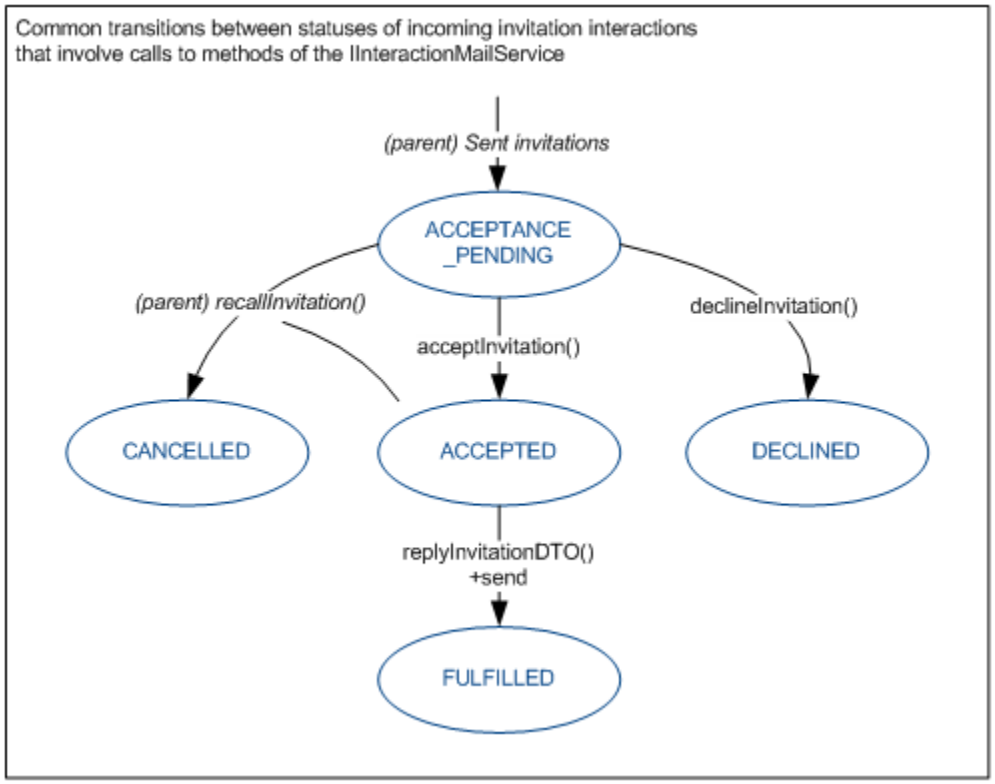
The collaborative reply is an outgoing e-mail replying to a child incoming invitation. As with an outgoing reply e-mail, some fields are filled at the interaction's creation—for instance, `interaction.mail:to` and`interaction.mail:from`.
Use a collaborative reply in the same way as an outgoing e-mail interaction. For further details, see Participating in a Collaboration Session.

Collaboration Status

The `CollaborationStatus` enumeration lists the possible collaboration statuses. Only collaborative interactions can have a collaboration status, available in the `interaction.mail:collaborationStatus` attribute. This status is an additional data. The `interaction:status` attribute is available for any collaborative interactions.
An application that initiates a collaboration has a specific interest in the collaboration status of its parent invitations.When a parent invitation takes on a `FULFILLED` status, the application can get the system identifier in the `interaction.mail.in.collaboration:collaborativeReply` attribute of this interaction to access the corresponding reply.

> **Important**
> Collaboration status changes do not launch additional `InteractionEvent` events. There is no notification.

To refresh the collaboration status of a collaborative interaction, your application must periodically read that status.
The following state diagram shows common transitions existing between collaboration statuses.



Generalized State Diagram for an Incoming Invitation (Incomplete)

Collaboration Handling

The typical `IInteractionMailService` actions upon e-mail invitations are presented in the following table.

Features For Collaboration

Agent	Actions	IInteractionMail Service	InteractionType
PARENT	Send an invitation	`sendInvitationTo*()`	`EMAIL_OUT EMAIL_OUT_REPLY`

Agent	Actions	IInteractionMail Service	InteractionType
	Remind an invitation	`remindInvitation()`	`EMAIL_OUT EMAIL_OUT_REPLY`
	Recall an invitation	`recallInvitation()`	`EMAIL_OUT EMAIL_OUT_REPLY`
	Retrieve DTOs for sent invitations.	`getSentInvitationsDTO()`	`EMAIL_OUT EMAIL_OUT_REPLY`
	Retrieve the DTO for a collaborative reply interaction.	`getCollaborativeReplyDTO()`	`COLLABORATION_INVIT_IN`
CHILD	Accept an invitation	`acceptInvitation()`	`COLLABORATION_INVIT_IN`
	Refuse an invitation	`declineInvitation()`	`COLLABORATION_INVIT_IN`
	Reply to an invitation	`replyInvitationDTO()`	`COLLABORATION_INVIT_IN`
	Send a collaborative reply	send()	`COLLABORATION_REPLY_OUT`

This table separately shows actions related to parent versus child invitation interaction. The `InteractionType` specified in the table's last column is the interaction type of the identifier parameter in the `IInteractionMailService` method call.

Managing a Collaboration Session

To request the collaboration of other agents, your application must be working on an outgoing e-mail interaction, as shown in Features For Collaboration. As your application initiates the collaboration, it becomes the parent of all sent invitations.
The following steps detail the general sequence of actions that your application is likely to follow:

1. Sending invitations to the participants.
2. Recalling, or reminding about, invitations if required.
3. Retrieving a DTO for each collaborative reply sent by a participant.
4. Sending the outgoing e-mail. For details, see Sending the E-Mail Interaction.

Sending Invitations

Depending on the method called to send invitations, your application activates a specific mode:

- `sendInvitations()`—Sends the invitations to the participants in the pull mode. The child invitations are available in workbins.
- `transferInvitations()`—Transfers the invitations to the participants in the push mode. Each participant receives the invitation as an incoming interaction.

A single call to these methods send all invitations, as shown in the following code snippet:

```
Participant[] myParticipants=new Participant[2];
myParticipants[0] = new Participant();
myParticipants[0].name = "agent0";
myParticipants[0].type = ParticipantType.AGENT;
myParticipants[1].name = "agent1";
myParticipants[1].type = ParticipantType.AGENT;
InteractionDTO[] mySentInvitations= myInteractionMailService.sendInvitations(myInteractionId,
myParticipants, "Do you have info about that?", //the trouble "Troubleshooting", //the
subject of the collaboration new string[]{"interaction.*:*"}); //the invitation attributes
//to return in DTOs.
```

The method returns an array of interaction DTOs. Each interaction DTO contains the data of a sent invitation.

Reminding About Invitations

Sometimes, agents who received invitations might forget to reply. Your application can use the `IInteractionMailService.remindInvitation()` method to remind a participant that a collaboration session is still in progress. This method does not inform all participants—only a single one. Call this method if the invitation's collaboration status is: `ACCEPTED` or `ACCEPTANCE_PENDING`. It takes as a parameter the interaction identifier of the parent invitation that corresponds to one participant.
The following code snippet shows a call to this method:

```
myInteractionMailService.remindInvitation( "parentInvitationIdForAgent0", "myPlaceId"); //
place ID of the agent reminding the invitation
```

Retrieving a Collaborative Reply

When a collaboration participant sends a reply to the inviting agent, your application propagates the identifier of the collaborative reply in the `interaction.mail.in.collaboration:collaborativeReply` attribute.
The following code snippet retrieves a collaborative reply interaction with the getCollaborativeReplyDTO() method:

```
InteractionDTO myReplyDTO = myInteractionMailService.getCollaborativeReplyDTO( "myReplyID",
// interaction ID of the reply
new string[]{"interaction.*:*"}); // attributes to get in DTO
```

> **Important**
> Replying to a collaborative reply is not possible.

Recalling an Invitation

If the initiating agent no longer needs the collaboration session—for example, he has found the necessary information—he or she can decide to recall some pending invitations.
In this case, your application uses the `IInteractionMailService.recallInvitation()` method to cancel the invitations. This method does not recall all invitations—only a single one. Call this method if the invitation's collaboration status is: `ACCEPTED` or `ACCEPTANCE_PENDING`. It takes as a parameter the interaction identifier of the parent invitation that corresponds to one participant. If the recall is successful, the collaborative status of the invitation changes to `CANCELLED`.
The following code snippet shows a call to this method:

```
myInteractionMailService.recallInvitation( "parentInvitationIdForAgent0",
"myPlaceId"); // place ID of the agent recalling the invitation
```

Participating in a Collaboration Session

In push mode, when an agent receives an incoming invitation, he or she receives an e-mail interaction of type `COLLABORATION_INVIT_IN` which has both `InteractionStatus.RINGING` and `CollaborationStatus.ACCEPTANCE_PENDING` statuses.
If the agent accepts the invitation, that agent participates in the collaboration session. To end his or her participation, the agent must reply to the invitation, as described in the following subsections.

Accepting an Invitation

Once the application has answered the interaction (see Common E-Mail Management), your application can accept the incoming invitation to enter the collaboration session, by calling the `IInteractionMailService.acceptInvitation()` method, as shown in the following code snippet.

```
myInteractionMailService.acceptInvitation( "myChildInvitationID", //Interaction ID
  "myPlaceID");
```

If the `ACCEPT_INVITATION` action is successful, the collaboration status of the the invitation interaction should become `CollaborationStatus.ACCEPTED`.

For further details about collaboration statuses, see Collaboration Status.

Replying to an Invitation

Your application can use the `IInteractionMailService.replyInvitationDTO()` method to create a collaborative reply interaction of type `COLLABORATION_REPLY_OUT`, as shown in the following code snippet.

```
InteractionDTO myCollaborativeReplyDTO = myInteractionMailService.replyInvitationDTO(
"myChildInvitationID",// Interaction ID
"myPlaceID",
new string[]{"interaction.**:*"}); /// attributes to retrieve

String myCollaborativeReplyID = myCollaborativeReplyDTO.interactionId;
```

The returned `InteractionDTO` contains the identifier of the created interaction. Use this identifier to

fill the e-mail using the `IInteractionService.setInteractionDTO()` method, as detailed in Filling an E-Mail Interaction.
Once the collaborative reply is filled, your application can send it as an outgoing e-mail using the `IInteractionMailService.send()` method, as shown in the following code snippet:

```
myInteractionMailService.send( myCollaborativeReplyID, myQueue); // The Queue (should not be
null)
```

Chat Interactions

The chat interaction service is the `IInteractionChatService` interface defined in the `com.genesyslab.ail.ws.interaction.chat` namespace. To use this service, your application works with classes and enumerations of this namespace, and with the classes and interface of the `com.genesyslab.ail.ws.interaction` namespace.

Introduction

The chat interaction service is designed around a set of actions for managing chat interactions. A chat interaction is a specific object representing chat message exchanges, that is, a chat session between an agent and other parties through a `CHAT` medium.
The chat interaction service manages the following tasks:

- Start and stop a chat session through a chat interaction.
- Send a message during a chat session.
- Transfer a chat interaction.

The chat interaction service performs actions only on chat interactions. The chat interaction service depends on the following other services:

- The event service:
  - Your application can receive events with the event service. Events are essentials in chat management.
- The agent service:
  - Before working with chat interactions, your application must first use the agent service to log in an agent on a chat medium.
  - While the agent is logged on a chat medium, your application can use the `IInteractionChatService` interface to perform actions on chat interactions associated with the chat medium. See The Agent Service.
- The interaction service:
  - To access `IInteractionChatService` attributes, your application uses the `IInteractionService` DTO methods.
  - Some chat attributes are published in events of type `InteractionEvent` which is described in the `IInteractionService` interface. For further information, see The Interaction Service.

Chat Interaction Essentials

The `IInteractionChatService` interface exposes methods and pertinent attributes to let your application manage multiple chat sessions with a multiple simultaneous chat interactions, each of

which is identified by a unique interaction ID.

Each chat interaction follows a sequence of states—for example, beginning in a `NEW` state, transitioning to a `DIALING` state, and moving through other states until its final state is `MARKED_DONE`.

For any particular state, the chat interaction service permits use of only a small subset of its possible actions (available to your application as method calls). For any one chat interaction, your application may apply only one action at the same time.
During a chat session—handled by a particular chat interaction—, your application can use the chat service to send chat messages and manage the chat session.

After the chat interaction service successfully applies an action to a particular chat interaction, the event service can receive an `InteractionEvent` that carries the `interactionId` identifying the corresponding chat interaction, along with a variety of attributes reflecting new state (and other data). To receive `InteractionEvent` events, your application must subscribe to them.

For each incoming chat message, each chat session update, or each successful chat action, the event service may receive a `ChatEvent` event that can propagate the published chat attribute values in order that your application can take them into consideration. To receive `ChatEvent` events, your application must subscribe to them.

For each incoming `InteractionEvent` or `ChatEvent` event, your application should test various attributes of the `IInteractionChatService` interface, including especially the `interaction.chat:actionsPossible` attribute (to determine which actions the application can currently apply).

The following sections present the details behind the above general description.

Chat Interaction Attributes

The `IInteractionChatService` has no methods to read chat attributes. Your application must call one of the `IInteractionService.getInteractionDTO*()` methods. See also Opening a Workbin Interaction.

Chat Attributes

For each chat interaction, the `IInteractionChatService` interface defines a set of attributes that are characteristic of a chat session in the `interaction:chat` domain. The following list is representative (but not exhaustive):

- `interaction.chat:parties`—The parties of the session.
- `interaction.chat:messages`—The list of exchanged messages.
- `interaction.chat:messageEvent`—Published chat message.
- `interaction.chat:duration`—The time duration of the chat session in seconds.
- `interaction.chat:actionsPossible`—Array of chat actions currently possible on a session.

Common Interaction Attributes

For each chat interaction, the attributes of the `IInteractionService` interface are also available. Your application may often use the following:

- `interaction:interactionId`—The system interaction identifier, required in calls to the methods of the chat interaction service.
- `interaction:status`—The interaction status defined with the `InteractionStatus` enumeration.
- `interaction:eventReason`—The `InteractionEventReason` value published when an `InteractionEvent` event occurs for a chat interaction.

Chat Actions

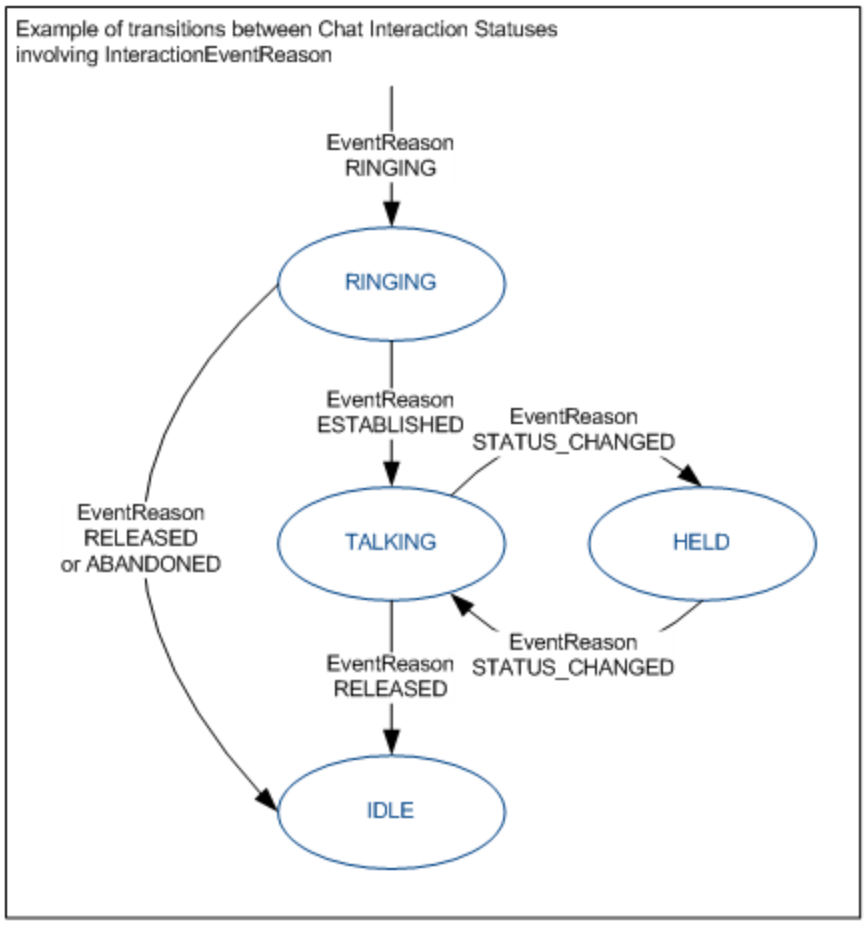
The `InteractionChatAction` enumeration defines voice actions of the `IInteractionChatService` interface. Constants each correspond to one chat interaction service method. For example, the `ANSWER_CALL` constant corresponds to the `IInteractionChatService.answerCall()` method.

> **Important**
> Each method call performs an action on one chat interaction. The method call does not apply to a set of interactions.

Possible actions for chat interactions can be accessed by reading the value of the `interaction.chat:actionsPossible` attribute of the `IInteractionChatService`.
The `IInteractionChatService` has no methods to read attributes. Your application must use one of the `InteractionService.getInteractionsDTO*()` methods. See Handling Interaction DTOs.
Changes in possible actions can be propagated in both `InteractionEvent` and `ChatEvent` events. See Chat Interaction Events.

Chat Interaction Status

The current state of a chat interaction is available as the value of the `interaction:status` attribute, which is defined in the `IInteractionService`. For a chat interaction, possible status values are listed in the `InteractionStatus` enumeration of the `com.genesyslab.ws.interaction` namespace.
The `IInteractionChatService` has no methods to read attributes. Your application must use one of the `InteractionService.getInteractionsDTO*()` methods. See Handling Interaction DTOs.
The status of a chat interaction may change if a successful action is confirmed by an event sent to the server-side application. For example, if your application successfully answers a ringing chat interaction, the status of the chat interaction changes to `InteractionStatus.TALKING`. Changes in interaction status are propagated in `InteractionEvent` events. See Chat Interaction Events.
The following diagram shows the possible interaction statuses for a chat interaction.



Example of Possible Transitions due to InteractionEventReason

> ## Warning
> This figure is provided as an informative example. It does not include all possible statuses and transitions.

Chat Interaction Events

When changes occur on a chat interaction and involve only the chat management of a chat interaction, the event service of your application receives `ChatEvent` events.
To properly take into account chat events, the published `interaction.chat:eventType` attribute value indicates the reason for a chat event. The `ChatEventType` enumeration lists the possible reasons for an occurring `ChatEvent` event.
`ChatEvent` events can propagate any published attribute, that is, any attribute of the `interaction.chat` domains that has the `event` property. The `eventType` attribute indicates attributes to test, as listed in the following table.

Chat Event Types and Attributes

ChatEventType	Attributes	Description
DISCONNECTED	interaction.chat:*	The chat session is terminated.
ERROR_RECEIVED	interaction.chat:*	An error occured.
MESSAGE_RECEIVED	interaction.chat: messageEvent	The messageEvent attribute contain the new incoming chat message.
USER_JOINED	interaction.chat: partyEvent	The partyEvent attribute contains the name of a new party who has joined the session.
USER_LEFT	interaction.chat: partyEvent	The partyEvent attribute contains the name of a party who has left the session.

When changes occur on a chat interaction, the event service of your application may also receive InteractionEvent events. Published attributes are IInteractionService attributes as well as those IInteractionChatService attributes that have the event property. A modification propagated with an InteractionEvent may include changes on that chat interaction's attributes, for example, interaction.chat:actionsPossible.
Your application must subscribe to the TopicsService objects defined for both IInteractionService and IInteractionChatService. Those TopicsService objects have to specify in their TopicsEvents the DTOs to retrieve. Moreover, they must include a trigger on the agent or on the place where the agent is logged in.

> **Important**
>
> For further details on the InteractionEvent mechanism, see Using IInteractionService.

The following code snippet shows how to receive InteractionEvent and ChatEvent events occurring on any chat interaction belonging to agent0.

```
/// Defining two TopicsService
TopicsService[] myTopicsServices = new TopicsService[2] ;

/// Defining a Topic Service for the chat service
myTopicsServices[0] = new TopicsService() ;
myTopicsServices[0].serviceName = "ChatService" ;
/// Defining a topic event
myTopicsServices[0].topicsEvents = new TopicsEvent[1] ;
myTopicsServices[0].topicsEvents[0] = new TopicsEvent() ;
```

```
/// the targeted events are ChatEvents
myTopicsServices[0].topicsEvents[0].eventName = "ChatEvent" ;
/// all the event attributes values are propagated in event objects
myTopicsServices[0].topicsEvents[0].attributes = new String[]{ "interaction.chat:*"};
/// Triggering ChatEvent for agent0
myTopicsServices[0].topicsEvents[0].triggers = new Topic[1];
myTopicsServices[0].topicsEvents[0].triggers[0] = new Topic();
myTopicsServices[0].topicsEvents[0].triggers[0].key = "AGENT";
myTopicsServices[0].topicsEvents[0].triggers[0].value = "agent0";

/// Defining a Topic Service for the interaction service
myTopicsServices[1] = new TopicsService() ;
myTopicsServices[1].serviceName = "InteractionService" ;

myTopicsServices[1].topicsEvents = new TopicsEvent[1] ;
myTopicsServices[1].topicsEvents[0] = new TopicsEvent() ;

/// the targeted events are InteractionEvent
myTopicsServices[1].topicsEvents[0].eventName = "InteractionEvent" ;

/// in case of a chat interaction, the interaction,
/// and chat attributes values are propagated in the Event object
myTopicsServices[1].topicsEvents[0].attributes = new String[]{ "interaction:*",
"interaction.chat:*"};

/// To receive those events for agent0, your application must
/// trigger events on agent0
myTopicsServices[1].topicsEvents[0].triggers = new Topic[1];
myTopicsServices[1].topicsEvents[0].triggers[0] = new Topic()
myTopicsServices[1].topicsEvents[0].triggers[0].key = "AGENT";
myTopicsServices[1].topicsEvents[0].triggers[0].value = "agent0";
```

For further information about events, see The Event Service.

Managing a Chat Session

Your application uses the chat service to manage chat interactions. Each chat interaction corresponds to a chat session.
To deal with chat interactions, your application must log in an agent on a CHAT medium, and must subscribe to interactions and chat events with the event service (see Chat Interaction Events). When subscribing to those events, your application must trigger on the targeted agent or on the place where its agent is logged.
When a ringing chat interaction occurs on the place, your application must answer the chat interaction to start the chat session. Then your application receives events for chat incoming messages and can send messages with the chat interaction service. To leave the session, your application can release the session and then mark it as done.
The following table presents the corresponding actions to perform on a chat interaction.

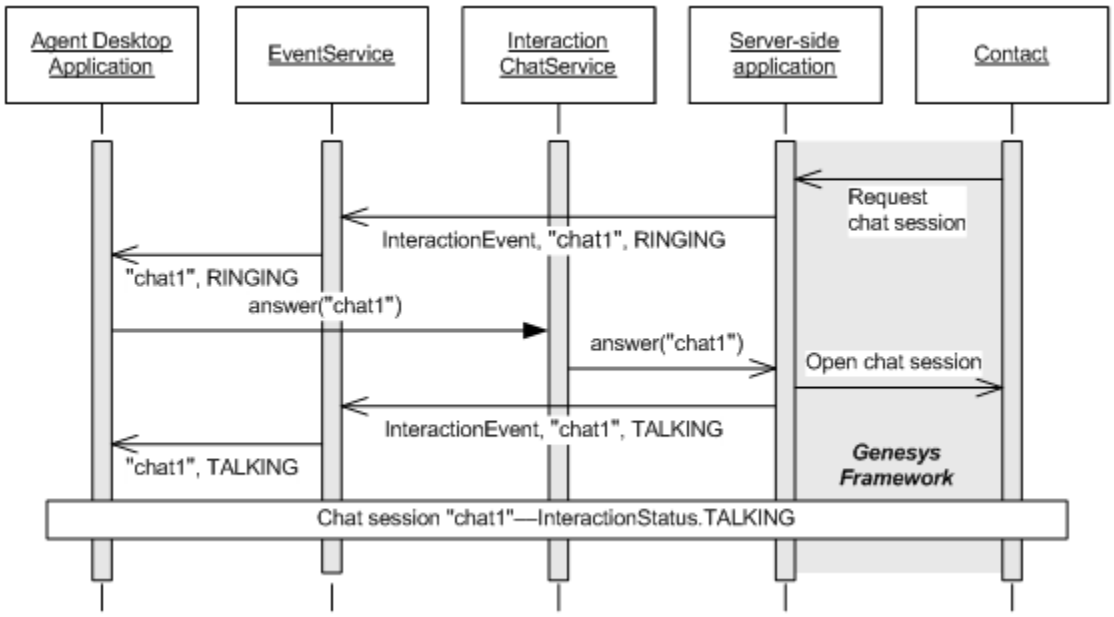
Management Actions for a Chat Interaction

Action	InteractionChatAction	IInteractionChatService Methods
Answer a chat interaction	ANSWER_CALL	answer()
Release a chat interaction	RELEASE_CALL	release()
Mark a chat interaction as done	MARK_DONE	markDone()

The following subsections detail these management steps for a chat session.

Answering a Chat Interaction

When a customer requests a chat session, an agent receives a chat interaction in a RINGING status. If the agent successfully answers this chat interaction, he starts a chat session between himself and the customer.
The following figure presents the corresponding sequence diagram involving the chat and event services of your application.



Answering a Chat Interaction

The InteractionEvent events received by the event service have a STATUS_CHANGED reason, and

their labels indicate the current status of the associated chat interaction.
Retrieve in the received `InteractionEvent` event the interaction ID and the updated possible actions of the chat interaction. Update your application with the possible chat actions.
If the `InteractionChatAction.ANSWER_CALL` is available in the interaction's`interaction.chat:actionsPossible` attribute, call the `IInteractionChatService.answer()` method, as shown in the following code snippet.

```
myInteractionChatService.answer(myInteractionID);
```

Getting Parties

A party is identified by his or her nickname during the chat session. The `ChatParty` class associates the party's information with its nickname. Its fields are the following:

- `nickname`—A party nickname, used as an identifier during the chat session.
- `connected`—`True`, if the party is connected.
- `type`—A string for the type of party.
- `visibility`—The chat party's visibility; a value of the `ChatPartyVisibility` enumeration.

The chat interaction service exposes the parties of a chat session in the `interaction.chat:parties` attribute of the chat interaction that handles the chat session. This attribute value contains an array of `ChatParty` objects. For example, your application can retrieve this attribute value with the `IInteractionService.getInteractionsDTO()` method as shown in the following code snippet.

```
InteractionDTO[] myChatInteractionDTO = myInteractionService.getInteractionsDTO( new
string[]{myInteractionId},
new string[]{"interaction.chat:parties"});

// The DTO array contains one DT which contains one attribute
/// Getting the parties:
KeyValue myAttrKeyValue = myChatInteractionDTO[0].data[0];
ChatParty[] myParties = (ChatParty[]) myAttrKeyValue.value;

/// Displaying information about parties
foreach(ChatParty myParty in myParties)
{
   String connected = "not connected";
   if(myParty.connected)
      connected = "connected";
   System.Console.WriteLine(myParty.nickname +" is " + connected + " (visibility="+
myParty.visibility.ToString()+")\n");
}
```

Sending Chat Messages

A chat message is a simple string to send. To send the message, call the `IInteractionChatService.sendMessage()` method as shown in the following code snippet:

```
myInteractionChatService.sendMessage( myInteractionId,//ID of the chat interaction handling
the session
"My message is sent to all parties"); // message to send
```

Conferences

During a chat session, an agent might invite another agent to join, or an agent or supervisor might want to join. The `IInteractionChatService` interface offers two methods for this purpose:

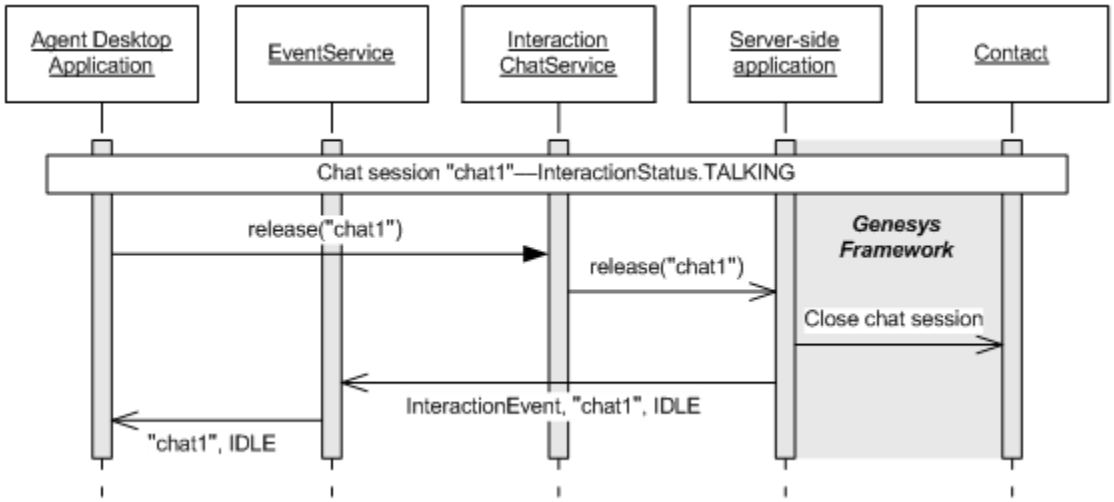
- `conferenceAgent()`—An agent is invited to join by receiving a `RINGING` chat interaction.
- `conferencePlace()`—A `RINGING` chat interaction is sent to a place.

If an application handling the target (agent or place) answers the `RINGING` interaction, your application event service receives a `ChatEvent` event specifying that a user has joined.
Your application specifies what is the visibility of the new invited party in the method call, as shown in the following code snippet.

```
myInteractionChatService.conferenceAgent(
   myInteractionID,// ID of the chat interaction // handling the session to join
   myTargetAgentId, // invited agent
   ChatPartyVisibility.ALL, //visibility if the target agent joins
   "Need information about defects"); // reason for joining
```

Releasing a Chat Interaction

To leave a chat session, your application must release the chat interaction with the `IInteractionChatService.release()` method. If the call is successful, the chat interaction's status changes to `InteractionStatus.IDLE` as shown in Releasing a Chat Interaction.



Releasing a Chat Interaction

To release a chat interaction, call the `IInteractionChatService.release()` method if the `InteractionChatAction.RELEASE_CALL` action belongs to the `interaction.chat:actionsPossible` attribute of the chat interaction:

```
myInteractionChatService.release(myInteractionId);
```

Marking a Chat Interaction as Done

When an agent has finished working with a chat session and has released this chat session, he can mark an interaction as done, so that the interaction is saved in the contact's history.
To mark a chat interaction as done, call the `IInteractionChatService.markDone()` method if the `InteractionChatAction.MARK_DONE` action belongs to the `interaction.chat:actionsPossible` attribute of the chat interaction:

```
myInteractionChatService.markDone(myInteractionId);
```

Transferring a Chat Interaction

Your agent application can transfer a chat interaction to another agent or to another place using the chat interaction service. These transfers are direct, that is, in a single step.
If the `interaction.chat:actionsPossible` attribute of a chat interaction includes the `InteractionChatAction.TRANSFER` action, your application can use one of the chat service transfer methods detailed in the following subsections.

Transferring to an Agent

Your agent application enable the user to enter the employee ID of the agent to whom the chat interaction should be transferred. In this case, use the `IInteractionChatService.transferAgent()` method, as shown in the following code snippet:

```
myInteractionChatService.transferAgent( myInteractionId, myTargetEmployeeId, "A message
justifying the interaction transfer");
```

Transferring to a Place

Your agent application can enable the user to choose a place (having a `CHAT` medium) to which to transfer the chat interaction. In this case, use the `IInteractionChatService.transferPlace()` method, as shown in the following code snippet:

```
myInteractionChatService.transferPlace( myInteractionId, myTargetPlaceId, "A message
justifying the interaction transfer");
```

The Contact Service

The contact service is the `IContactService` interface defined in thecom.genesyslab.ail.ws.contact namespace. It handles the management of contacts.

Introduction

The following sub-sections present the contact service of Agent Interaction Service API.

What Is a Contact?

A contact is a customer with whom the agent may interact through a medium. Each contact has an ID which is a unique system reference used in the Genesys Framework. The *Universal Contact Server* (UCS) stores the contact data—that is, names, e-mail addresses, phone numbers, and other information.
This server also stores the history of a contact—that is, processed interactions. For further details, see The History Service.

What Is the Contact Service?

The contact service is an interface that lets your application access contacts' data using the contacts' IDs. Your application can create, retrieve, and modify a contact's data set.
The contact service is independent from other services. To use the contact service, your application does not require specifics such as a logged in agent.
However, your application can use this service together with other services to fit agents' needs. The contact is important information, used to deal with interactions and callback records. The interaction includes two participants, the agent and the contact.
The following scenarios illustrate uses of the contact service:

- The agent wants to see a contact's phone numbers. Your application displays the list of phone numbers. The agent selects the contact's mobile phone number and your application offers to launch a call.
- Your application receives an `InteractionEvent` for a ringing call.
  - It uses the contact ID to display who is calling the agent.
  - The contact is unknown: As soon as the agent answers the call, the application displays a wizard to collect the contact information.

Contact Information

The `IContactService` interface does not use specific DTO classes to deal with contacts. It uses a set of `ContactXxx` classes defined in the `com.genesyslab.ail.ws.contact` namespace.
The following subsections detail how the `IContactService` interface uses these classes.

Contacts' Attributes

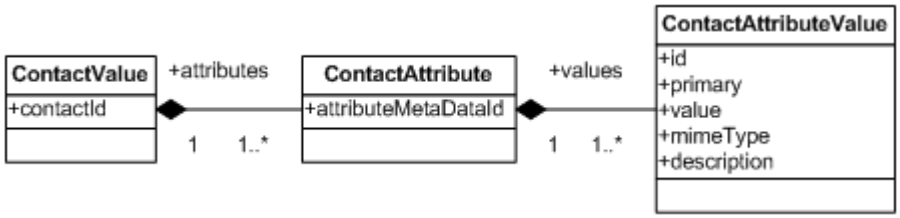
Most of the time, a contact is an instance of the `ContactValue` class of the `com.genesyslab.ail.ws.contact` namespace. The `IContacService` interface lets your application manage `ContactValue` objects containing the contacts' information.
The `ContactValue` class has two attributes:

- `contactId`—The unique system reference for the contact.
- `attributes`—An array of `ContactAttribute` containing the contact's attributes.

The contact service's attributes are characteristic values of a contact. For example, an attribute may be the contact's last name, first name, or a set of e-mail addresses.

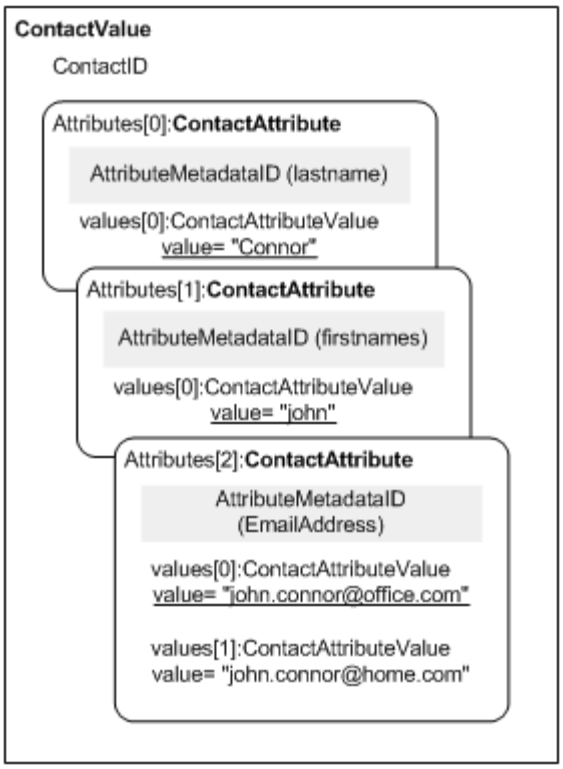
Attribute Values

The following class diagram shows how the contact information is structured.



ContactValue Class Diagram

The `ContactAttribute` objects define the data of a contact. Each `ContactAttribute` contains the existing values for an attribute. The attribute values are available in the `ContactAttribute.values` field, which is an array of `ContactAttributeValue` . Each `ContactAttributeValue` object contains a single attribute value which has a unique system reference available in the `ContactAttributeValue.id` field.
For example, if a contact has one or several e-mail addresses, `EmailAddress` is considered an attribute. A single `ContactAttribute` contains its values—that is, all the contact's e-mail addresses—and each `ContactAttributeValue` contains an e-mail address, as illustrated below.



Example of Contact Information

> **Important**
>
> Each contact attribute is associated with a metadata object. See MetaData in Other Contact Attribute Classes.

Primary Attributes

The primary attribute value of a contact is defined for a contact attribute. It is one of the attribute values marked as primary.
For example, if a contact has several e-mail addresses, the work e-mail address might be the primary e-mail attribute. In the above example, the primary attribute is underlined for each type of attribute.
To retrieve primary attributes, see Retrieving Contact Information.

> **Important**
>
> There is only one primary attribute value per attribute's type.

MetaData

The Universal Contact Server defines a metadata for each type of attribute. For example, the last name is a type of contact attribute specified by a metadata. For the last name attribute, the metadata specifies that the attribute name is `LastName`, the type of the attribute value is a string, the display name is `Last name`, and so on.
A single metadata is available for each type of attribute; this metadata has a unique system identifier and a unique name. For example, a single metadata is available for all the existing last names' attribute values. The metadata is independent from the contacts' attribute values.
The Agent Interaction SDK (Web Services) lets your application access the metadata information with classes detailed in the following subsections.

ContactAttributeMetadata

Class

The `ContactAttributeMetadata` class defines the attribute metadata. The following are some of the main `ContactAttributeMetaData` fields:

- `id`—The unique system identifier for this metadata.
- `name`—The unique attribute name.
- `active`—`true` if the attribute is active in the Contact Server.
- `displayName`—The attribute display name.
- `predefinedValues`—A list of predefined contact attribute values, or `null`.
- `searchable`—`true` if the attribute is searchable. For example, a last name might be searchable, whereas a title, such as `Mr.` or `Ms`, might not be searchable.
- `sortable`—`true` if the attribute can be used to sort the contacts. For example, a last name might be sortable.
- `type`—The type of attribute defined by `ContactAttributeMetaDataType.` (See below).

> **Important**
> To retrieve attributes, the metadata identifier id is required.

Type

The `ContactAttributeMetaDataType` enumeration defines the type for the values specified by a metadata:

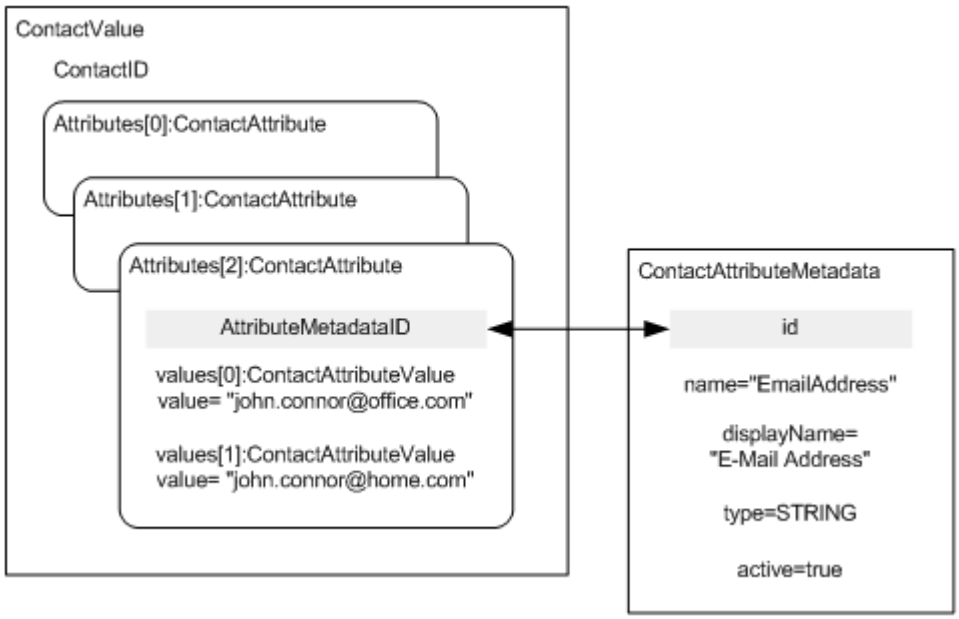
- `ContactAttributeMetaDataType.BINARY`—Type for binary contact attribute values.
- `ContactAttributeMetaDataType.STRING`—Type for string contact attribute values.
- `ContactAttributeMetaDataType.DATE`—Type for date contact attribute values.

MetaData in Other Contact Attribute Classes

Each `ContactAttribute` object associates an array of `ContactAttributeValue` values with a metadata:

- The `ContactAttribute.attributeMetaDataId` field is the ID of the available metadata for the set of attribute values.
- For each `ContactAttributeValue`, the `name` field points out the name of the metadata.

The example below presents a `ContactAttributeMetaData` related to a `ContactAttribute` and its array of `ContactAttributeValue`.



Example of ContactAttributeMetaData and ContactAttribute

Predefined MetaData

Some contact attributes are fixed and the associated metadata are predefined.
The `ContactAttributeMetaDataPredefinedType` enumerated type lists the existing predefined attributes handled by the contact service:

- `TITLE`—Fixed contact attribute for the title.
- `FIRSTNAME`—Fixed contact attribute for the first name.
- `LASTNAME`—Fixed contact attribute for last names.
- `PHONE_NUMBER`—Fixed contact attribute for phone numbers.
- `EMAIL_ADDRESS`—Fixed contact attribute for e-mail addresses.

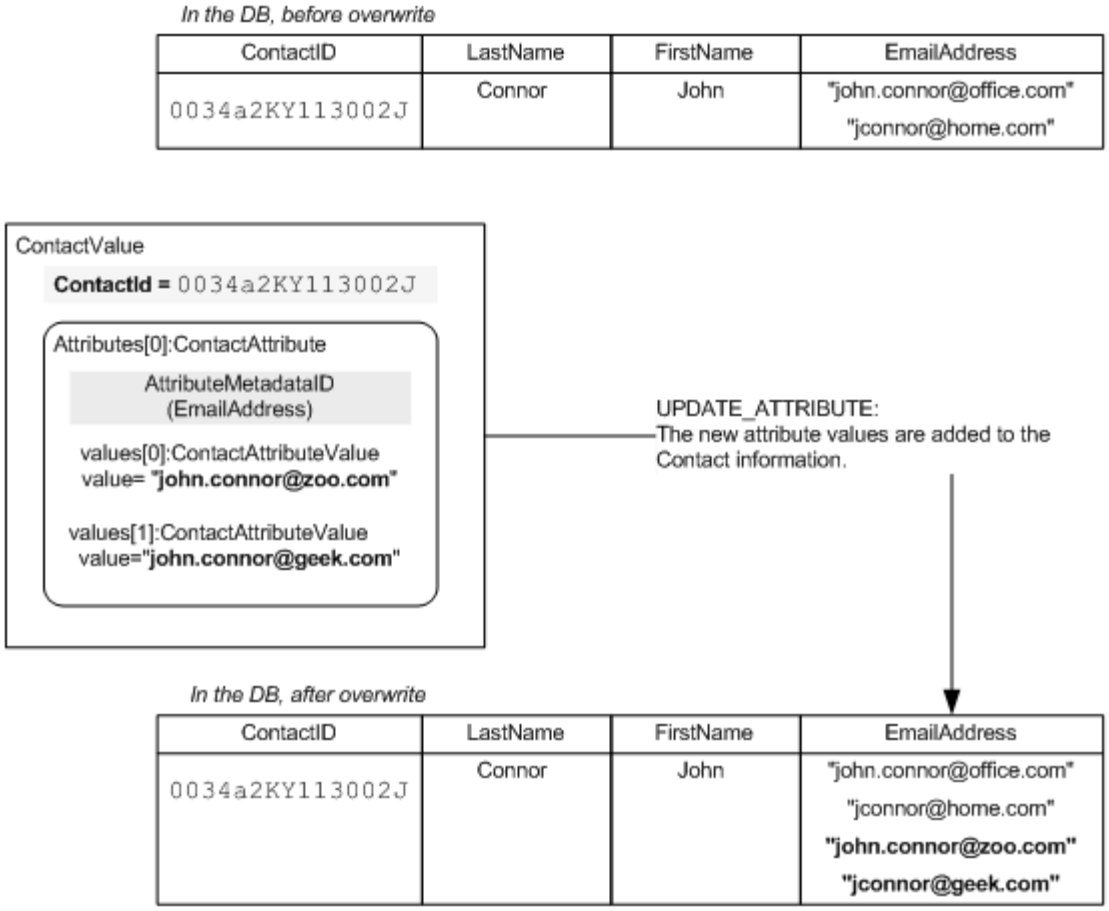
The `ContactAttributeMetaDataPredefined` class associates a `ContactAttributeMetaDataPredefinedType` with a `ContactAttributeMetaData` object. Use these objects to get fixed `ContactAttributeMetadata` without the name or the identifier of these metadata. See Getting Predefined MetaData.

Information Update

The`com.genesyslab.ail.ws.contact.ContactUpdateType` enumeration corresponds to the types of update that the contact service can perform on the information of several contacts at the same time.
For each contact to update, your application builds a `ContactValue` object. The content of the array of `ContactValue` is used to update the contacts' information in the database according to the type of update. The following sub-sections detail these update types. For implementation, refer to Setting Attribute Values.

Attributes Update

When the agent using your application wants to fill in some contact information or add additional values for a contact attribute, your application can use the `ContactUpdateType.UPDATE_ATTRIBUTE` mode. For example, your application might add new e-mail addresses for a customer, as shown below.

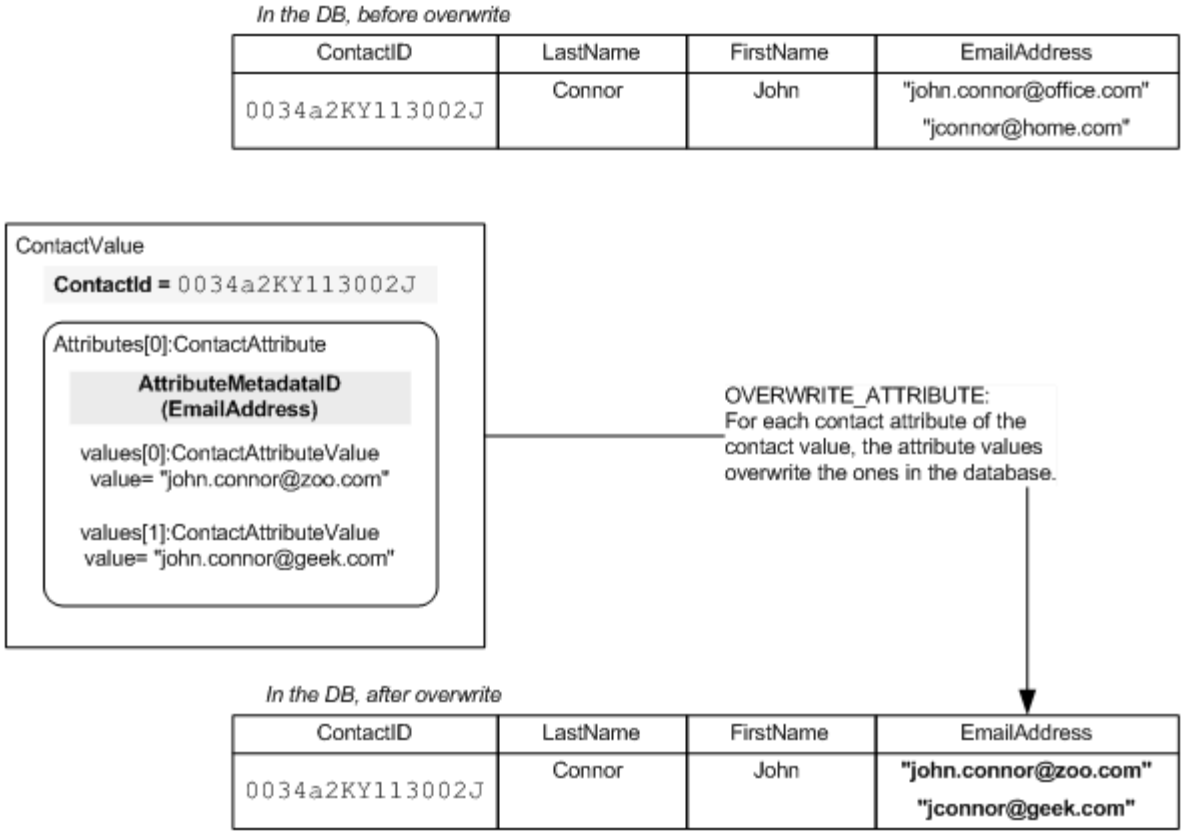


Updating Contact Attribute Values

In Updating Contact Attribute Values, the application defines a `ContactValue` object for the contact John Connor. This `ContactValue` instance has an `Email Address ContactAttribute` containing two `ContactAttributeValue` s, which define new e-mail addresses. As a result of the update, the new addresses are added to the database. The other contact information has not been affected by the update, and the former e-mail addresses remain.

Overwrite Attributes

When the agent using your application wants to modify the existing values for a contact attribute, your application can use the `ContactUpdateType.OVERWRITE_ATTRIBUTE` mode. For example, if a contact's e-mail addresses are obsolete, your application has to replace them with new e-mail addresses, as shown below.

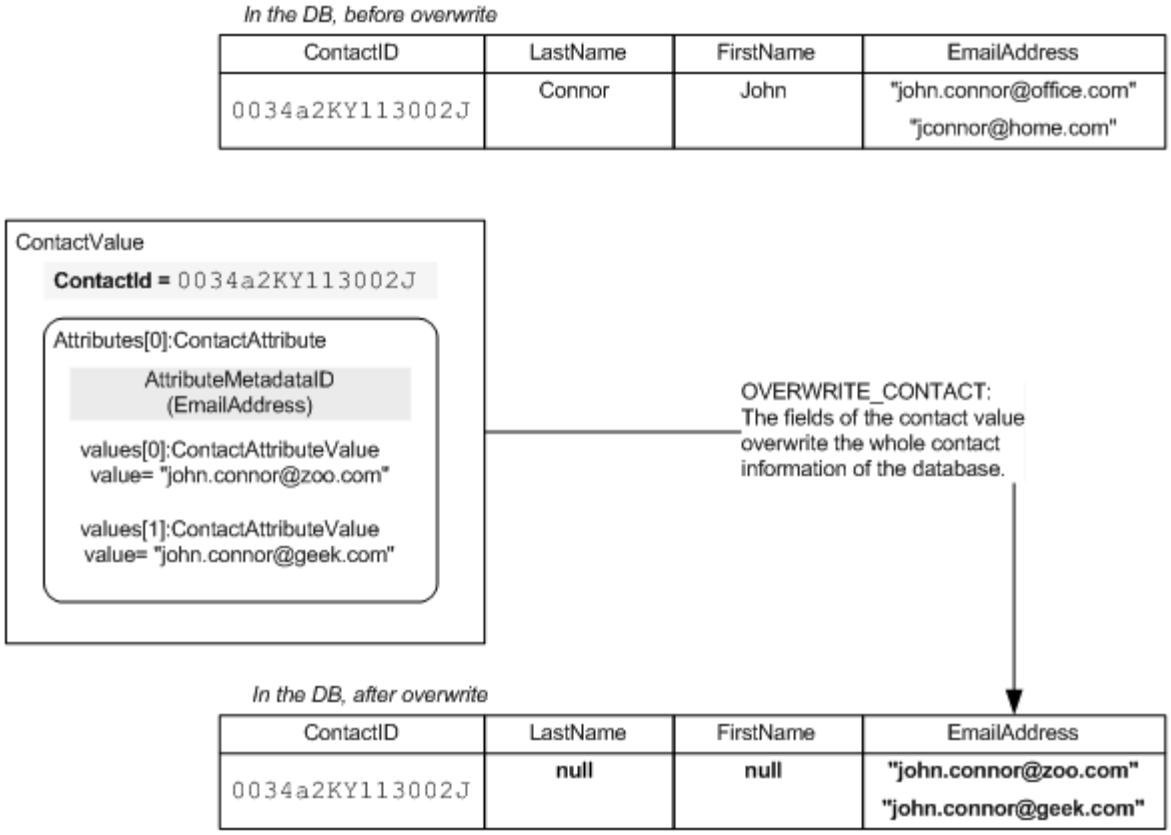


Overwrite of some Contact Attributes

In Overwrite of some Contact Attributes, the application defines a `ContactValue` object for the contact John Connor. This `ContactValue` instance contains an `EmailAddress` `ContactAttribute` containing two `ContactAttributeValue` s which define the available e-mail addresses. As a result of the update, the available e-mail addresses replace the former e-mail addresses in the database. The other types of contact attributes have not been modified by the update. Only the specified `ContactAttribute` objects of the `ContactValue`.attributes array overwrite the contact attributes in the database.

Overwrite a Contact

When the agent using your application wants to change all the contact information, your application can use the `ContactUpdateType.OVERWRITE_CONTACT` mode. To properly overwrite the contact, define all the attributes that must remain after the update. The diagram below shows this type of overwrite with the same `ContactValue` than in previous figures (Updating Contact Attribute Values and Overwrite of some Contact Attributes).



Overwrite of all the Contact Information

In Overwrite of all the Contact Information, the application defines a `ContactValue` object for the contact John Connor. This `ContactValue` instance only contains an `EmailAddress` `ContactAttribute` containing two `ContactAttributeValue` s which define the available e-mail addresses. As a result of the update, the available e-mail addresses replace the former e-mail addresses in the database and the last name and first name information is removed (since no ContactAttribute object defines a new value for them).

Retrieving Contact Information

The `IContactService` interface uses several other container classes in methods calls to manage contacts. The following table summarizes the relationship between container classes and methods.

Contacts, Methods, and Containers of the IContactService

IContactService Methods	Container classes	Description
createContacts()	ContactAttributeCreate	This method uses the ContactAttribute of the container to create a new contact.
mergeContacts()	ContactMergeForm	This method merges the contact pairs of each container instance.
removeContactAttributes()	ContactAttributeRemove	This method removes only contact attribute values specified in the container.
searchContacts()	ContactSearchTemplate	This method uses the container to restrict the contact search.

Later sections of this chapter provide details about the operations listed above.

Retrieving Contact MetaData

Your application must retrieve the ContactAtributeMetaData objects of the required attributes. The IDs of these objects are necessary to retrieve the contact attribute information. The following subsection provides you with two ways of getting metadata.

Getting Predefined MetaData

Predefined metadata are the metadata of the fixed attributes defined in the UCS. The IContactService.getContactAttributePredefinedMetaData() method lets your application retrieve all the existing ContactAttributePredefinedMetaData objects as shown in the following code snippet.

```
/// Retrieving the predefined metadata
ContactAttributePredefinedMetaData[] myPredefMetaData =
myContactService.getContactAttributePredefinedMetaData();

/// Displaying the name of each predefined metadata
foreach(ContactAttributePredefinedMetaData predef in myPredefMetaData)
{
   System.Console.WriteLine("Predefined Attribute: " + predef.predefinedType
   +"MetaData name: "
   + predef.metaData.name);
}
```

> **Important**
> Retrieving predefined metadata does not require any metadata name or identifier.

Getting MetaData

To retrieve the metadata, the `IContactService` interface offers the two following methods:

- `getContactAttributeMetaDataByName()` retrieves the metadata corresponding to the names specified in the array passed as parameter.
- `getContactAttributeMetaDataById()` retrieves the metadata corresponding to the IDs specified in an array passed as parameter.

Retrieving Contact Values

When your application needs to retrieve contact values, it must specify which attributes must be retrieved. Therefore, to retrieve contact values, the `IContactService` interface needs:

- The list of contact IDs involved in the information retrieval.
- The types of attributes to retrieve for each contact.

Setting a List of Attributes to Retrieve

First, create an array of `ContactAttributeRetrieve` objects. Each `ContactAttributeRetrieve` object sets:

- `metaDataId`—The ID of the metadata corresponding to a type of attribute to retrieve.
- `primary`—If `true`, only the primary values of this type of contact attribute are retrieved; otherwise, all the values of this attribute type are retrieved for each contact. For further information, see Primary Attributes.

```
ContactRetrieveAttribute[] toRetrieve = new ContactRetrieveAttribute[2];

/// For each contact, retrieve the last name of the contact
toRetrieve[0] = new ContactRetrieveAttribute();
toRetrieve[0].primary = true;
toRetrieve[0].attributeMetaDataId = myLastNameMetaData.id;

/// For each contact, retrieve all the e-mail addresses /// of the contact
toRetrieve[1] = new ContactRetrieveAttribute();
toRetrieve[1].primary = false;
toRetrieve[1].attributeMetaDataId = myEMailAddressMetaData.id;
```

Retrieving the Contacts' Values

To retrieve the contacts values, use the previously defined array of `ContactAttributeRetrieve` objects to call the `IContactService.getContacts()` method, as shown in the following code snippet.

```
/// Retrieving attributes of myContactId
ContactValue[] myContacts = myContactService.getContacts( new string[]{myContactId},
toRetrieve);

/// Displaying the retrieved values
foreach(ContactValue myContactValue in myContacts)
{
   System.Console.WriteLine("**********");
   foreach(ContactAttribute myContactAtt in myContactValue.attributes)
   {
      /// Displaying the contact attribute values
      System.Console.WriteLine("Attribute Values: ");
      foreach(ContactAttributeValue myValue in myContactAtt.values)
      {
          System.Console.WriteLine(myValue.value.ToString()+" ");
      }
   }
}
```

Note that the above code snippet displays all the contents of the retrieved contacts' values.

Searching Contacts

The contact service includes an advanced search feature for contacts. With the `IContactService` interface, your application can search contacts according to several attributes' values and their associated metadata IDs.
The following sections detail how to build a contact filter tree—equivalent to a search request—and how to implement the contact service's search feature.

Contact Filter Trees

The`com.genesyslab.ail.ws.contact` namespace contains classes and enumerations to build filter trees corresponding to search requests that approximate SQL requests to the UCS. Those filter trees are equivalent to arithmetic expressions.
Wildcards are autorized in leaves to facilitate the search:

- The * wildcard matches any string of zero or more characters,
- The `\*` sequence matches one character.

For example, `((LastName="B*" and FirstName="A*") or (EMailAddress="ab*@company.com"))` searches for any contact whose first name begins with an `A` and whose last name begins with a`B`, or for any contact whose e-mail address begins with `ab` and finishes with `@company.com`.
To build a filter tree, your application must define filter nodes and filter leaves, as detailed in the following subsections.

Filter Leaves

A filter leaf contains a terminal expression that defines a search value for a contact attribute, such as: `LastName="B*"`. Your application can create a filter leaf with an instance of the

ContactFilterLeaf class, which associates a metadata ID with a contact attribute value. The ContactFilterLeaf class contains the following fields:

- attributeMetaDataId—the metadata ID of a contact attribute.
- operator—a ContactFilterLeafOperator value, which can be EQUAL or NOT_EQUAL.
- value—an attribute value.
- primary—boolean; if true, the leaf defines a search restricted to the primary values of a contact attribute.

The following code snippet implements a ContactFilterLeaf object for the LastName=”B*” expression:

```
ContactFilterLeaf myLeaf = new ContactFilterLeaf();
myLeaf.attributeMetaDataId = "my LastName Metadata ID";
myLeaf.@operator = ContactFilterLeafOperator.EQUAL;
myLeaf.value = "B*";
myLeaf.primaryOnly = true;
```

Filter Nodes

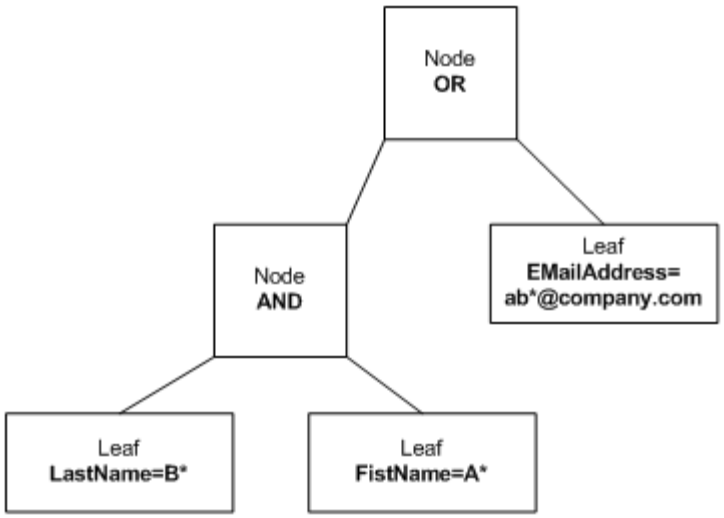
A filter node contains a non terminal expression that defines an operation for several non-terminal (node) or non terminal (leaf) expressions, as in the following examples:

- a or b or c
- a and b and c
- a or b
- a and b

In the preceding examples, a, b, and c can be other filter nodes or leaves, and the terms or and and are operators.
Your application can create a filter node by creating an instance of the ContactFilterNode class. This class contains the following files:

- operator—A ContactFilterNodeOperator value which can be AND or OR.
- leaves—An array of ContactFilterLeaf objects affected by the operator.
- nodes—An array of ContactFilterLeaf objects affected by the operator.

The following example presents a filter node containing the following expression: (LastName="B*" and FirstName=”A*”) or (EMailAddress="ab*@company.com")



An Example of Filter Node

Here is the corresponding code snippet:

```
/// The lower filter node defines: LastName="B*" AND FirstName="A*"
/// Creating a contact filter node
ContactFilterNode myLowerNode = new ContactFilterNode();
myLowerNode.@operator = ContactFilterNodeOperator.AND;

/// This node applies an AND operation to two leaves
myLowerNode.leaves = new ContactFilterLeaf[2];

/// Defining a leaf for LastName="B*"
myLowerNode.leaves[0] = new ContactFilterLeaf();
myLowerNode.leaves[0].attributeMetaDataId = "my Last Name Metadata ID";
myLowerNode.leaves[0].@operator = ContactFilterLeafOperator.EQUAL;
myLowerNode.leaves[0].value = "B*";
myLowerNode.leaves[0].primaryOnly = true;

/// Defining a leaf for FirstName="A*"
myLowerNode.leaves[1] = new ContactFilterLeaf();
myLowerNode.leaves[1].attributeMetaDataId = "my Fist Name Metadata ID";
myLowerNode.leaves[1].@operator = ContactFilterLeafOperator.EQUAL;
myLowerNode.leaves[1].value = "A*";
myLowerNode.leaves[1].primaryOnly = true;

/// The upper filter node defines:
/// myLowerNode OR (EMailAddress=”ab*@company.com”)

/// Creating a contact filter node
ContactFilterNode myUpperNode = new ContactFilterNode();
myUpperNode.@operator = ContactFilterNodeOperator.OR;

/// This node applies an OR operation to a node and a leaf
/// Adding myLowerNode to the nodes of myUpperNode
myUpperNode.nodes = new ContactFilterNode[1];
myUpperNode.nodes[0] = myLowerNode;
```

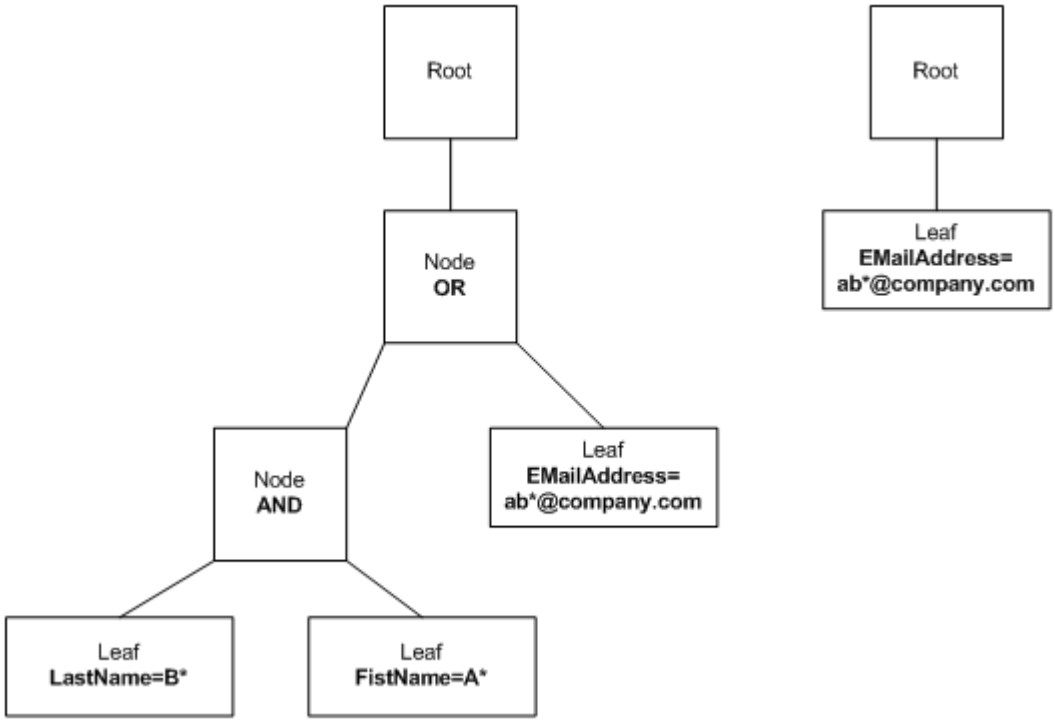
```
/// Defining a leaf for (EMailAddress="ab*@company.com")
myUpperNode.leaves = new ContactFilterLeaf[1];
myUpperNode.leaves[0] = new ContactFilterLeaf();
myUpperNode.leaves[0].attributeMetaDataId = "my EMail Address Metadata ID";
myUpperNode.leaves[0].@operator = ContactFilterLeafOperator.EQUAL;
myUpperNode.leaves[0].value = "ab*@company.com";
myUpperNode.leaves[0].primaryOnly = false;
```

Filter Root

A filter root is the entry point of a filter tree. Your application can define a filter tree with the `ContactFilterRoot` class, which contains the following fields:

- `node`—The root ContactFilterLeaf node of a tree.
- `leaf`—The root and single ContactFilterLeaf leaf of a tree.

Although the class has a node and a leaf field, your application must define the leaf or the node field, but not both. The following figure illustrates this, showing the two types of possible trees that a `ContactFilterRoot` instance defines.



Filter Roots

The left tree shows a filter root containing a filter node, and the right tree shows a filter root containing a filter leaf. Those two trees could not be merged into a single tree whose root would contains a filter node and a filter leaf.
The following code snippet shows how to create a `ContactFilterRoot` object for the left tree of

Filter Roots diagram.

```
// Creating a contact filter root
ContactFilterRoot myFilterRoot = new ContactFilterRoot();

// Setting its node field with the upper node of the filter tree
myFilterRoot.node = myUpperNode;
```

Contact Search

To search a contact with the contact service, your application has to:

- Create a filter tree, as detailed in Contact Filter Trees.
- Fill in a contact search template.
- Call the `searchContacts()` method of the `IContactService` interface.

A contact search template delimits the contact search result associated with a filter tree. Your application can set the number of contacts to retrieve, and which attributes to retrieve, in the search call. This is useful for (as one example) displaying a contact search by pages, with a limited set of attributes.
To fill in a contact search template, your application must create an instance of the `ContactSearchTemplate` class, which defines the following fields:

- `filter`—A contact filter tree defining the search request.
- `index`—The index of the first `ContactValue` to retrieve in the list of matching `ContactValue` objects.
- `length`—The length of the list of `ContactValue` objects returned by the `searchContact()` method call.
- `retrieveAttributes`—The array of contact attributes to retrieve. For further information, see Setting a List of Attributes to Retrieve.
- `sortAttributes`—The array of contact attributes to sort.

The call to the `IContactService.searchContacts()` method returns an array of `ContactValue` objects matching the specifications of the search template that was passed as a parameter.
The following code snippet implements a simple search for the contact filter root example on Filter Roots.

```
/// Defining a contact search template
ContactSearchTemplate mySearchTemplate= new ContactSearchTemplate();
mySearchTemplate.filter = myFilterRoot;

// search of the ten first results
mySearchTemplate.index=0;
mySearchTemplate.length=10;

//Request a search for this template
ContactValue[] myFoundContacts = myContactService.searchContacts(mySearchTemplate);
```

Tuning the Contact Search

The definition of this ContactSearchTemplate form is very important because it determines the

processing times of your search requests. There are several aspects to take into account to fit your application needs and fine-tune your search requests.

Number of Attributes

The number of attributes used in the filter tree to refine your search impacts the request's processing time. The more attributes you set up, the finer your search is, but the longer the request takes.
Additionally, if you set up a wide search with few attributes or vague values, that returns a large collection of instances. This increases the processing time as well, because it impacts the network activity. The problem is similar if you set up a great number of attributes to be returned with each matching instance: the more instances that match, the more data will slow down the network activity.

is-searchable

Genesys recommends that your application uses attributes marked as `is-searchable`. This ensures calls to the appropriate UCS search algorithms, and gets responses with good performance.
To set up `is-searchable` attributes, open the targeted `Attribute Value` object in the `Contact Attributes` list of your Configuration Manager. In the `Annex` tab of the attribute object, open `settings` and set to `true` the `is-searchable` option.
In the Agent Interaction Services API, call the`isSearchable()` method of the `ContactAttributeMetadata` instance to determine whether the associated attribute values are searchable.
If your application searches for any attributes, regardless of whether they are marked as searchable, this will be time-consuming, and will slow down your application. In particular, if your application is a server, this is inappropriate, and detracts from good performances.

SearchPrimaryValueOnly flag

If you set the `primaryOnly` flag to `true` by calling the `FilterLeaf.setPrimaryOnly()` method, you restrict the search to the attributes' primary values. The more you search for primary values, the less the SQL request is being complex, and thus, UCS requires less time to return a result.
Also, for the quickest search, set the `SearchPrimaryValueOnly` flag to `true` by calling the `SearchContactTemplate.setSearchPrimaryValueOnly()` method. The search is restricted to attributes' primary values, regardless of the definition of `FilterLeaf` objects (which are part of the filter tree).

Managing Contacts

The `IContactService` includes features to manage a set of contact data. The following sections detail the most commonly used features.

Creating a Contact

The `IContactService` provides you with the `createContact()` method. For each contact to create, fill a `ContactAttributeCreate` object with `ContactAttribute` objects (containing, for

each type of attribute, the set of corresponding values).

```
/// Creating a contactAttributeCreate for john connor
ContactAttributeCreate myNewContact = new ContactAttributeCreate();
/// The contact is created with 2 attributes only:
/// last name + e-mail addresses
myNewContact.attributes = new ContactAttribute[2];

/// Filling the contact attribute for lastname
myNewContact.attributes[0]=new ContactAttribute();
myNewContact.attributes[0].attributeMetaDataId = myLastNameMetaData.id;
// Creating a single attribute value for the last name
myNewContact.attributes[0].values = new ContactAttributeValue[1];
myNewContact.attributes[0].values[0] = new ContactAttributeValue();
myNewContact.attributes[0].values[0].value = "Connor";
myNewContact.attributes[0].values[0].primary = true;

/// Filling the contact attribute for the e-mail addresses
myNewContact.attributes[1]=new ContactAttribute();
myNewContact.attributes[1].attributeMetaDataId = myEMailAddressMetaData.id;
/// Creating two attribute values for the e-mail
myNewContact.attributes[1].values = new ContactAttributeValue[2];
// Creating the attribute value for the primary e-mail address
myNewContact.attributes[1].values[0] = new ContactAttributeValue();
myNewContact.attributes[1].values[0].value = "John.Connor@company.com";
myNewContact.attributes[1].values[0].primary = true;
// Creating the attribute value for another e-mail address
myNewContact.attributes[1].values[1] = new ContactAttributeValue();
myNewContact.attributes[1].values[1].value = "jconnor@home.com";
myNewContact.attributes[1].values[1].primary = false;
```

The `createContact()` method returns an array of `ContactResult` objects as shown in the following code snippet.

```
ContactResult[] myResults = myContactService.createContacts( new
ContactAttributeCreate[]{myNewContact}, true);
```

The `ContactResult` class contains the result of a contact creation:

- `contactId`—The unique system identifier created for the new contact; `null` if not created.
- `contactError`—The string for the contact error if an error occurred during the contact creation; otherwise `null`.
- `attributes`—The `ContactAttribute` array of successfully created attributes; all the `ContactAttributeValue.id` fields have been created.
- `attributesErrors`—The array of `AttributeErrors` containing the metadata IDs, and the corresponding error for each attribute not created.

The following code snippet displays the returned `ContactResult` array:

```
/// Displaying the ContactResult array returned /// in the previous code snippet
foreach (ContactResult myContactResult in myResults)
{
    // the contact has not been created
    if(myContactResult.contactError != null)
    {
```

```
       System.Console.WriteLine("Error at contact creation: " +
myContactResult.contactError);
    }
    /// The contact is created
    else
    {
       // Displaying the retrieved contact id
       System.Console.WriteLine("Contact successfully created: " +
myContactResult.contactId);
       // Displaying the created attributes
       System.Console.WriteLine("* Attributes successfully created: ");
       foreach( ContactAttribute attribute in myContactResult.attributes )
       {
          if(attribute.attributeMetaDataId == myLastNameMetaDataId)
              System.Console.WriteLine(" Last name");
          else if(attribute.attributeMetaDataId == myEMailMetaDataId)
              System.Console.WriteLine(" E-Mail Address");
       }
       // Displaying which attributes were not created
       System.Console.WriteLine("* Attributes with error at creation: ");
       foreach( AttributeError myError in myContactResult.attributeErrors )
       {
           if(myError.attribute == myLastNameMetaDataId)
             System.Console.WriteLine(" Last name: "+myError.error);
           else if(myError.attribute == myEMailMetaDataId)
             System.Console.WriteLine(" E-Mail Address:" +myError.error);
       }
   }
}
```

Merging Contacts

If an agent finds out that two contacts are the same person, he or she might want to merge those contacts to avoid information duplication.
The `IContactService` interface offers a `mergeContact()` method that merges two contacts. The contacts' identifiers are passed as parameters: the information of the *from* contact is copied to the *To* contact. The *from* contact identifier disappears, and the remaining contact identifier is the *To* ID.
Your application can merge several pairs of contacts in a single call to the `IContactService.mergeContact()` method. For each pair of contacts to merge, your application fills in a `ContactMergeForm` that contains the two contacts' identifiers.
The following code snippet merges two contacts.

```
/// Creating an array of merge form
ContactMergeForm[] myMergeForms= new ContactMergeForm[1];

/// Filling a form of the array
myMergeForms[0] = new ContactMergeForm();
myMergeForms[0].contactIdFrom=myContactIdFrom;
myMergeForms[0].contactIdTo=myContactIdTo;

/// Merging each contact pair of the forms
myContactService.mergeContacts(myMergeForms);
```

Setting Attribute Values

The following table summarizes the three ways of updating contact attributes with the contact service. See also Information Update.

Updating Contacts

ContactUpdateType	Description
UPDATE_ATTRIBUTE	Adds new values to contacts attributes
OVERWRITE_ATTRIBUTE	Overwrites the attribute values of a contact with new attribute values.
OVERWRITE_CONTACT	Overwrites all the contacts attribute.

Your application can set attribute values only for a contact that exists in the UCS. Therefore, your application needs a contact's identifier in order to modify that contact's attributes.
For each contact to update, whatever the type of update, your application must:
Create a ContactValue object.

1. Set the `ContactValue.contactId` field with the contact's identifier.
2. Create an array of `ContactAttribute` objects, and for each `ContactAttribute` object:
   - Set the corresponding `ContactAttributeMetaData` identifier.
   - Fill in the `ContactAttribute.values` array with `ContactAttributeValues` containing the new contact attribute values to input.

The following code snippet illustrates these steps.

```
// Creating a ContactValue array having 1 ContactValue
ContactValue[] datas = new ContactValue[1];
// Creating a ContactValue object for myContact
datas[0]=new ContactValue();
datas[0].contactId = myContactId;

// The update consists in adding an email address to the contact
// Creating a Single ContactAttribute
datas[0].attributes = new ContactAttribute[1];
datas[0].attributes[0] = new ContactAttribute();

//Setting the metadata id of the email metadata
datas[0].attributes[0].attributeMetaDataId = myEMailMetaData.id;

//Creating
datas[0].attributes[0].values = new ContactAttributeValue[1];
datas[0].attributes[0].values[0] = new ContactAttributeValue();
datas[0].attributes[0].values[0].value = "contact@new.email.com";
datas[0].attributes[0].values[0].primary = false;

myContactService.setContactAttributes(datas, ContactUpdateType.UPDATE_ATTRIBUTE, false);
```

Removing Attribute Values

The `IContactService.removeContactAttributes()` method uses the `ContactAttributeRemove` and `ContactAttributeValueRemove` container classes.
Each `ContactAttributeRemove` object contains:

- The ID of the contact involved in the removal of some attribute values.
- An array of `ContactAttributeValueRemove` objects; each`ContactAttributeValueRemove` object contains:
  - The ID of an attribute value to remove.
  - The ID of the corresponding attribute metadata.

The following code snippet shows an example of a call to the `removeContactAttributes()` method.

```
///Creating a ContactAttributeRemove object
ContactAttributeRemove toRemove = new ContactAttributeRemove();

// Setting the id of the contact
toRemove.contactId = myContactId;
toRemove.attributes = new ContactAttributeValueRemove[1];

// Creating a ContactAttributeValueRemove object to // remove an e-mail address of the contact
toRemove.attributes[0] = new ContactAttributeValueRemove();

// Setting the metadata id for the e-mail address
toRemove.attributes[0].attributeMetaDataId = myEMailMetaDataId;

// Setting the ID of the e-mail address to remove
toRemove.attributes[0].attributeValueIds = new string[]{ emailValueIdToRemove };

// Calling the remove method of the contact service.
myContactService.removeContactAttributes( new ContactAttributeRemove[]{toRemove});
```

The Callback Service

The callback service is the `ICallbackService` interface defined in thecom.genesyslab.ail.ws.callback namespace. It covers the management of callback records.

Introduction

The callback service is a feature that depends on the Genesys voice callback server. If your call-center includes a callback server, customers can request a callback as soon as possible or at a specific calendar time. The voice callback server records the request and puts it in an appropriate queue.
The callback service deals with the records—customers' callback requests—managed by the voice callback server. Each record contains the information required to call back the customer.
If an available agent using your application is logged in, he cans receive a callback record event and an interaction event. The agent can accept the record and process the callback with a voice interaction.
The callback service is designed to let your application perform the following agent actions on records:

- Accept or reject a record.
- Mark a record as processed when the callback is done.
- Reject or reschedule a record.
- Access a record's information.

The callback service cannot be used independently from:

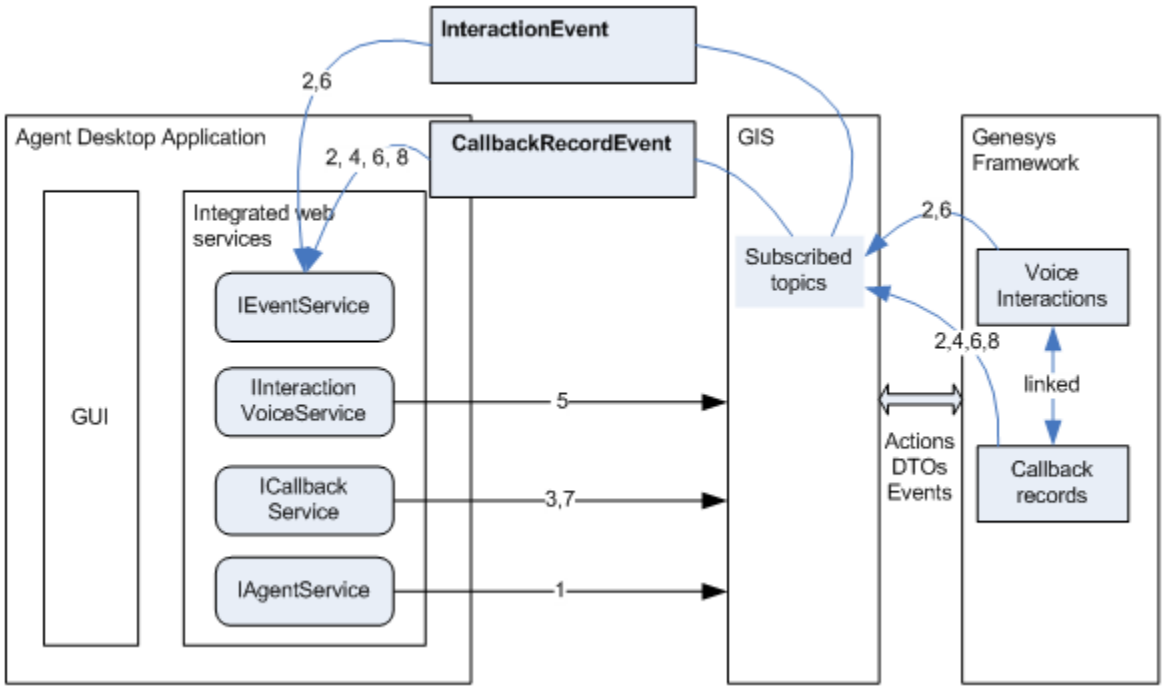
- The agent service—This service lets your application log an agent in on a voice media (DN) to process the record.
- The voice interaction service—This service deals with the voice interactions that process the records.
- The event service—This service required to receive voice and record events.

The following sections details the callback service.

Callback Essentials

The callback service of your agent application processes callback records from the point of view of a calling agent. Therefore, it manages callback records as data attached to a voice interaction.
When an agent gets a callback record from the OCS (Outbound Contact Server), he also gets a voice interaction to dial the callback. An agent can either accept or reject a callback record. An agent processes a callback as a normal phone call with this voice interaction. The management of this voice interaction does not differ from the management described in Voice Interactions.
The data of a callback record includes the ID of the corresponding voice interaction. Once the call is

terminated, the agent can mark the callback record as processed.
The management of a callback voice interaction correlates to the management of its callback outbound record. Your application should use the voice interaction service to deal with all voice-specific aspects of a callback voice interaction.
The following figure shows how your application should integrate the callback service to handle records.



Callback Integration Example

Login an agent on a voice media.

1. Record and interaction events for a callback to process.
2. The logged-in agent accepts the record.
3. Receiving a record event for the record status change.
4. Actions on the voice interaction to process the callback.
5. Receiving record and interaction events due to 5.
6. Marking the record as processed when call is IDLE.
7. Receiving record event due to 7.

The Callback Integration Example shows the general callback service handling in an application that lets an agent accept and process records. The `ICallbackService` interface exposes methods and pertinent attributes that your application uses to manage record's callback data, such as marking a call as processed or rescheduling a call.
The `IInteractionVoiceService` interface manages actions that are restricted to a voice interaction

used to process a callback record (for example, dialing a call, releasing a call, or marking a call as done). For any particular state of an callback treatment, the callback service permits use of only a small subset of its possible actions (available to your application as method calls).
Actions of the callback and voice interaction services may modify these states. The event service receives both `InteractionEvent` and `CallbackRecordEvent` events that carry identifiers along with a variety of attributes reflecting new state and other data. To receive those events, your application must subscribe to them.
For each `CallbackRecordEvent` incoming event, as well as for `InteractionEvent` event, your application should test various attributes of the `ICallbackService` interface.
The following sections present the details behind the above general description of the callback service. For voice management, see Voice Interactions.

Record Attributes

The `callback.record` domain of the the `ICallbackService` interface defines a callback record's characteristic data. The following list of data attributes is representative (but not exhaustive):

- Attributes for informative purposes:
  - `callback.record:contactInfo`—Contact information needed to perform a callback (for example, a contact phone number).
  - `callback.record:customFields`—Collected data useful to an agent calling the customer back.
  - `callback.record:scheduleDateTime`—Date and time scheduled for processing a callback record (in the format: `mmddyyyyhhmm`).
- Attributes for management purposes:
  - `callback.record:recordId`—Unique system ID of a record.
  - `callback.record:interactionId`—Unique system identifier of an interaction used to process a callback record.
  - `callback.record:reason`—Current reason for the callback record's status.
  - `callback.record:actionsPossible`—Possible callback actions that can be performed on a callback record.

Depending on attribute properties, your application can read the attributes of this domain with the `ICallbackService.getRecordsDTO()` method, and can modify them with the `ICallbackService.setRecordsDTO()` method. From the agent's point of view, a callback record does not exist independently from a voice interaction. Therefore, your application can use either a callback record ID or an interaction ID to access and manage records' attribute values with those get and set methods. This choice is specified with a boolean parameter in those methods'calls.
The `ICallbackService.getRecordsDTO()` and `ICallbackService.setRecordsDTO()` methods use `RecordDTO` objects. The `RecordDTO` class associates a record ID with a key-value list of attributes. For further information, see DTOs Handling.

Record Actions

The `RecordAction` enumeration defines callback actions on records. Constants each correspond to one `ICallbackService.*Record()` method. For example, the `RESCHEDULE` constant corresponds to the `ICallbackService.rescheduleRecord()` method. The actions apply to all records.
From an agent's point of view, a record does not exist independently from a voice interaction.

Therefore, your application can use either an interaction ID or a record ID to perform callback record actions. This choice is specified with a boolean parameter in those methods'calls.
To determine which callback record actions are possible at a certain time, read the `callback.record:actionsPossible` attribute of the `ICallbackService` interface.When changes occur on possible actions, this attribute is published in `CallbackRecordEvent` events only. See Record Events.

Record Status

The `RecordStatus` enumeration lists the possible statuses for a callback record. The purpose of a callback record status is to determine if a certain callback record is previewed, opened, processed, or closed. Your application can display this status. Changes are propagated in a `CallbackRecordEvent`.

Record Events

When changes occur on a callback record, the event service of your application can receive `CallbackRecordEvent` events for the agent handling the callback.
`CallbackRecordEvent` events can propagate any published attribute, that is, attributes of the `callback.record` domain that have the `event` property. To properly take into account callback events, the published `callback.record:eventReason` attribute value indicates the reason for an event. The `RecordReason` enumeration lists the possible reasons for an occurring `CallbackRecordEvent` event. Most of the time, it points out a status change as presented in Record Event Reasons.

Record Event Reasons

RecordReason	Attributes	Description
`CANCELLED`	`callback.record:status` `callback.record:possibleActions`	Record canceled.
`PROCESSED`	`callback.record:status` `callback.record:possibleActions`	Record processed.
`RESCHEDULED`	`callback.record:estimatedWaitTimedate` `callback.record:estimatedWaitTime`	Record rescheduled.
`REJECTED`	`callback.record:status` `callback.record:possibleActions`	Record rejected.
`UNKNOWN`	`callback.record:*`	Unknown event.

The following code snippet subscribes to both `CallbackRecordEvent` and `InteractionEvent`.

```
/// Defining two TopicsService
TopicsService[] myTopicsServices = new TopicsService[2] ;

/// Defining a Topic Service for the callback service
myTopicsServices[0] = new TopicsService() ;
myTopicsServices[0].serviceName = "CallbackService" ;
/// Defining a topic event
myTopicsServices[0].topicsEvents = new TopicsEvent[1] ;
myTopicsServices[0].topicsEvents[0] = new TopicsEvent() ;

/// the targeted events are CallbackRecordEvent
myTopicsServices[0].topicsEvents[0].eventName = "CallbackRecordEvent" ;
/// all the event attributes values are propagated in event objects
myTopicsServices[0].topicsEvents[0].attributes = new String[]{ "callback.record:*"};

/// Triggering CallbackRecordEvent for agent0
myTopicsServices[0].topicsEvents[0].triggers = new Topic[1];
myTopicsServices[0].topicsEvents[0].triggers[0] = new Topic();
myTopicsServices[0].topicsEvents[0].triggers[0].key = "AGENT";
myTopicsServices[0].topicsEvents[0].triggers[0].value = "agent0";

/// Defining a Topic Service for the interaction service
myTopicsServices[1] = new TopicsService() ;
myTopicsServices[1].serviceName = "InteractionService" ;

myTopicsServices[1].topicsEvents = new TopicsEvent[1] ;
myTopicsServices[1].topicsEvents[0] = new TopicsEvent() ;

/// the targeted events are InteractionEvent
myTopicsServices[1].topicsEvents[0].eventName = "InteractionEvent" ;

/// in case of a voice interaction, the interaction, and
/// voice interaction attributes values are propagated in the
/// Event object
myTopicsServices[1].topicsEvents[0].attributes = new String[]{ "interaction:*",
"interaction.voice:*"};

/// Triggering CallbackRecordEvent for agent0
myTopicsServices[1].topicsEvents[0].triggers = new Topic[1];
myTopicsServices[1].topicsEvents[0].triggers[0] = new Topic();
myTopicsServices[1].topicsEvents[0].triggers[0].key = "AGENT";
myTopicsServices[1].topicsEvents[0].triggers[0].value = "agent0";
```

For further information about events, see The Event Service.

Records Management

The following table presents actions of the callback service which provides your application with callback management.

Callback Record Actions and Methods

RecordAction	ICallbackService Method	Description
ACCEPT	acceptRecord()	Agree to process a callback record.
CANCEL	cancelRecord()	Cancel a callback record.
REJECT	rejectRecord()	Reject a callback record
RESCHEDULE	rescheduleRecord()	Reschedule a callback record.
PROCESSED	processedRecord()	Mark a callback record as processed.

When an agent has to process a callback, your event service receives both an `InteractionEvent` and a `CallbackRecordEvent` event. An agent can either accept or reject processing of the callback. If the agent has accepted a callback, your application uses the `IInteractionVoiceService` features to process the phone call, and uses the `ICallbackService` to manage the corresponding callback record.
A `CallbackRecordEvent` event propagates status changes for a callback record, which is identified by the `callback.record:recordId` attribute. Your application should test the `callback.record:actionsPossible` attribute to determine which actions are currently possible on this record.
The following sections detail the corresponding calls to methods of the callback service.

Accepting a Record

To accept a callback record, call the `ICallbackService.acceptRecord()` method. Your application can process the callback action with either the callback record ID or the companion voice interaction ID (represented by the `callback.record:interactionId` attribute).
Once the agent using your application has accepted the record, your application can provide him or her with management of the corresponding voice interaction.
The following code snippet shows a callback record accepted by its record ID for the `agent0`:

```
myCallbackService.acceptRecord("agent0",
false, // the call is not performed with the Interaction ID
myRecordId); // Id of the callback record accepted by agent0
```

Rejecting a Record

When an agent does not want to process a record, he or she can reject the record, so that the callback record returns in a queue and another agent processes the callback record.
Use the `ICallbackService.rejectRecord()` method to reject the record, as shown in the following code snippet:

```
myCallbackService.rejectRecord("agent0",
```

```
false, // the call is not performed with the Interaction ID
myRecordId); // Id of the callback record accepted by agent0
```

Canceling a Record

When an agent cancels a callback record, the callback record is removed from the queue and will not be processed further.
Use the `ICallbackService.cancelRecord()` method to cancel the record, as shown in the following code snippet:

```
myCallbackService.cancelRecord("agent0",
false, // the call is not performed with the Interaction ID
myRecordId); // Id of the callback record accepted by agent0
```

Rescheduling a Record

A customer might ask an agent to reschedule the call. Once the callback is rescheduled, it takes its place in the appropriate queue. A new companion voice interaction is created according to the scheduled date and time.
Use the `ICallbackService.rescheduleRecord()` method to reschedule the record, as shown in the following code snippet:

```
myCallbackService.rescheduleRecord("agent0",
false, // the call is not performed with the Interaction ID
myRecordId, // ID of the callback record accepted by agent0.
myDate); // long date in second, UTC
```

Marking a Record as Processed

When a callback record has been processed, an agent must mark the record as processed.
Use the `ICallbackService.processedRecord()` method to mark the record as processed, as shown in the following code snippet:

```
myCallbackService.processedRecord("agent0",
  false, // the call is not performed with the Interaction ID
   myRecordId); // ID of the callback record accepted by agent0.
```

The SRL Service

The SRL (Standard Response Library) service provides standard responses to help agents process interactions.

Introduction

The Agent Interaction Service API includes access to a self-learning categorization system to help agents by providing responses when they process an interaction.
The Standard Response Library (SRL) system it self-learning: it “teaches” itself with new incoming messages, according to agents’ feedback. For further information about the SRL, refer to Genesys Multi-Channel Routing documentation, which includes information about Universal Contact Server.
The following subsections detail how the SRL service interacts with the Standard Response Library database.

Standard and Suggested Responses

A standard response is a prewritten response stored in the Standard Response Library database. An agent may choose to reply to a customer with a response from the Standard Response Library. A standard response may have tags in its body that your application can automatically replace with contacts’ data.
When an agent processes an interaction, your application can display a tree of standard responses or the ones contained in the interaction’s suggested categories (if any).
The system selects this suggested categories according to categorization criteria. For details, see What Is a Category? below.
Your application can insert standard responses as replies into any e-mail or chat message, or can display them so that an agent can read them to the contact during a voice interaction.

What Is a Category?

A category is a group of standard responses and categories that are available for similar interactions. For example, a company might define a `Defect` category, which contains standard responses to provide when customers report a product defect. In this `Defect` category, this company might define a category for each product. Each category defines a set of more-specific responses for the product’s identified defects.
Your application can display categories as trees, allowing the agent to select a category and a standard response with that category. Your application can also propose an interaction’s suggested category. For instance, if an e-mail interaction has suggested categories, they are available in the `interaction.mail:suggestedCategories` attribute.
An agent can accept or reject the system’s choice of a selected category, in order to provide feedback to the Standard Response Library’s self-learning system.

What Is the SRL Service?

The SRL service provides your application with access to categories and standard responses stored in the Standard Response Library database. This service can be used independently from other Agent

Interaction SDK (Web Services).
This chapter's remaining sections cover the SRL service's main features:

- Using interfaces to standard responses and categories.
- Providing feedback to the self-learning system.
- Getting standard responses and categories.
- Adding categories to an agent's favorites.

Using Standard Responses and Categories

Your application can use the `ISRLService` interface to access standard responses when an agent processes an interaction. Your application can use the information retrieved by the SRL service to display categories, and standard responses, in trees.

Using Standard Response

A Standard Response is prewritten message text that your application can insert in an outgoing e-mail or chat interaction, or that your application can display for a voice interaction. This prewritten message includes variables that correspond to contact information, and also additional data such as attachments. The SRL service can replace the SRL variables with the appropriate contact information. For each standard response, the SRL service exposes information in the `srl` domain:

- `srl:srlReference` —ID of a standard response; used to access a standard response information.
- `srl:name` —The name of a response.
- `srl:body` —Message text containing a response and contact variables.
- `srl:agentDesktopUsageType` —An SRLUsageType enumerated value, which indicates how to use this response.
- `srl:isActive` —`true` if this response is active in the Configuration Layer.
- `srl:hasAttachement` —`true` if a response has attachments.
- `srl:attachments` —Attachments to a response.

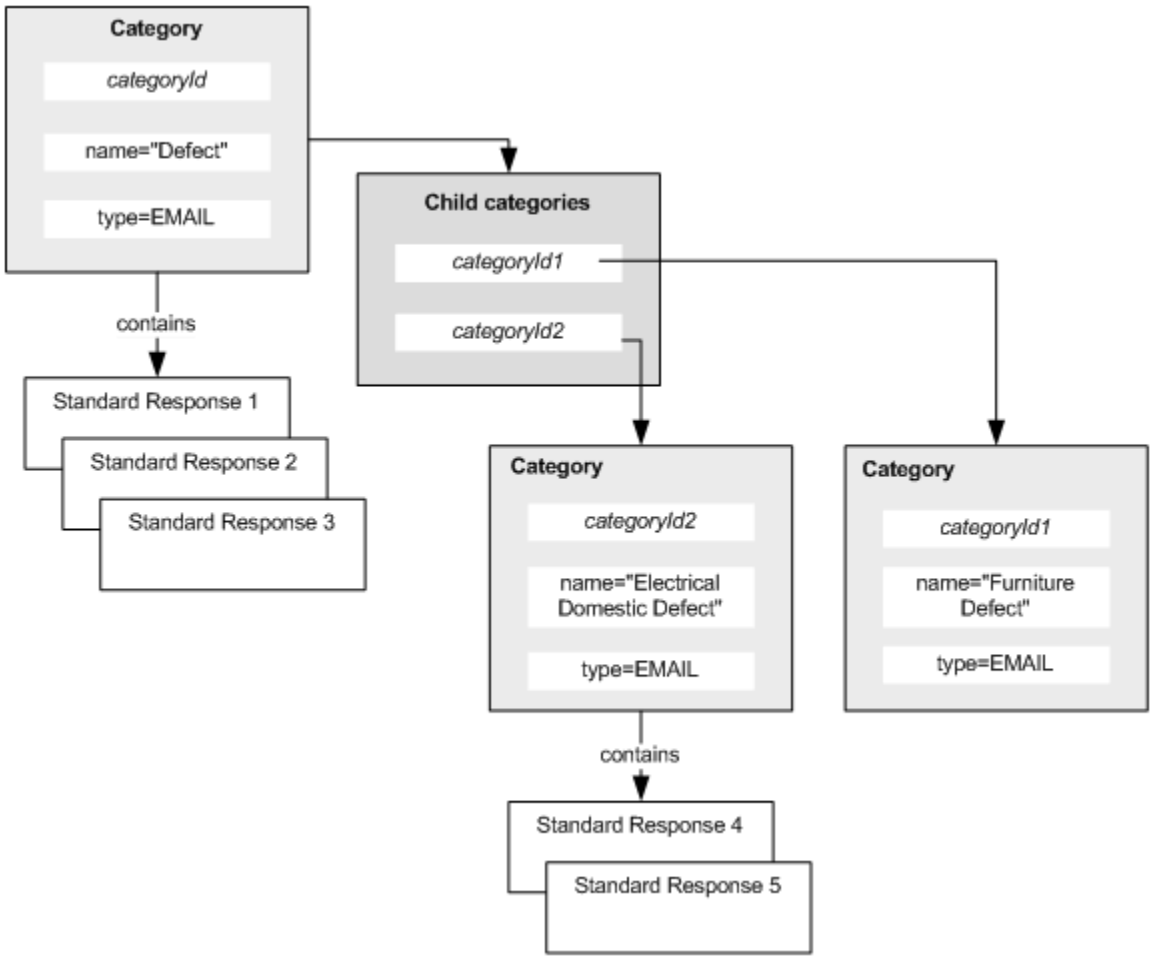
Your application might have to get this information to display a standard response, or to add it into an interaction. The `ISRLService` interface provides some `getStandardResponse*DTO()` methods to retrieve standard response information in `StandardResponseDTO` objects.
The `StandardResponseDTO` class associates a standard response ID with a key-value array corresponding to `srl:*` attributes. Data Transfer Object

Category Information

In the SRL database, a category is composed of:

- Its own data, such as the category name, ID, language and type.
- A set of standard responses that belong to the category.

- An array of its child category IDs.



Category in the SRL Database

Category in the SRL Service

For each category, the `ISRLService` interface defines a set of attributes in the `srl-category` domain. The following `srl-category` attributes characterize a category (this is a non-exhaustive list):

- `srl-category:categoryId` —system ID of a category.
- `srl-category:name` —name of a category.
- `srl-category:type` —type of a category.

Your application can retrieve the corresponding key-value pairs in `SRLCategoryDTO` objects. Each `SRLCategoryDTO` instance contains the information for a single category:

- `categoryId` —ID of the relevant category.
- `data` —The key-value array, containing the `srl-category` data for the relevant category.
- `srlsDTO` —An array of DTO, containing the standard responses for the relevant category.

Child Categories

The `srl-category:childCategories` attribute contains a category's child categories in an array. This array, in turn, contains key-value arrays containing the attributes retrieved for the category's child categories.
The standard responses of the child categories are not retrieved. For details, see Getting Category DTO.

Feedback and Interaction Services

Your application can use the `ISRLService` interface to display category information when an agent is working with interactions. For each interaction, the self-learning system can suggest categories. The `IInteractionService` interface exposes the following attributes to provide access to the suggested categories:

- `interaction:categoryId` —ID of the current category assigned to an interaction by the self-learning system, or assigned manually by an agent.
- `interaction.*:addSuggestedCategories` —Adds or updates the suggested categories for this e-mail.
- `interaction.*:suggestedCategories` —An array containing the IDs of the suggested categories.
- `interaction.*:isCategoryApproved` —A boolean, whose value is 1 if an agent approves the category identified by the attribute value for an interaction or `0` , if he or she disapproves. When your application writes a value for this attribute, it provides an agent's feedback to the self-learning system.

When an agent chooses a response in a category, he or she should be able to validate his or her choice for the self-learning system. For that, he or she has to assign the `interaction:categoryId` and`interaction.*:isCategoryApproved` attributes, see Category and Feedback.

Category and Feedback

Feedback	categoryId	isCategory Approved	addSuggested Categories
The agent uses the standard response of a suggested category.	Suggested category's ID.	`true`	none
There is no satisfactory category in the suggested categories.	ID of the suggested category that has the best relevancy.	`false`	none

Feedback	categoryId	isCategory Approved	addSuggested Categories
The agent uses the standard response of a non-suggested category.	ID of the satisfactory category.	`true`	`SuggestedCategory` object that contains the category's ID and a `null` relevancy.

Getting Categories and Standard Responses

The following subsections detail how to get information from the SRL service.

Getting Category DTO

Your application can get categories' data using the `getCategorieDTO()` method. This method returns an array of `SRLCategoryDTO` objects. Your application can specify a set of category identifiers in parameters or can retrieve all the root categories of the SRL, as shown in the following code snippet.

```
SRLCategoryDTO[] myRootCategories = mySRLService.getCategoriesDTO( null,//To return the data
of all the root categories
   new string[]{ // attributes to return for each category
      "srl-category:categoryId",
      "srl-category:name",
      "srl-category:childCategories",
      "srl-category:description"},
      new string[]{"srl:*"}); // attributes for each Standard Response of the category
```

If your application requests the `srl-category:childCategories` attribute for each category, then the attributes specified in parameters are also retrieved for the child categories—including the `srl-category:childCategories` attribute.

> **Important**
>
> The standard responses of the child categories are not retrieved.

Getting Standard Responses

Your application can retrieve the standard responses in different ways:

- When retrieving categories, the categories' standard responses are included in `SRLCategoryDTO.srlsDTO` field.
- Your application can retrieve the `srl` data for a set of standard response IDs with the getStandardResponsesDTO() method.
- Your application can retrieve the `srl` data for all standard responses of a particular category with the `getStandardResponsesByCategoryDTO()` method.

The following code snippet shows how to get the name and the description of the standard response.

```
StandardResponseDTO[] myResponses = mySRLService.getStandardResponsesDTO(
    new string[]{"SR01"},
    new string[]{"srl:name","srl:description"});
```

Using the getStandardResponseBody( ) Method

As presented in the previous sections, the body text of a standard response may include some code that can be replaced with contact data (for example, the name of the contact). Your application can use the `getStandardResponseBody()` method for this purpose, as shown in the following code snippet.

```
String myFilledBodyText = mySRLService.getStandardResponseBody("SR01", //ID of the response
   "agent0", // ID of the agent requesting the body text
   "myInteractionId"); // ID of the processed interaction

System.Console.WriteLine(myFilledBodyText);
```

Managing Favorites

Your application can use the SRL service to manage a list of an agent's favorite responses. The SRL service can retrieve those favorites, add new favorites, or remove some favorites, as detailed in the following subsections.

Getting the Favorite Standard Responses

Your application can get the favorite standard responses of a set of agents, using the `getStandardResponsesFavoritesDTO()` method. This method retrieves all the requested attributes about the favorite standard responses for each specified agent.
The following code snippet shows how to retrieve the favorites of the agent `agent0`.

```
//Retrieving the favorites of agent
SRLUsernameDTO[] myAgentFavoritesDTO = mySRLService.getStandardResponsesFavoritesDTO(
   new string[]{"agent0"}, // List of agents
   new string[]{"srl:*"}); // List of attributes to get for each response

if(myAgentFavoritesDTO[0].username == "agent0")
{
   StandardResponseDTO[] myFavoriteResponsesDTO = myAgentFavoritesDTO[0].srlDTO;
   /// displaying agent's favorite responses
   /// ...
}
```

> **Important**
> The user name of an agent is his or her agent ID.

Adding Standard Responses to Favorites

Your application can add new favorite standard responses using the `ISRLService.addSRLFavorites()` method.
To add new favorite standard responses for an agent, your application must first fill `SRLform` objects, as shown in the following code snippet:

```
/// Creating an SRL form
SRLForm mySRLForm = new SRLForm();
// Each SRLForm is used to add a response to // the favorites of an agent
mySRLForm.username = "agent0";
mySRLForm.standardResponseId = "SR01";
// Adding SR01 to the favorites of agent0
SRLFormResult[] mySRLFormResults= mySRLService.addSRLFavorites(new SRLForm[]{mySRLForm});

// Displaying the result for each standard response // that the service tried to add to the
favorites of an agent.
foreach(SRLFormResult myFormResult in mySRLFormResults)
{
   if(        myFormResult.done == true) {
      System.Console.WriteLine( myFormResult.SRLForm.standardResponseId
        +" has been added to the favorites of " +    myFormResult.SRLForm.username+"\n");
   }
}
```

Removing Standard Responses from Favorites

Your application can remove favorite standard responses using the `ISRLService.removeSRLFavorites()` method.
To remove standard responses from the favorites, your application must first fill `SRLform` objects, as shown in the following code snippet:

```
/// Creating an SRL form
SRLForm mySRLForm = new SRLForm();
// Each SRLForm is used to remove a response from the favorites
// of an agent
mySRLForm.username = "agent0";
mySRLForm.standardResponseId = "SR01";

// Removing SR01 from the favorites of agent0
SRLFormResult[] mySRLFormResults= mySRLService.removeSRLFavorites(new SRLForm[]{mySRLForm});

//Displaying the result for each standard response
//that the service tried to remove from the favorites of an agent.
foreach(SRLFormResult myFormResult in mySRLFormResults)
{
   if(        myFormResult.done == true)
   {
     System.Console.WriteLine( myFormResult.SRLForm.standardResponseId +
      " has been removed from the favorites of " + myFormResult.SRLForm.username+"\n");
   }
}
```

The Outbound Service

The outbound service is the `IExtendedOutboundService` interface defined in the`com.genesyslab.ail.ws.outbound` namespace. It covers the management of outbound campaigns.
The `IExtendedOutboundService` replaces the deprecated `IOutboundService`.

> **Important**
>
> You must still use the `IOutboundService` if your Agent Interaction Service Configuration Layer `enable-chain-75api` option (in the `outbound` section) is set to `false`.

For detailed information on the `IOutboundService`, see the 7.2 version of this *Developer's Guide.*

Introduction

The following subsections provide an overview of the outbound service and its constituent campaigns, outbound chains, records, and interactions.

Outbound Campaigns

An *outbound campaign* is a flexible master plan that organizes calling lists for generating calls to customers, handling customer data, and managing call results.
The Outbound Contact Server (OCS) manages outbound campaigns and automates outbound call dialing. For further information, refer to the *Outbound Contact 7.6* documentation.

Outbound Records

An *outbound record* is an element of a calling list for an outbound campaign. It contains the information required to enable an agent to call a contact—for example, a phone number, the contact name, and how to process the record (type and status).
Chained records are multiple records for the same contact in a calling list. These occur when a contact has several available phone numbers. Each record of the chain can have different time boundaries, as well as different values stored in its business data field.

Outbound Chains

Each *outbound chain* instance contains customer outbound data, provided as a collection of outbound record objects. For example, in the context of a voice outbound call, each record of the chain contains a phone number associated with the customer to be called. If the call with a given record fails, the agent can get a chained record to attempt a new call for this customer.
Regardless of the campaign mode, your application can receive events for new or modified outbound

chains that the agent should process. To determine which outbound chain is associated with an interaction, check the `interaction:outboundChainId` attribute.

The Outbound Service

The outbound service interfaces with outbound campaigns and lists of records. Your application should use this service to let an agent participate in an outbound campaign by processing the records of a calling list.

Features

Your application can use the outbound service to:

- Stay informed about a campaign's progress once the campaign is started.
- Modify a campaign:
  - Add new records to a campaign.
  - Change the campaign dialing mode (if possible).
- Process an agent's outbound voice interactions of an agent:
  - Dial a record.
  - Stay informed about an outbound call's progress.
  - Change the active record for an outbound voice interaction.
  - Cancel, mark as `Do Not Call`, or reject a record or a chain of records.

Mandatory Services

Your application cannot use the outbound service independently from the following services:

- The agent service:
  - Before dialing calls, your application first must use the agent service to log in an agent on a DN. See also The Agent Service.
- The voice interaction service:
  - While the agent is logged on a DN, your application can use the `IInteractionVoiceService` interface to process the calls corresponding to records.
    See also Voice Interactions.
- The interaction service:
  - To access the attributes of your voice interactions, your application uses the `IInteractionService` DTO methods.
  - All the voice interaction attributes are published in events of type `InteractionEvent`. This type is described in the `IInteractionService` interface. Outbound chain information is accessible from this service.
    See also The Interaction Service.

Outbound Campaigns

An agent who participates in an active or running outbound campaign, should be informed of those campaigns' characteristics and also of campaign changes when they occur.
The `IExtendedOutboundService` interface lets your application access campaign information, as detailed in the following.

Campaign Attributes

For each outbound campaign, the `IExtendedOutboundService` interface defines a set of attributes in the `outbound.campaign` domain. Your application can use these attributes for information purposes:

- `outbound.campaign:name`—The name of the outbound campaign.
- `outbound.campaign:description`—The description of the outbound campaign; for example, its purpose.
- `outbound.campaign:mode`—The current mode of the campaign.

For each campaign, your application uses the following attributes for management:

- `outbound.campaign:campaignId`—The system ID of the campaign.
- `outbound.campaign:state`—The status of the campaign.
- `outbound.campaign:eventType`—When present, this event indicates the type of campaign.
- `outbound.campaign:actionsPossible`—The possible actions that the agent can perform on the campaign at a certain point in time.

The following sections detail how to use these attributes.
Use the `IExtendedOutboundService.getCampaignsDTO()` method to read attributes having the `read` property. A call to this method returns an array of `CampaignDTO` objects. The `CampaignDTO` class is a container that associates a campaign ID with a key-value list. For further information about DTO, see Data Transfer Object.
The `IExtendedOutboundService` interface has no writable campaign attributes. An agent using your application cannot change the campaign attribute values. However, actions and external events can modify the attribute values. Changes are propagated in events. See Campaign Events.

Campaign Dialing Modes

The *campaign dialing modes* determine how an agent, or a group of agents, participates in an outbound campaign. The table below presents the possible dialing modes of an outbound campaign.

Campaign Dialing Modes

CampaignMode	Description
PREVIEW	In this dialing mode, an agent requests one or several records from the OCS, previews the record(s), and decides to process one of them.
PUSH_PREVIEW (also called PROACTIVE)	In this mode, the agent receives the record, and does not need to request it in order to preview it.
PROGRESSIVE	The OCS dials a record in the list as soon as an agent is available.
ENGAGED_PROGRESSIVE	The OCS creates a voice interaction to dial a record in the list when an agent is available and engaged.
PREDICTIVE	The OCS predicts agent availability and dials a record. Your agent application gets a dialing voice interaction and an outbound chain for this record.
ENGAGED_PREDICTIVE	The OCS predicts when an agent is available and engaged, and creates a voice interaction to dial a record in the list.
UNKNOWN	Unknown campaign mode.

> **Important**
>
> In a mode that is not ENGAGED_, the contact may be online before the agent. For further information, refer to the Outbound Contact 7.6 documentation.

Campaign Actions

The `CampaignAction` enumeration lists the possible actions that your agent application can perform on a campaign using certain `IExtendedOutboundService` methods. `CampaignAction` usually pertains to campaigns in preview dialing mode.
Test the `outbound.campaign:actionsPossible` attribute of the `IExtendedOutboundService` interface to determine which actions are possible at a certain point in time. When possible actions on a campaign change, a `CampaignOutboundEvent` event may propagate the new value of the `outbound.campaign:actionsPossible` attribute for this campaign. See Campaign Events.

Campaign Status

The current status of a campaign is available as the value of the interface's `outbound.campaign:state` attribute. The `CampaignStatus` enumeration lists the possible status values of an outbound campaign. For further information, refer to the *Agent Interaction SDK 7.6 Services API Reference*.
The status of an outbound campaign can change due to the OCS management of campaigns. Status changes are propagated with `CampaignOutboundEvent` events. See Campaign Events.

Campaign Events

When changes occur on an outbound campaign, the event service of your application can receive `CampaignOutboundEvent` events only for agents who take part in the outbound campaign.
To properly take into account those events, your application must subscribe to them with the event service and must retrieve at least the following published attributes:

- `outbound.campaign:campaignId`—Determines which campaign is related to the event.
- `outbound.campaign:eventType`—When present, this event indicates the type of campaign.
- `outbound.campaign:actionsPossible`—Updated possible actions for the related campaign.

The published `outbound.campaign:eventType` attribute points out which value changes are propagated in an `CampaignOutboundEvent` event. The `CampaignEventType` enumeration lists the possible values of this attribute. The table below shows which published attributes to read according to the `CampaignEventType` value.

CampaignEventType and Published Attribute

Campaign-EventType	Associated Attribute	Description
`CAMPAIGN_ADDED`	`outbound.campaign:*`	A campaign was added in the outbound service.
`CAMPAIGN_REMOVED`	`outbound.campaign:*`	A campaign was removed from the outbound service.
`CAMPAIGN_MODE_CHANGED`	`outbound.campaign:mode`	The mode of a campaign has changed.
`CAMPAIGN_STATE_CHANGED`	`outbound.campaign:state`	The status of a campaign has changed.
`UNKNOWN`	`outbound.campaign:*`	Unknown type of event on an outbound campaign.

For further information about events and subscribing, see The Event Service.

Outbound Chain Events

When changes occur on an outbound chain, the event service of your application can receive`OutboundChainEvent` events only for agents who take part in the outbound campaign. To properly take into account those events, your application must subscribe to them with the event service, and must retrieve at least the published attributes listed in the following table.

Outbound Chain Events

Attribute	Description
`outbound.chain:activeRecordId`	The active record of the chain.
`outbound.chain:recordIds`	The list of outbound record identifiers in this chain. The number of records can change during the life cycle of the interaction. The interaction has one record at initialization (the initial record), and at least one record after calling `requestChainedRecords`.
`outbound.chain:eventType`	The type of the event.[#pgfId-1025870 1]
`outbound.chain:outboundChainId`	The interaction identifier.a
`outbound.chain:reason`	The reason.a
`outbound.chain:campaignMode`	The campaign mode related to this outbound chain.a
`outbound.chain:interactionIds`	The interaction identifers related to this chain.

Outbound Chains and Records

The Introduction and Outbound Campaigns sections introduced you to the design of the outbound feature and noted that outbound data does not interfere with interaction management. Now, to handle campaign information and outbound records, you need to add some code and make certain modifications to your agent application.
These changes you need to make require that you address two main issues:

- Subscribe to the correct events.
- Identify if a given interaction has outbound information.

Subscribe to Outbound and Chain Events

In order to get notified of changes to active outbound campaigns and chains, you need to subscribe to `CampaignOutboundEvents` and `OutboundChainEvents`. Use the following code snippets as guidelines for how to subscribe:

```
/// Creating topic objects for the outbound service
TopicsService [] topicServices = new TopicsService[1];
topicServices[0] = new TopicsService();
topicServices[0].serviceName = "IExtendedOutboundService";
topicServices[0].topicsEvents = new TopicsEvent[2];
topicServices[0].topicsEvents[0] = new TopicsEvent();

// Creating a topic event for campaign outbound events
topicServices[0].topicsEvents[0].eventName = "CampaignOutboundEvent";
topicServices[0].topicsEvents[0].attributes =
new String[]{"outbound.campaign:*"};
topicServices[0].topicsEvents[0].triggers = new Topic[1];
topicServices[0].topicsEvents[0].triggers[0] = new Topic();
topicServices[0].topicsEvents[0].triggers[0].key = "PLACE";
topicServices[0].topicsEvents[0].triggers[0].value = mPlaceId;
topicServices[0].topicsEvents[1] = new TopicsEvent();

// Creating a topic event for outbound chain events
topicServices[0].topicsEvents[1].eventName = "OutboundChainEvent";
topicServices[0].topicsEvents[1].attributes =
new String[]{"outbound.chain:*"};
topicServices[0].topicsEvents[1].triggers = new Topic[1];
topicServices[0].topicsEvents[1].triggers[0] = new Topic();
topicServices[0].topicsEvents[1].triggers[0].key = "PLACE";
topicServices[0].topicsEvents[1].triggers[0].value = mPlaceId;
```

Check Interactions for Outbound Information

When an interaction arrives at your agent application while an outbound campaign is active, depending on the mode of the campaign, you may need to check to see if interactions with `NEW` and `DIALING` status are associated with outbound information. If that is the case, add the following code to your application so that it handles the appropriate `InteractionEvents` for such outbound interactions:

```
InteractionDTO[] myInteractionDTO =
   myInteractionService.getInteractionsDTO(
      new string[]{myInteractionId},
      new string[]{"interaction.outboundChainId"});
KeyValue myAttrKeyValue = myInteractionDTO[0].data[0];
String myOutboundChainId = (String) myAttrKeyValue.value;
```

Outbound Attributes

The `IExtendedOutboundService` interface exposes two types of attributes used to process an outbound record, one type each, respectively, in the `outbound.chain` and`outbound.record` domains.

Outbound Chain Attributes

The `outbound.chain` domain defines a subset of outbound data for an outbound chain. See Outbound Chain Events for details. These outbound chain attributes make the links between interactions and outbound records.

Record Attributes

For each outbound record, the outbound.record attributes define a set of data characteristics related to an outbound record. The following tables list representative (but non exhaustive) record attributes. The first list contains information attributes and the second, attributes for management purposes.

Record Attributes for Information (Selected)

Attribute	Description
outbound.record:callingListName	The name of the calling list to which a record belongs.
outbound.record:phone	The phone number to dial to process a record.
outbound.record:campaignId	The ID of the campaign to which a record belongs; this ID can be used to retrieve information about the corresponding campaign. See Outbound Campaigns for details.
`outbound.record:outboundChainId`	The `OutboundChain` to which this record belongs, or `null`.

Record Attributes for Management (Selected)

Attribute	Description
`outbound.record:recordId`	The ID of a record.
`outbound.record:status`	The status of a record.
`outbound.record:actionsPossible`	The possible actions on a record.

Attribute	Description
`outbound.record:callResult`	The call result of a record.

From the agent point of view, a record does not exist independent of an outbound voice interaction. So your application needs an interaction ID to access a given record's attribute values.
Using the attribute properties, your application can read the attributes of the domain with the`IExtendedOutboundService.getRecordsDTO()` method. It can then write those attributes with the `IExtendedOutboundService.setRecordsDTO()` method. These methods use `OutboundRecordDTO` objects. The `OutboundRecordDTO` class associates a record ID with a key-value list of attributes. For further information, see Handling Interaction DTOs.

> **Important**
>
> When your application has set new values for a record with the `IExtendedOutboundService.setRecordsDTO()` method, your application must call the`IExtendedOutboundService.updateRecord()` method to commit all modifications in the OCS.

Outbound Actions

The `IExtendedOutboundService` interfaces provide you with outbound methods that the agent calls to perform outbound actions. These include `cancel`, `do not call`, and`reschedule`. Such actions are independent from the interaction used to process the outbound record or the outbound chain. They manage record information on the Outbound Server.

Outbound Campaign in Preview Mode

In preview and predictive dialing modes, an agent may have to use the `START_PREVIEW_MODE`, `GET_PREVIEW_RECORD`, and `STOP_PREVIEW_MODE` actions according to the options set in the OCS and Configuration Layer. For further details, refer to the Outbound Contact 7.6 Reference Manual.
When an agent explicitly initiates preview mode with the outbound service, he or she notifies the OCS that he or she is ready to participate in a specific campaign.
In this specific mode, an agent requests one or several records from the OCS, previews each record, and decides whether or not to dial a call. For an agent to stop preview mode with the outbound service, he must notify the OCS that he no longer is participating in the specified campaign. The `IExtendedOutboundService` interface provides your application with three methods to work with preview mode.

Outbound Service Methods for Preview Mode

Method	Description
`startPreviewMode()`	An agent is ready to start participating in a campaign.
`getPreviewRecordDTO()`	An agent retrieves a preview record from a campaign, and this method returns the record data in a DTO. Your application receives an interaction event for the outbound voice interaction associated with this outbound record.
`stopPreviewMode()`	An agent stops participating in a campaign.

> **Important**
>
> If your campaign mode is push preview, your application does not need to request the preview record.

The following code snippet shows how to use the startPreviewMode(), getPreviewRecordDTO(), and stopPreviewMode() methods:

```
/// agent0 is ready to participate in a campaign identified by
/// myCampaignID
myOutboundService.startPreviewMode(“agent0”,myCampaignID);
///...
///Retrieving a preview record for agent0
OutboundRecordDTO myRecordDTO =
myOutboundService.getPreviewRecordDTO("agent0", myCampaignId,
new string[]{"outbound.record:*"}); /// all record attributes
/// Displaying the DTO content
foreach(KeyValue myPair in myRecordDTO.data)
{
   System.Console.WriteLine("Key="+ myPair.key
   +" value="+myPair.value.ToString());
}
/// agent0 no longer participates in a campaign identified by
/// myCampaignID.
myOutboundService.stopPreviewMode(“agent0”,myCampaignID);
```

Then, if a record is available, you get two events:

- `OutboundChainEvent`
- `InteractionEvent`

Use the `OutboundChainEvent` event to inform the user that a new outbound record should be processed.

The corresponding InteractionEvent references an interaction object with the status NEW. Use this interaction to retrieve the outbound chain that contains the preview record. To get this record, get the outbound.chain:activeRecordId attribute using OutboundChainDTO, as follows:

```
myOutbondChainDTO = myOutboundService.getOutboundChainDTO(myPlaceId, myOutboundChainId,
   new string[]{"outbound.chain:activeRecordId"}
);
```

Then, use the Interaction instance as you normally would to process the outbound record. Get outbound record data to fill in Interaction data and the parameters for your methods. After that, you need to create an interaction to process a voice call and use the record data to dial the call.
If you deal with multimedia interactions, multimedia information is available in the record's custom fields (outbound.record:customFields). When the interaction is processed, you mark the corresponding outbound chain as processed:

```
myOutboundService.markChainProcessed(myPlaceId,myOutboundChainId);
```

Outbound Campaign in Predictive Mode

Handling a predictive outbound interaction is simpler than handling a preview outbound interaction. The setup is the same, but you do not have to request records since Outbound Server is in charge of distributing records. Refer to outbound documentation for further details. For a predictive outbound campaign, your application just waits for RINGING interactions and outbound chains.

Active Campaigns

To determine if your agent is to participate in a predictive outbound campaign, test whether the outbound.campaign:mode is PREDICTIVE. To do this, get information in CampaignOutboundEvent events.

Handling a Predictive Outbound Interaction

If your agent is to participate in a predictive campaign, your application should get interactions in DIALING or TALKING status, which you process as usual. For further details, see The Interaction Service, and other interaction-handling chapters.
When your application gets these predictive interactions through interaction events, identify the outbound data. Each received outbound interaction is associated with an outbound chain that contains the record. Use this data to fill in interaction data. To get this record, get the outbound.chain:activeRecordId attribute using OutboundChainDTO:

```
myOutbondChainDTO = myOutboundService.getOutboundChainDTO(myPlaceId,
   myOutboundChainId, new string[]{"outbound.chain:activeRecordId"});
```

When the agent has processed the outbound record, he or she must specify the processing result using the IExtendedOutboundService.setRecordDTO method. When the interaction is processed, you mark the corresponding outbound chain as processed:

```
myOutboundService.markChainProcessed(myPlaceId,myOutboundChainId);
```

To determine whether the agent must get a record or wait for a new interaction, refer to Outbound Solution Documentation.

Expert Contact

The expert contact service is the `IExpertService` interface defined in the`com.genesyslab.ail.ws.expert` namespace. It covers the management of expert contact features.

Introduction

The Expert Contact service supports features for an application that lets expert users, who are not part of an enterprise's Contact Center, provide their expertise to Contact Center agents or customers. This section defines what an expert, an expert contact application, and the expert service are.

What is an Expert?

Contact Center agents, in the course of responding to customers, sometimes need information beyond their training and benefit from contacting with people with special expertise.
But typically, such experts are not part of the Contact Center CTI infrastructure: their telephones connect to a switch that does not have a T-Server and Framework support. In such cases, experts' interactions with Contact Center agents (or their customers) are not tracked, and neither the agents nor the experts can benefit from information the Genesys Solutions can provide.

What is an Expert Contact Application?

In a site without a CTI link, the expert receives phone calls directly from a public network without involvement of any Genesys platform components. Therefore such calls are not automatically monitored or controlled. For example, there is no way to detect the state of the expert's telephone (`Ready`, `OnCall`, and so on).
The purpose of an Expert Contact application is to provide a means for an expert to reflect call state and perhaps to present information from the Genesys Framework components to the expert. But there is no T-Server for the switch for the expert's telephone.
A Genesys CTI-Less T-Server works without monitoring a switch yet provides a connection to the expert's desktop application and to a T-Server in the Contact Center. A CTI-Less T-Server provides a virtual CTI environment to track the expert's telephone states, send messages to other Genesys server components, handle data for current interactions, and coordinate voice and data delivery to the expert's desktop application.
In the case that an expert receives a call transferred from a Genesys supported contact center, the Genesys platform components communicate with the CTI-Less T-Server, which signals the expert's desktop application of an incoming phone call. The application presents an indication to the expert, who may choose to accept or reject the phone call.
If the expert accepts the phone call, the desktop application can present available information, including contact history if there is a connection to a Genesys Contact Server.
As the phone call progresses, the expert must use the application to reflect progress. The application passes the expert's activity to the CTI-Less T-Server, which in turn passes the data to the Contact Center Framework components.
The application must be able to process events from the CTI-Less T-Server.

What Is the Expert Contact Service?

The expert contact service interfaces with expert contexts (or expert contact data). Expert contexts are associated with voice interactions and pass expert activities on these interactions.

Features

Your application can use the expert contact service to:

- Access the expert context associated with a call—that is, a voice interaction.
- Pass expert activity:
  - Specify whether the expert is on call.
  - Update the expert status on a call.
  - Manage preview requests for outbound campaigns.
- Re-route calls.

Required Services

Your application cannot use the expert contact service independently from the following services:

- The agent service:
  - Before dialing calls, your application first must use the agent service to log in an agent on a CTI-Less DN. See also The Agent Service.
- The voice interaction service:
  - While the agent is logged on a CTI-Less DN, your application can use the `IInteractionVoiceService` interface to pass the expert's activity. See also Voice Interactions.
- The interaction service:
  - To access the attributes of your voice interactions, your application uses the `IInteractionService` DTO methods.
  - All the voice interaction attributes are published in events of type `InteractionEvent`. This type is described in the `IInteractionService` interface.
  - For further information, see the Interaction Service.
- The event service—This service is required to receive agent, voice, and expert events. See also the Event Service.

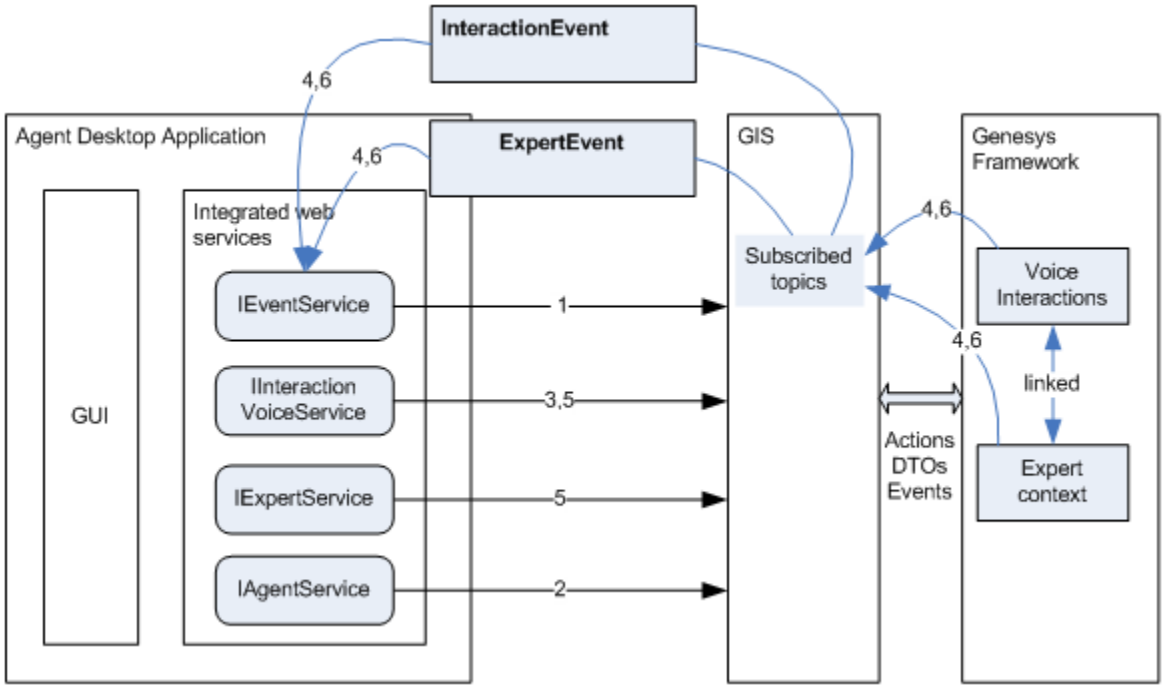
Configuration

An Expert Contact desktop application built on the Web Services must connect via the GIS to the CTI-Less T-Server and also a Configuration Layer that has information about the CTI-Less T-Server, person information for the expert, and so on.

> **Important**
>
> See the Genesys Expert Contact Solution 7.6 Deployment Guide for instructions on how to configure an Genesys Expert Contact application in Configuration Layer.

Expert Contact Essentials

Your agent application uses the expert contact service to manage expert contact information as an expert context attached to a voice interaction.
When an expert contact application creates or gets a voice interaction, it uses the expert contact service to get the associated expert context and manage particular expert actions. The voice interaction management does not differ from the management described in Voice Interactions. The particularity is that actions on voice interactions are passing expert agent actions to the CTI-Less T-Server.
The expert context of a voice interaction includes the expert context status and possible expert actions. Your expert application concurrently uses the voice interaction service and the expert contact service to pass his or her activity.
The following example shows how to integrate the expert contact service to handle expert contexts.



Expert Contact Service Integration Example

1. Subscribing to TopicsEvent for integrated services, including the expert and interaction services.

2. Requesting an agent login with the agent service login action.
3. Creating a voice interaction to pass a new dialed call.
4. Receiving an expert event for a new expert context, and an interaction event for the created interaction.
5. Managing both the expert context and the voice interaction.
6. Receiving expert and interaction events due to 5.

This diagram shows the general expert contact service handling in an application that lets an expert pass the creation of a new call. The `IExpertService` interface exposes methods and pertinent attributes that your application uses to manage the expert context of voice interactions, such as re-routing a call or updating the expert context status.
The `IInteractionVoiceService` interface passes the expert actions restricted to voice interaction management, such as for example, answering a call, releasing a call, and marking a call as done.
For any particular state of an expert context, the expert contact service permits the use of only a small subset of its possible actions (available to your application as method calls).
Actions of the voice interaction and expert contact services may modify these states. The event service receives both `InteractionEvent` and `ExpertEvent` events that carry identifiers, along with a variety of attributes reflecting new state and other data. To receive those events, your application must subscribe to them.
For each `ExpertEvent` event, as well as for `InteractionEvent` events, your application should test various attributes of the `IExpertService` interface.
The following sections present the details behind the above general description of the expert contact service. For voice management, see Voice Interactions.

Expert Context Attributes

The `expert` domain defines the expert context of a voice interaction. The following list is representative (but not exhaustive):

- `interactionId`—The voice interaction associated with this expert context.
- `customerNumber`—The customer number.
- `status`—The status of this expert context.
- `reason`—The reason for the status of this expert context.

An expert context does not exist independently from a voice interaction. Therefore, your application requires an interaction ID to access context attribute values.
Your application can read the attributes of this domain with the `IExpertService.getExpertContextDTO()` method. This method retrieves an `ExpertContextDTO` object that contains the context associated with a voice interaction.

> **Important**
> If no expert context exists for a voice interaction ID, your application gets a `error.expert.ExpertContextNotFound` exception. Your application can use it to determine whether it handles a voice interaction for an expert or not.

The ExpertContextDTO class associates an interaction ID with a key-value list of expert attributes. For further information, see DTOs Handling.

Expert Context Actions

The ExpertContextAction enumeration defines expert actions on voice interactions and expert contexts. Constants each correspond to one IExpertService method. For example, the REJECT constant corresponds to the IExpertService.reject() method.
When using these actions, your application can modify the expert context and data of a voice interaction (depending on the action). For further details, Using Expert Contact Features.
To determine which expert actions are possible at a certain point in time, read the expert:actionsPossible attribute of the IExpertService interface.
When changes occur on possible actions, this attribute can be published in an ExpertEvent event, depending on your subscription to the event service. See Expert Events below.

Expert Context Status

The ExpertContextStatus enumeration lists the possible statuses for an expert context. Since the expert phone is CTI-Less, the purpose of this status is to identify and tracks the expert context progress. For example, when the expert context status is ExpertContextStatus.PREVIEW, the expert previews the call; when the status is ExpertContextStatus.IN_PROGRESS, the expert processes the call; and so on.
The ExpertContextStatus.STATUS_REQUEST value corresponds to the status request feature. This status indicates that the expert must update his or her status at the CTI-Less T-Server's request. See section Managing Status Request for further details.
To determine what the expert contact status is at a certain point in time, read the expert:status attribute of the IExpertService interface. When changes occur on the expert context status, this attribute can be published in an ExpertEvent event, depending on your subscription to the event service. See Expert Events below.

Expert Events

When changes occur on an expert context, the event service of your expert application can receive ExpertEvent events. This occurs, for instance, when possible actions or expert context status change. Use expert events to update your expert application.
ExpertEvent events can propagate any published attribute, that is, attributes of the expert domain that have the event property. To properly take into account expert events, the published expert:eventReason attribute value indicates the reason for an event. The ExpertContextReason enumeration lists the possible reasons for an occurring ExpertEvent event. Refer to the *Agent Interaction SDK 7.6 Services API Reference* for further details.
The following code snippet subscribes to both ExpertEvent and InteractionEvent.

```
/// Defining two TopicsService
TopicsService[] myTopicsServices = new TopicsService[2] ;

/// Defining a Topic Service for the callback service
myTopicsServices[0] = new TopicsService() ;
myTopicsServices[0].serviceName = "ExpertService" ;
/// Defining a topic event
myTopicsServices[0].topicsEvents = new TopicsEvent[1] ;
myTopicsServices[0].topicsEvents[0] = new TopicsEvent() ;

/// The targeted events are ExpertEvent
myTopicsServices[0].topicsEvents[0].eventName = "ExpertEvent" ;
```

```
/// All the event attributes values are propagated in Event objects
myTopicsServices[0].topicsEvents[0].attributes = new String[]{"expert:*"};

/// Triggering ExpertEvent for agent0
myTopicsServices[0].topicsEvents[0].triggers = new Topic[1];
myTopicsServices[0].topicsEvents[0].triggers[0] = new Topic();
myTopicsServices[0].topicsEvents[0].triggers[0].key = "AGENT";
myTopicsServices[0].topicsEvents[0].triggers[0].value = "agent0";

/// Defining a Topic Service for the interaction service
myTopicsServices[1] = new TopicsService() ;
myTopicsServices[1].serviceName = "InteractionService" ;

myTopicsServices[1].topicsEvents = new TopicsEvent[1] ;
myTopicsServices[1].topicsEvents[0] = new TopicsEvent() ;

/// The targeted events are InteractionEvent
myTopicsServices[1].topicsEvents[0].eventName = "InteractionEvent" ;

/// In case of a voice interaction, the interaction, and /// voice interaction attributes
values are propagated in the
/// Event object
myTopicsServices[1].topicsEvents[0].attributes = new String[]{ "interaction:*",
"interaction.voice:*"};

/// Triggering CallbackRecordEvent for agent0
myTopicsServices[1].topicsEvents[0].triggers = new Topic[1];
myTopicsServices[1].topicsEvents[0].triggers[0] = new Topic();
myTopicsServices[1].topicsEvents[0].triggers[0].key = "AGENT";
myTopicsServices[1].topicsEvents[0].triggers[0].value = "agent0";
```

For further information about events, see The Event Service.

Using Expert Contact Features

The following table presents actions of the expert contact service which provides your application with management of expert features.

Expert Context Actions and Methods

RecordAction	IExpertService Method	Description
(none)	`onCall()`	Creates an interaction and the associated expert context.
`ACCEPT`	`accept()`	Accept to process a preview call.
`REJECT`	`reject()`	Reject a callback record

RecordAction	IExpertService Method	Description
CONFIRM_STATUS	confirmStatus()	Confirms the expert context status.
REJECT_STATUS	rejectStatus()	Confirms that the expert is not handling an interaction anymore.
REROUTE	reroute()	Reroutes an interaction.

The following subsections detail the actions presented above. It also introduces the Easy New Call and Auto Mark Done features that depend on your configuration settings.

Managing On Call

The expert can receive direct calls on his or her CTI-Less phone. The expert has to notify these calls to the CTI-Less T-Server. To notify the call, the expert application calls the IExpertService.onCall() method as shown in the following code snippet:

```
// Passing the ID of a CTI-Less DN in argument
myExpertService.onCall(myDNID);
```

The onCall() method sends a message to the CTI-less T-Server to send a new interaction to the specified CTI-Less DN. It also creates an expert context for the voice interaction. Your application receives InteractionEvent and ExpertEvent events.
Then, the expert uses the voice interaction to notify his or her actions on the call. For instance, when the expert answers manually the call, he uses your application to perform an ANSWER_CALL action on the voice interaction.
To pass the expert voice-specific activity, see Voice Interactions.

Managing Preview Calls

The CTI-Less T-Server can send a call to the expert. In this case, your application gets a new interaction and a context for the preview call. It receives an InteractionEvent event and an ExpertEvent event. The InteractionEvent event propagates the NEW interaction status and the ExpertEvent event propagates the ExpertStatus.PREVIEW context status. This happens for example when an expert participates in an outbound campaign.
The expert can either accept or reject the call. Use the accept() and reject() method of the IExpertService interface in this purpose.
In the following code snippet, the expert application accepts to process the call.

```
myExpertService.accept(myInteractionID);
```

A call to the accept() method sends the call on the expert phone. Then, the expert uses the voice interaction to notify his or her actions on the call. To pass the expert voice-specific activity, see Voice Interactions.

Managing Status Request

Your application can receive status requests from the CTI-Less T-Server. In this case, your application gets an ExpertEvent event propagating the ExpertStatus.STATUS_REQUEST of the expert context associated with a voice interaction. This periodically happens when the CTI-Less T-Server requests the expert to notify his or her voice interaction's status.
The expert can choose between the ExpertContextAction.CONFIRM_STATUS or ExpertContextAction.REJECT_STATUS actions.
A call to the IExpertService.confirmStatus() method indicates the expert is still on call. A call to the IExpertService.rejectStatus() method indicates that the expert has terminated the call.
In the following code snippet, the expert application notifies that the expert no longer processes the call.

```
myExpertService.rejectStatus(myInteractionID);
```

> **Important**
>
> When the expert makes an ExpertContextAction.REJECT_STATUS action, the voice interaction is automatically released.

Managing Re-Route

Depending on your configuration settings, the expert is able to re-route calls. See the *Interaction SDK 7.6 GIS Deployment Guide* for further details.
If you properly set kworker routing options, your application can use the IExpertService.reroute() method to notify the CTI-Less T-Server of the voice interaction re-routing.

Easy New Call and Auto Mark Done

When the expert makes a phone call, he or she first creates, then dials a voice interaction on his or her CTI-Less DN using your expert application. Your application processes the voice interaction creation as for a standard one.
Depending on your configuration settings, your application can benefit from the Easy New Call feature. This feature changes the voice interaction status to TALKING at interaction creation on CTI-Less DNs. The interaction creation is less time-consuming for the expert.
To activate the Easy New Call feature, set the easy-newcall option to true. See the *Interaction SDK 7.6 GIS Deployment Guide* for further details.
When the expert hangs up a call, he or she should release and mark the voice interaction as done. Depending on your configuration settings, your application can benefit from the Auto Mark Done feature. This feature automatically marks as done released interactions for CTI-Less DNs.
To activate the Auto Mark Done feature, set the auto-markdone option to true. See the *Interaction SDK 7.6 GIS Deployment Guide* for further details.

Additional Services

This chapter presents services that provide additional information to complement other services, or to assist you in monitoring your application. For instance, the history service complements the contact service because it provides contacts' histories. The workflow service enables an agent to use workbins to put or pull interactions. The system service determines whether or not your application is correctly connected to the servers that provide interactions, contacts, and other resources.

The History Service

The *Universal Contact Server* (UCS) stores contacts' data, including the contact history. The history service gives access to a set of contact histories managed by the UCS. For each contact, its history contains a set of interactions involving the contact. Within the history service, your application can retrieve data about those interactions.
The history service is the `IHistoryService` interface defined in the `com.genesyslab.ail.ws.history` namespace. To use this service, your application works with classes and enumerations of this namespace.

History Information

A Contact's history information consists of interactions and is organized into history items. Each history item contains the history information of one interaction associated with a contact.
The `history` domain of the `IHistoryService` interface lists the accessible data of a history item:

- `history:interactionId`—ID of the interaction involved.
- `history:dateCreated`—Creation date of the interaction.
- `history:interactionType`—Type of the interaction.
- `history:sender`—Initiator of the interaction.
  If this is an incoming interaction, the name of the customer
  If this is an outgoing interaction, the name of the agent who has processed this interaction.
- `history:subject`—Subject of the interaction.

This list is not exhaustive. For a full and exhaustive description of each UCS interaction data, refer to the UCS multimedia data model document.

Your application can retrieve a history item in a `HistoryItemDTO` instance. The `HistoryItemDTO` class associates a contact with the key-value array data that contains `history` attribute keys and values describing an interaction of within the contact history. For further details, see Getting History Information.
The`history.additional` domain provides a set of additional interaction attributes. These attributes are dynamic and depend on the Configuration Layer. To get the attribute keys of this domain, retrieve the metadata name of these attributes with the `IResourceService` methods. For further details, see Interaction Attributes.

Your application can use metadata names as attribute keys when retrieving history DTOs, as, in this example: `history.additional:<metadata.name>` (where `<metadata.name>` is the string corresponding to a metadata name).

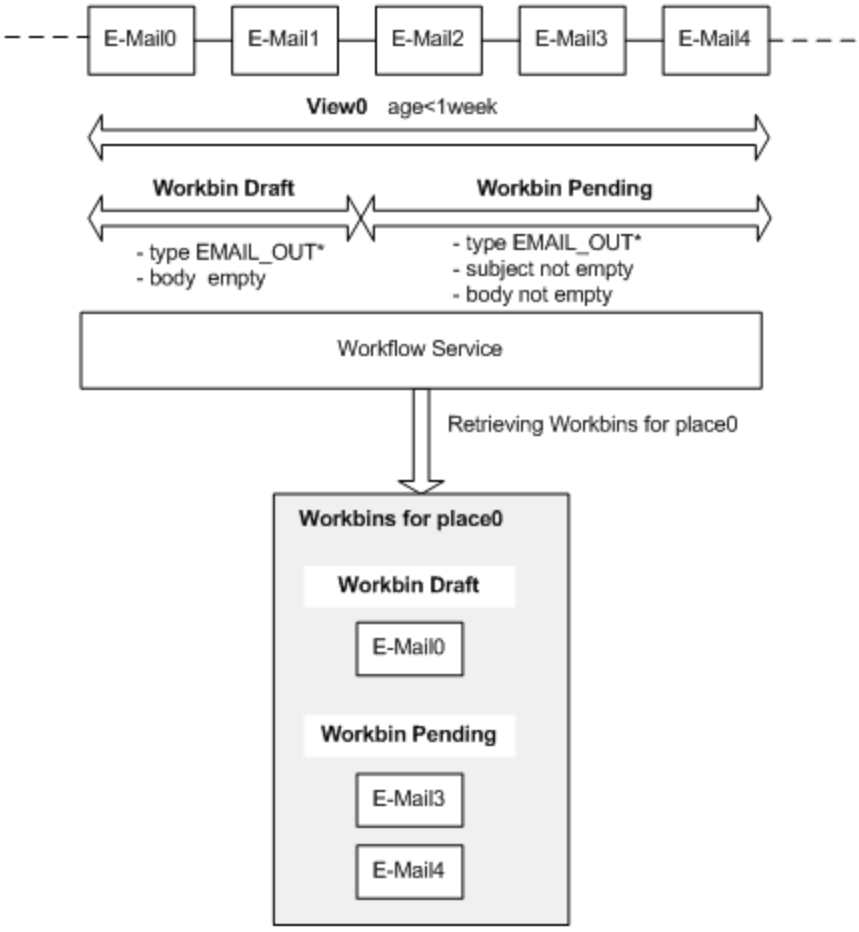
Getting History Information

To retrieve a contact's `HistoryItemDTO`, your application must first create and fill an `InteractionSearchTemplate` instance, and then call the `IHistoryService.getHistoryDTO()` method.
An interaction search template delimits the retrieved interaction history items. Your application can set the index and numbers of history items, and can specify which attributes to retrieve in the DTO result. These features are useful to (for example) display the interaction data of a contact history by pages with a limited set of attributes.
The following code snippet retrieves the first 10 interactions of a contact history.

```
///Filling the template
InteractionSearchTemplate mySearchTemplate = new InteractionSearchTemplate();
mySearchTemplate.index = 0;
mySearchTemplate.length = 10;

/// Retrieving the history item DTOs for the ten first contacts
HistoryItemDTO[] myItems = myHistoryService.getHistoryDTO("my contact ID",
mySearchTemplate, true, new string[]{"history:*"}); // Each DTO contains all the // historic
attribute values.

/// Displaying the retrieved historic data
foreach( HistoryItemDTO item in myItems)
{
   System.Console.WriteLine("— History item —\n");
   foreach(KeyValue myPair in item.data)
   {
      System.Console.WriteLine("Key: "+myPair.key + " Value: "+ myPair.value.ToString()+"\n");
   }
}
```

The Workflow Service

The workflow service provides your application with workbins. From an agent's point of view, a workbin is a sort of interaction directory from which your application can pull, or into which it can put, interactions.
To define workbins more precisely: a queue contains interactions, and a view filters a queue's interactions according to a set of criteria. A workbin filters a view's interactions according to a further set of criteria. The following diagram shows an example of a view and workbins defined for a queue.



Example for Workbins, Views, and Queues

This example shows a queue containing e-mail interactions. For this queue, `View0` lets your application see only e-mail interactions that are no older than a week.
In this view, two workbins coexist: one for draft e-mail interactions, and one for pending e-mail interactions. The workflow service can use those filters to retrieve a set of interactions organized in workbins for a particular place.
Here, the workflow service retrieves interactions no older than a week and available for `place0`. `E-Mail1` and `E-Mail2` are not retrieved as they should not be treated in `place0`.
Use the Configuration Layer to define views and workbins. For further details, refer to your Configuration Layer Documentation.
The workflow service is the `IWorkflowService` interface defined in the `com.genesyslab.ail.ws.workflow` namespace. To use this service, your application works with classes and enumerations of this namespace.
Use the workflow service to:

- Display workbins and their filtered interactions.
- Enable an agent to put an interaction in a workbin.

Handling a Workbin Interaction

By default, all interactions contained in a workbin are in the `IDLE` state. To handle a workbin's interaction, you must pull it using `IInteractionService.openInteractionForAgentDTO()` or `IInteractionService.openInteractionForPlaceDTO()` methods from the `IInteractionService` interface. For more details, see Opening a Workbin Interaction.

Workbin Information

Workbin information is composed of a set of attributes and their content in terms of interactions.

Workbin Attributes

The workflow service defines a workbin's information in the `workbin` domain. The following list is not exhaustive:

- `workbin:id`—The system identifier of a workbin; used to retrieve all information about a workbin.
- `workbin:viewId`—The system identifier of the view that contains a workbin.
- `workbin:placeId`—The system identifier of the place that contains a queue's views.
- `workbin:type`—A workbin's type.
- `workbin:reason`—A string to display when an event occurs on the content of a workbin.

Workbin DTO

Your application can retrieve a workbin's attributes and content—with respect to a particular place or agent—using a `WorkbinDTO` instance. The `WorkbinDTO` class has the following attributes:

- `workbinId`—System ID of a workbin.
- `data`—Key-value list of workbin attributes.
- `worbinInteractionsDTO`—Array of DTOs; each DTO contains the information for an interaction of the workbin.

Workbin Interaction Information

The level of information provided for a workbin interaction depends on the type of interaction. The workflow service provides more data for multimedia interactions—that is, `CHAT*`, `COLLABORATION*`, or `EMAIL*` interactions.
The workflow service includes two domains of attributes for interactions:

- `workbin-interaction`—Attributes summarizing an interaction.
  - `interactionId`—System ID of an interaction.
  - `interactionType`—Type of interaction.
  - `subject`—Subject of an interaction.

- `workbin-interaction.multimedia`—Additional information specific to an multimedia interaction. The following list of attributes is not exhaustive:
  - `from`—Sender field of an interaction.
  - `to`—Receiver field of an interaction.
  - `contactId`—System ID of a contact in the UCS. See also The Contact Service.

To get information about workbin interactions, use one of the `getWorkbins*()` methods of the `IWorkflowService` interfaces.

Getting Information

The following table presents the methods of the `IWorkflowService` interface, which accesses workbin information.

Methods to Get Workbin Information

IWorkflowService Methods	Description
`getQueues()`	Gets simple containers for queues information.
`getWorkbinsDTO()`	Gets workbin DTOs for a particular place or agent (including workbin information and workbin interactions DTO).
`getWorkbinsContentDTO()`	Gets only the interactions contained in a workbin for a particular place or agent.
`getWorkbinsContentForAllDTO()`	Gets all interactions contained in a workbin (regardless of agent or place).

For example, the following code snippet uses the `IWorkflowService.getWorkbinsDTO()` method to retrieve information from all the workbins defined for a view for `place0`. For each workbin, it displays the DTO content, including workbin interaction DTOs.

```
///Getting Workbin DTO for place0
WorkbinDTO[] myWorkbinsDTO = myWorkflowService.getWorkbinsDTO("place0",
   null, /// all workbins are retrieved
   new string[]{"workbin:*"},///with all info for each workbin
   new string[]{"workbin-interaction:*",        ///with all info for each
   "workbin-interaction.multimedia:*"}); /// contained ixn

   ///Displaying each Workbin DTO content
   foreach(WorkbinDTO myDTO in myWorkbinsDTO)
   {
      /// Displaying workbin ID:
      System.Console.WriteLine("Workbin: "+myDTO.workbinName+"\n");

      /// Displaying workbin data: (workbin domain)
```

```
      foreach(KeyValue myWorkbinPair in myDTO.data)
      {
         System.Console.WriteLine("key: "+myWorkbinPair.key
            + " value:"+myWorkbinPair.value.ToString()+"\n");
      }

      /// Displaying DTO content for each workbin interactions
      foreach(WorkbinInteractionDTO myWorkbinIxn in myDTO.workbinInteractionsDTO)
      {
         /// Displaying interaction ID:
         System.Console.WriteLine("Ixn Id: " +myWorkbinIxn.workbinInteractionId+"\n");

         /// Displaying data of current interaction: /// (workbin-interaction.*:* domain)
         foreach(KeyValue myWorkbinIxnPair in myWorkbinIxn.data)
         {
            System.Console.WriteLine("key: "+myWorkbinIxnPair.key +
               " value:"+myWorkbinIxnPair.value.ToString()+"\n");
         }
      }
   }
```

The System Service

The system service is the `ISystemService` interface of the `com.genesyslab.ail.ws.system` namespace. This service informs your application of the state of the Genesys servers connected to your client application. This service includes:

- Methods to retrieve information about the currently connected servers.
- Events to inform your application of real-time changes about the connected servers, using `SystemEvent`.

For further information about the Genesys servers, refer to your Genesys documentation.

Server Information

System service attributes provide your application with information about the connected servers that perform services requests. Your application can either retrieve these attributes with the `ISystemService` interface methods, or propagate them in `SystemEvent`.
The following are the `ISystemService` interface's:

- `system.server-info:name`—A `string` for the AIL Framework service name.
- `system.server-info:host`—A `string` for the name of the host where the service should run.
- `system.server-info:port`—A `string` for the port on which the service should run.
- `system.server-info:status`—The `ServerStatus` value.
- `system.server-info:type`—The `ServerType` value.
- `system.server-info:switch`—The `Switch` object associated with this server (if the type is a telephony type); else `null`.

Server Type

Each server is associated with a service, and this association is provided in the `system.server-info:type` attribute. The `ServerType` enumeration lets your application with which server the application is connected through the server-side application. The following table comments the existing `ServerType` values.

Server and Services

ServerType	Description
`CHAT`	The server manages the chat service.
`TELEPHONY`	The server manages connections to voice media.
`CONFIGURATION`	The Configuration Layer server manages the configuration.
`DATABASE`	The server is the Universal Contact Server.
`STATISTIC`	The server manages the statistics service.
`AIL`	The server manages the server-side application connected to the Genesys Framework. Your client application interacts with this server.
`IS`	The server is the Interaction Server which manages voice, e-mail, and chat interactions.

Server Status

Combined with the server type attribute, the server status attribute lets your application determine whether a service is available or not. The service associated with the server is enabled if the `system.server-info:status` attribute's value is `ServerStatus.ON`.
For other `ServerStatus` values, refer to the *Agent Interaction SDK 7.6 Services API Reference*.

Retrieving Server Information

To retrieve servers' information, use the getServersDescriptionDTO() method of the `ISystemService` interface. This method takes a list of attribute keys as its parameter, and retrieves an array that contains information for all available servers. Each `ServerDescriptionDTO` object of the array contains the key-value pairs that corresponds to the key list.
The `ServerDescriptionDTO` class associates the server name with the server's information. This class has the following attributes:

- name—The server name.
- data—A key-value array containing the attribute values.

The following code snippet shows how to call the getServersDescriptionDTO() method, and how to then display the contents of the DTO objects retrieved for the available servers.

```
/// Retrieving the DTOs
ServerDescriptionDTO[] myServerDescriptionDTOs =
mySystemService.getServersDescriptionDTO(new string[]{"*"});

/// Displaying the content of each DTO
foreach(ServerDescriptionDTO description in myServerDescriptionDTOs)
{
   System.Console.WriteLine(description.name+": ");
   // Displaying name and value for each attribute
   foreach(KeyValue pair in description.data)
   {
      System.Console.WriteLine(pair.key +" - "+pair.value.ToString());
   }
}
```

System Events

The SystemEvent event of the ISystemService interface occurs when a value of one of this interface's published attributes has changed. (The published attributes are all ISystemService attributes.)
See also The Event Service.

The Resource Service

The resource service provides access to configuration data defined in the Configuration Layer. For example, your application can use the resource service to retrieve action codes, DN information, and interaction attribute metadata.
The resource service is the IResourceService interface defined in the com.genesyslab.ail.ws.resource namespace. To use this service, your application works with classes and enumerations of this namespace.

Resource Information

Two domains are defined to provide your application with resource information:

- resource.dn —DN summary data.
- resource.common —Common data.

The following subsections describe those domains.

DN Summary

Your application can use the resource service to get information about the DNs of a switch without registering those DNs. This information corresponds to DN summaries and is defined in the

resource.dn domain.
Retrieve this information with the IResourceService.getDnSummariesDTO() method.The information is returned in a DnSummaryDTO object. This class associates the DN identifier with this DN's DTO data.

Common Information

Your application can retrieve two types of common information defined in the resource.common domain:

- resource.common:actionCodes —The codes that can be used to specify a reason in calls to the agent service methods. All action codes are defined in the ActionCodeType enumeration.
- resource.common:incomingAddresses —The call center's e-mail addresses.

Use the IResourceService.getCommonsDTO() method to retrieve the corresponding attributes as key-value pairs.

Enumerators

Use enumerators to get the key-value pairs of attributes defined in the Configuration Layer. The Enumerator class describes an enumerator and contains its associated values:

- defaultValue —The default value of this enumerator, if any.
- description —The enumerator's description.
- enumeratorId —The name of this enumerator.
- type —The type of this enumerator; null if no type is defined.
- values —The values of this enumerator. Each value of an enumerator is an EnumeratorValue instance.

Use the IResourceService.getCommonsDTO() method to retrieve the enumerators defined in the Configuration Layer.

Interaction Information

The Configuration Layer defines interaction attributes and interaction custom properties in the Business Attributes section. Your application can get the corresponding metadata objects using the resource service and access the corresponding values in the attached data of an interaction using the metadata name as an attached data key. See Attached Data.

Interaction Attributes

Values

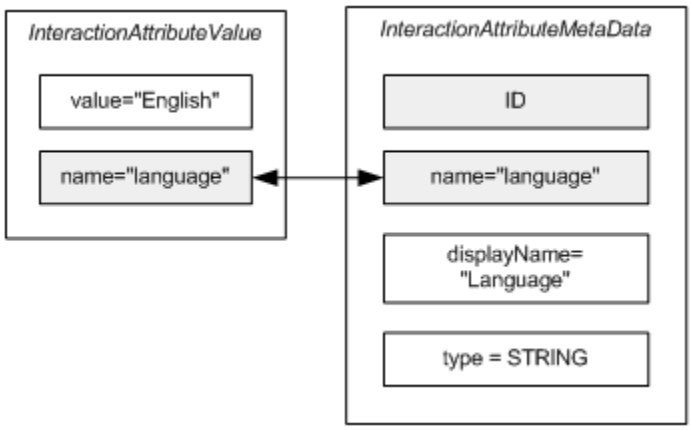
The InteractionAttributeValue class characterizes the information in an interaction attribute's value:

- id—The system ID of the interaction attribute's value.

- `name` —The name of the corresponding interaction attribute's metadata.
- `value` —The value of the interaction attribute.
- `description` —The description of the interaction attribute's value.
- `default` —`True` if the interaction attribute value's is a default value.
- `type` —The type of interaction attribute.

Metadata

Each interaction attribute value has a type that specifies the corresponding interaction attribute. This type resides in a `InteractionAttributeMetaData` instance, as shown below.



Interaction Attribute Value and Metadata

The `InteractionAttributeMetaData` class' main fields are the following:

- `id`—The unique system identifier for this metadata.
- `name`—The unique attribute name.
- `active`—`True` if the attribute is active in the Contact Server.
- `displayName`—The attribute display name.
- `predefinedValues`—A list of predefined contact attribute values, or `null`.
- `sortable`—`True`, if the attribute can be used to sort the contacts. For example, a last name may be sortable.
- `type`—Type of the corresponding attribute values defined in the `InteractionAttributeMetaDataType` enumeration

Your application can retrieve interaction attributes' metadata using the following `IResourceService` methods:

- `getInteractionAttributeMetaDataById()`—To retrieve metadata with their IDs.

- `getInteractionAttributeMetaDataByName()`—To retrieve metadata with their names.

Then, to get interaction attribute values, use the metadata name as the key when retrieving `history.additional` data with the history service. See also History Information.

Custom Attached Data

The `Interaction Custom Properties` in the Configuration Layer correspond to the `CustomAttachedData` objects that your application can retrieve using the `IResourceService.getCustomAttachedDataByXxx()` method.
The `CustomAttachedData` class describes a metadata for a custom property. This class includes method to get the corresponding name, display name, and description of a custom property. It also provides the predefined values for the custom attached data (if any).
Call the `CustomAttachedData.getName()` method to get the name of a custom property and use it as a key to access or modify the corresponding value in an interaction's attached data map. See Attached Data.

Getting Resource Information

The following code snippet shows how to get and display some common information.

```
///Getting Common DTOs
KeyValue[] myCommonDTOs = myResourceService.getCommonsDTO(new string[]{"resource.common:*"});

///Displaying the common DTOs
foreach(KeyValue myCommonDTO in myCommonDTOs )
{
   /// If action codes, displaying them
   if(myCommonDTO.key == "resource.common:actionCodes")
   {
      ActionCode[] myActionCodes = (ActionCode[]) myCommonDTO.value;
      System.Console.WriteLine( "Configuration Layer: agent actions codes");
      foreach(ActionCode myCode in myActionCodes)
      {
         System.Console.WriteLine(myCode.ToString());
      }
   }
   /// If e-mail addresses, displaying them
   else if(myCommonDTO.key == "resource.common:incomingAddresses")
   {
      System.Console.WriteLine( "Configuration Layer: Call Center addresses");
      string[] myCallCenterAddresses = (string[]) myCommonDTO.value;
      foreach(string myAddress in myCallCenterAddresses)
      {
         System.Console.WriteLine(myAddress.ToString());
      }
   }
}
```

The Monitor Service

The `MonitorService` interface provides monitoring features for agent status. With this service, you can subscribe to an agent and get real-time information about that agent's status (which is available in the `AgentCurrentState` category from Stat Server).

Monitor Information

MonitorService information consists of a set of current-state data:

- monitor-status:agent —The current status of an agent and all his or her media.
- monitor-status:media —The current status of a media associated with an agent.

Getting Monitor Information

To access monitor information, you can either subscribe to StatusEvent (details on events are in The Event Service), or you can use the PeekStatus method. The two snippets that follow correspond to these two options, respectively.

Subscribing to StatusEvent

```
/// Creating topic objects for the monitor service
TopicsService [] topicServices = new TopicsService[1];
topicServices[0] = new TopicsService();
topicServices[0].serviceName = "MonitorService";
topicServices[0].topicsEvents = new TopicsEvent[1];
topicServices[0].topicsEvents[0] = new TopicsEvent();

// Creating a topic event for status events
topicServices[0].topicsEvents[0].eventName = "StatusEvent";
topicServices[0].topicsEvents[0].attributes =
new String[]{"monitor-status:agent","monitor-status:media"};
topicServices[0].topicsEvents[0].triggers = new Topic[1];
topicServices[0].topicsEvents[0].triggers[0] = new Topic();
topicServices[0].topicsEvents[0].triggers[0].key = "STATUS";
topicServices[0].topicsEvents[0].triggers[0].value = mValue;
//objectType:objectid:notificationMode:notificationValue
```

Values

- objectType—type of the object (PERSON or QUEUE).
- objectId—identifier of the object.
- notificationMode—CHANGED_BASED orTIME_BASED.
- notificationValue—the value of the notification.
  - CHANGED_BASED —an event will be sent only if the value changes by more than this notificationValue.
  - TIME_BASED—an event is sent every notificationValue seconds.

> **Important**
> If the parameter contains a ":" (colon) character, escape it with a "\" (slash) character (for example, Agent:100 becomes Agent\:100).

Example

Here is an example of the full parameter string you might use:

```
PERSON:Agent10:CHANGED_BASED:10
topicServices[0].topicsEvents[0].filters = null;
```

Using the PeekStatus Method

```
//Retrieving an Agent status:
MonitorEventStatus myMonitorEventStatus = myMonitorService.peekStatus(PERSON, myAgentId)

//Display this agent's status data:
System.Console.WriteLine(myMonitorEventStatus.agentStatus.userName+":
"+myMonitorEventStatus.agentStatus.agentStatus.ToString());

//and all his media statuses
foreach(MonitorEventMediaStatus mediaStatus in myMonitorEventStatus.agentStatus.mediaStatuses)
{
   System.Console.WriteLine(mediaStatus.name+": "+mediaStatus.mediaStatus.ToString());

}
```

Best Coding Practices

This chapter is for developers who are familiar with the Agent Interaction Services. It reviews the rules you would otherwise find throughout this book for developing a high-performance application on top of the Agent Interaction Services.

Introduction

When you develop an agent application on the Agent Interaction SDK Services API, observing a few basic rules will help optimize your application's performance in your production environment.
The Agent Interaction Services API is not a traditional API, and its service complexity is hidden behind the service interfaces. For instance, as detailed later in this chapter, calling the get methods reduces the code complexity but increases the network traffic.
This chapter is intended to help you make a few checks and corrections to your applications before you start testing them in production environments.

Avoid Wildcards

Although the Agent Interaction Services API includes wildcards to deal with with DTOs (see DTOs and Wildcards) or events (see Wildcards), they should be used for your application’s development only.

The * Wildcard

In a production-like environment, the * wildcard should be avoided if you want fast application responses. This wildcard is likely to disturb the network traffic and slow down your application’s performance, (since it usually retrieves wide data sets).

The default Wildcard

If you get more attributes than you need with the `default` wildcard, or, on the contrary, if the `default` wildcard does not retrieve enough attributes, you should instead specify the exact list of attributes that you need.
Note that the `default` wilcard can be use in production-like environment. The related attributes should not cause any performance issue.

No Wildcards

Get exactly what you need, neither more, nor less, so that your application can work without significantly increasing network activity.
Achieve the best performance for your application by removing all wildcards from your code before you start load-testing it.

Avoid This:

```
// Creating a topic event for voice media events
TopicsEvent voiceMediaTopicsEvent = new TopicsEvent();
voiceMediaTopicsEvent.eventName = "VoiceMediaEvent";
voiceMediaTopicsEvent.attributes = new String[]
{
   "agent:*"
};
```

Do This:

```
// Creating a topic event for voice media events
TopicsEvent voiceMediaTopicsEvent = new TopicsEvent();
voiceMediaTopicsEvent.eventName = "VoiceMediaEvent";
voiceMediaTopicsEvent.attributes = new String[]
{
   "agent:voiceMediaInfo",
   "agent:dnActionsPossible"
};
```

Tips for Events Processing

A Single Subscriber

You need a single subscriber per client application. That is, a single instance of SubscriberResult for your application at runtime. Once it is created, you should use this reference for all your event subscriptions and un-subscriptions.
For details about how you get this instance, see Subscribing to the Events of a Service.

About Subscriptions

When you subscribe to a set of events (see Building TopicsEvent), it is important to register for the minimum number of attributes, because this data travels through the network each time that an event reports a modification. Given this, you should also design your application to modify subscriptions, asking for additional attributes only when needed.

About Notification

As a general guideline, to make sure that notification works fine:

- **Do not** perform extensive event processing in the methods related to the notification.
- **Avoid** calls to service methods during notification processing; these calls retrieve data through the network and slow down event publishing, and may even block it.

If you want to perform an extended treatment, or a treatment for making calls to services methods, be sure that your application implements this code in a separate thread.

Tips for DTOs

Guidelines

Each time you call a `XxxService.getXxxDTO()` or `XxxService.setXxxDTO()` method, your application makes a server request. As a result, data travels through the network. Even if your application repeats an identical call to a `XxxService.getXxxDTO()` method, the request is repeated and, again, data travels through the network.
The Agent Interaction Services API does not cache data on the client's side, and it does not keep any reference either. So, when you develop your application, you should take into account the cost of each server request.
For instance, if your application uses some agent data in an information panel, do not request agent DTOs each time that the user displays the panel. You are better off keeping a local variable and updating data with agent events.
You can also create a cache locally to save the network bandwidth. For instance, contact data does not often change, so your application could retrieve contact data at startup.

Additional Details

You may notice that some DTOs, methods, or attributes retrieve wide collections of objects or heavy data (such as binaries and attachments). When using items from Critical Areas, it is critical that you understand that their improper use can lead to significant increases in network traffic.

Critical Areas

Service	Methods, DTOs, or Attributes
`IHistoryService`	`IHistoryService.getHistoryDTO()`
`ISRLService`	`StandardResponseDTO` `srl:attachments`
`IWorkflowService`	`workbin-interaction:attachedData` `IWorkflowService.getWorkbinsContentForAllDTO()`
`IInteraction`	`interaction:attachedData` `interaction:contentBinary`

`IInteractionMailService`	`interaction.mail:attachments`
	`interaction.mail:mimeMessage`
	`interaction.mail:structuredText`

Tips

- When your application calls methods that retrieve wide collections of items (SRL, history, workflow, interactions), you can reduce network activity with pagination. For instance, when your application gets `HistoryItems`, set up a positive value for the `length` field of the `InteractionSearchTemplate` that you pass as an argument of the `IHistory.getHistoryDTO()` method.
- When your application retrieves interactions lists, it should specify attribute names for `ShortAttachment` values instead of `Attachment`. Each `ShortAttachment` instance describes an `Attachment` and provides enough information to be useful for the user at first glance.Your application should request attachments only when the user explicitly asks for them.

Additional Information About Workbins

The `IWorkbinService` methods collect wide sets of interaction data, so use special care with attributes passed in as arguments. In particular, the `getWorkbinsContentForAllDTO()` method retrieves workbins for **all** agents and is very dangerous in regards to the number of interactions concerned and the list of attributes your application can ask for each of them.

Tips for High Availability

The Agent Interaction Services offer high availability embedded in the Genesys Integration Server. High availability is not your responsability, but you should take into account the following guidelines during the implementation of your application.

Voice Interactions in Specific Service Status

For voice interactions, `NEW` and `RELEASED` statuses are specific to the Agent Interaction Services. Interactions in these statuses do not exist in the T-Servers. So, after a switchover, it is impossible to recover them from the T-Servers and they are lost.
The following tips should help you avoid losing interactions.

Interactions in NEW Status

Interactions are assigned the `NEW` status when `MakeCall` is invoked in two steps, the first step being the interaction creation, the second being the dialing step. The solution is to use the one-step `MakeCall` in which the destination directory number (`destDn`) is provided:

```
InteractionVoiceErrorDTO err = interactionVoiceService.createInteractionFromDnDTO(
   sourceDn , null , destDn ,null ,
   MakeCallType.REGULAR , attachedData , null , null );
```

Interactions in RELEASE Status

Interactions are assigned the `RELEASED` status when T-Server releases them. At this point they still exist in the Agent Interaction Services to allow agents to add data and comments. These

modifications are saved in the UCS database when your application calls the `InteractionVoiceService.markDone()` method.
In case of switchover, these interactions are lost and cannot be saved. To avoid losing comments, they should be saved before the call is released and your application should mark done the interaction as soon that the RELEASED event is received:
`interactionVoiceService.markDone(interactionId);`

E-Mail and Open Media Interactions

In case of switchover, the outgoing e-mail and open media interactions are no longer associated with the agent's Place. Thus, your application needs to call the `InteractionService.openInteractionForPlaceDTO()` method in order to retrieve these interactions and make them available for the agent.

Update Status

Your application should subscribe for the event sent to the event service. If your application's subscriber changes its GIS node, this event's attribute `subscriber:isSubscriberChangeNode` is set to `true`.
In this case, your application should retrieve data to update the status of agents, places and medias, and interactions as shown in the following code snippet.

```
if (event.serviceName.Equals("EventService")) {
   bool isSubscriberChangeNode = false;
   foreach( KeyValue attribute in event.attributes) {
      if (attribute.key.equals("subscriber:isSubscriberChangeNode")) {
          isSubscriberChangeNode = true;
      }
   }
   if (isSubscriberChangeNode ) {
   // open a window to signal the event to the agent
   // update voice interactions
   InteractionPlaceDTO[] interactions =
      interactionService.getInteractionsDTOFromPlace(
      new String[] { placeId }, new String[] {"default"} );
   for (int i=0; interactions!= null && i < interactions.Length; i++) {
   InteractionPlaceDTO ia = interactions[i];
   for (int j=0; j < ia.interactionsDTO.Length; j++) {
      InteractionDTO item = ia.interactionsDTO[j];
      for (int k=0; k< item.data.Length; k++) {
         String key = item.data[k].key;
         Object obj = item.data[k].value;
         //...
      }
   }
}

// update agent status
PersonDTO[] persons = agentService.getPersonsDTO(
   new String[] { "smith" }, // agentId
   new String[] {"agent:loggedDns", "agent:loggedMedias"} );
for (int i=0; persons!= null && i < persons.Length; i++) {
   PersonDTO personDTO = persons[i];
   if (personDTO.data.Length > 0) {
      for (int j=0; j< personDTO.data.Length; j++) {
         String key = personDTO.data[j].key;
         if (key.Equals("agent:loggedDns")) {
```

```
            VoiceMediaInfo[] voiceMediaInfos = (VoiceMediaInfo[])personDTO.data[j].value;
            for (int k=0; k< voiceMediaInfos.Length; k++) {
            VoiceMediaInfo voiceMediaInfo = voiceMediaInfos[k];
            String dnId = voiceMediaInfo.dnId;
            VoiceMediaStatus voiceMediaStatus = voiceMediaInfo.status; // NOT_READY READY ...
            // update status
          }
      }
    if (key.Equals("agent:loggedMedias")) {
      MediaInfo[] mediaInfos = (MediaInfo[])personDTO.data[j].value;
      for (int k=0; k< mediaInfos.Length; k++) {
         MediaInfo mediaInfo = mediaInfos[k];
         String mediaName = mediaInfo.name;
         MediaStatus mediaStatus = mediaInfo.status; // NOT_READY READY ...
        // update status
    }
}
```

The Agent Status Example

This chapter details the implementation of the agent status example, an agent application based on services, which is available on the documentation CD in the `sdk_exmpl_services-agent.zip` file. This example is developed in C#. It is a GUI form that monitors the status of an agent on a place and allows agent actions on this place, such as login, logout and so on. This chapter discusses the example's architecture and the integration of the Agent Interaction Services into this GUI application.

Introduction

The agent status example is a .NET Framework windows form application based on the Agent Interaction SDK (Web Services). It provides you with a C# example for integrating the services into a GUI application.
This example is an agent application for managing an agent's place:

- It displays the agent status on the media and DNs of his or her place.
- It performs agent actions—login, logout, ready, and not ready—on media and DNs of an agent's place.
- It refreshes in response to events propagated from the integrated services.

To understand what are the agent's place, DNs, and media, see Understanding Place, DNs, and Media.
In order to get and update DNs and media information, this example integrates the services presented in the following table.

Integrated Services

Service Name	Integration Purpose	Further Details
`ServiceFactory`	Connects and creates the services used in this example.	About the Examples.
`IAgentService`	Manages the agent actions on the place (such as login, logout, ready, and not ready) and accesses agent data.	The Agent Service.
`IPlaceService`	Retrieves information about the place.	Place, DNs, and Media.
`IEventService`	Subscribes to and gets events.	The Event Service.

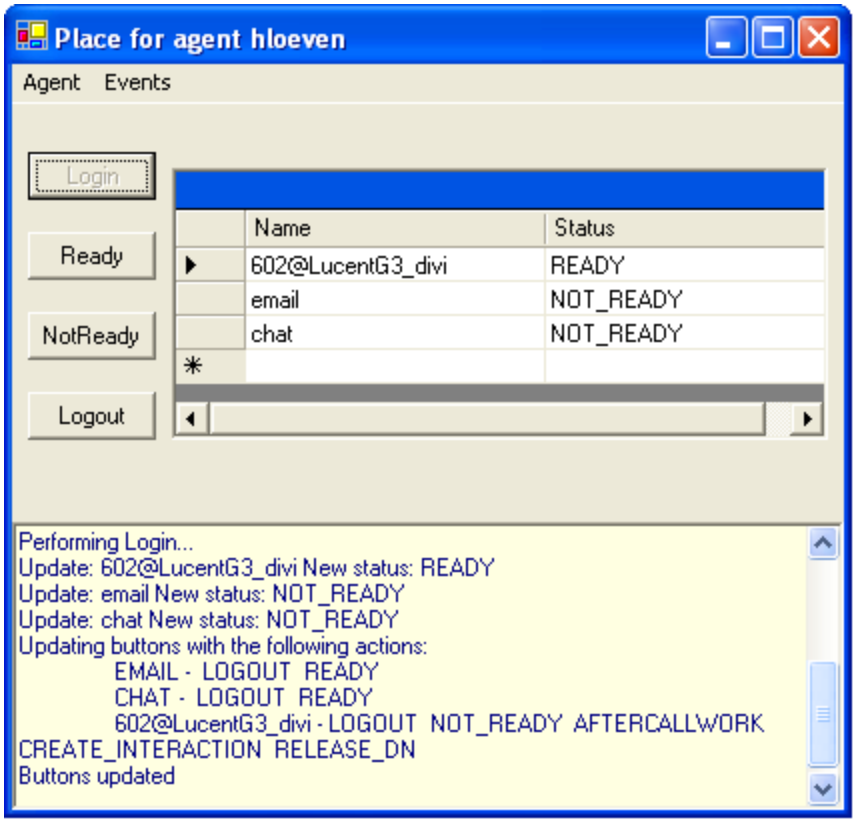
Agent Status Short Description

Agent Status GUI Description

The agent status example is a `System.Windows.Forms.Form` instance that includes the following `System.Windows.Forms` components:

- A `DataGrid` object to display the information about the agent's place.
- Buttons to perform agent actions on the place (such as login, logout, ready, and not ready).
- A `RichTextBox` to display some traces.
- A `MenuBar` object with `MenuItem` objects to:
  - Switch agents and monitor a new agent place.
  - Switch event modes (push or pull, see Getting Events).

For further details on `System.Windows.Forms` component, refer to Microsoft .NET Framework help. The following screenshot shows the agent status application at runtime, when an agent is logged in on his or her place.

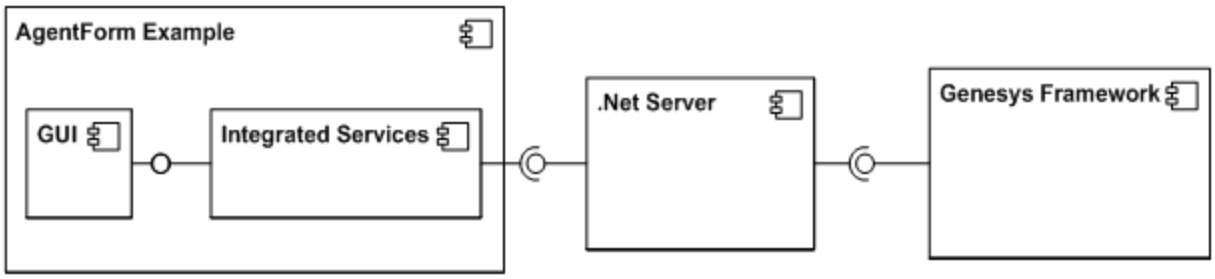


The Agent Status Example at Runtime

In the above screenshot, the user has clicked the `Login` button and logged in on the place. The application refreshed the `DataGrid` and buttons with the new agent status and the possible agent actions on the place.

Agent Status Architecture Overview

The agent status example separates the GUI classes from the classes that integrate the services, as shown below.



Agent Status Architecture Overview

To refresh or get information, GUI classes interface with the classes that integrate the services. This architecture makes it easier to concurrently manage GUI and services complexity.

AgentStatusExample Project

Project Structure

Unzip the contents of the `sdk_exmpl_ixn_services-agent.zip` archive to get the `AgentStatusExample` directory, which contains two directory structures:

- The `AgentStatusExample` directory contains the MS Visual Studio project files:
  - `AgentStatusExample.cspro` j—The Agent Status Example project file.
  - `AgentStatusForm.cs` —The source file for the application form.
  - `LoginForm.cs` —A dialog box source file.
  - `Global.cs` —The source file for the classes integrating the Agent Interaction SDK Services.
- The `ExternalDependencies` directory contains all the references you need. To implement this example, copy the following files (available on the product CD) into this directory:
  - `ail -configuration.xml` —The XML configuration file.
  - The .NET proxy `AilLibrary.dll` available on the product CD.

Before You Start

1. Set the properties of the ail-configuration.xml file. Refer to About the Examples for instructions on what to modify.
2. Open the `AgentStatusExample` project in Visual Studio .NET.
3. Set the `ExternalDependencies` directory as your working directory:
   - Select `Project > AgentStatusExample Properties`.
   - Select `Configuration Properties`.
   - Select `Debugging`: In the `Start Options` section, assign your `ExternalDependencies` directory to

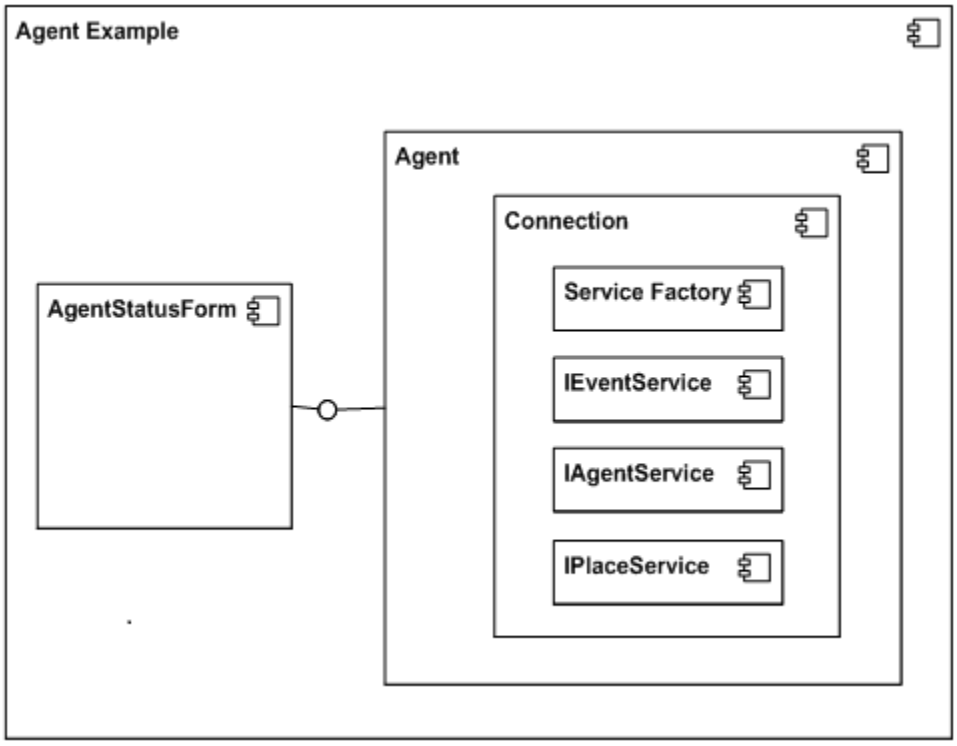
the `Working Directory` option. This ensures that the running application takes into account the correct `ail-configuration.xml` file.

4. Make sure that all references point to the `ExternalDependencies` directory.
5. If your Genesys Interface Server integrates an AIL version prior to `7.0.104.00`, uncomment the specified code in the `AgentStatusForm.UpdateButtons()` method of the `AgentStatusForm.cs` file.

You can now build and start the application.

Agent Status Architecture

The architecture of the agent status example integrates the services in classes separated from the GUI part of the application.
The Component Diagram presents the main components of this example.



The Agent Status Example Component Diagram

Service Components

Two classes deal with the Agent Interaction Service API:

- The `Connection` class:

  - Manages the connection with the GIS through the `ServiceFactory` component
  - Creates the services.
  - Includes facilities to hide DTO complexity .

    > **Important**
    > See Data Transfer Object for further information about DTOs.

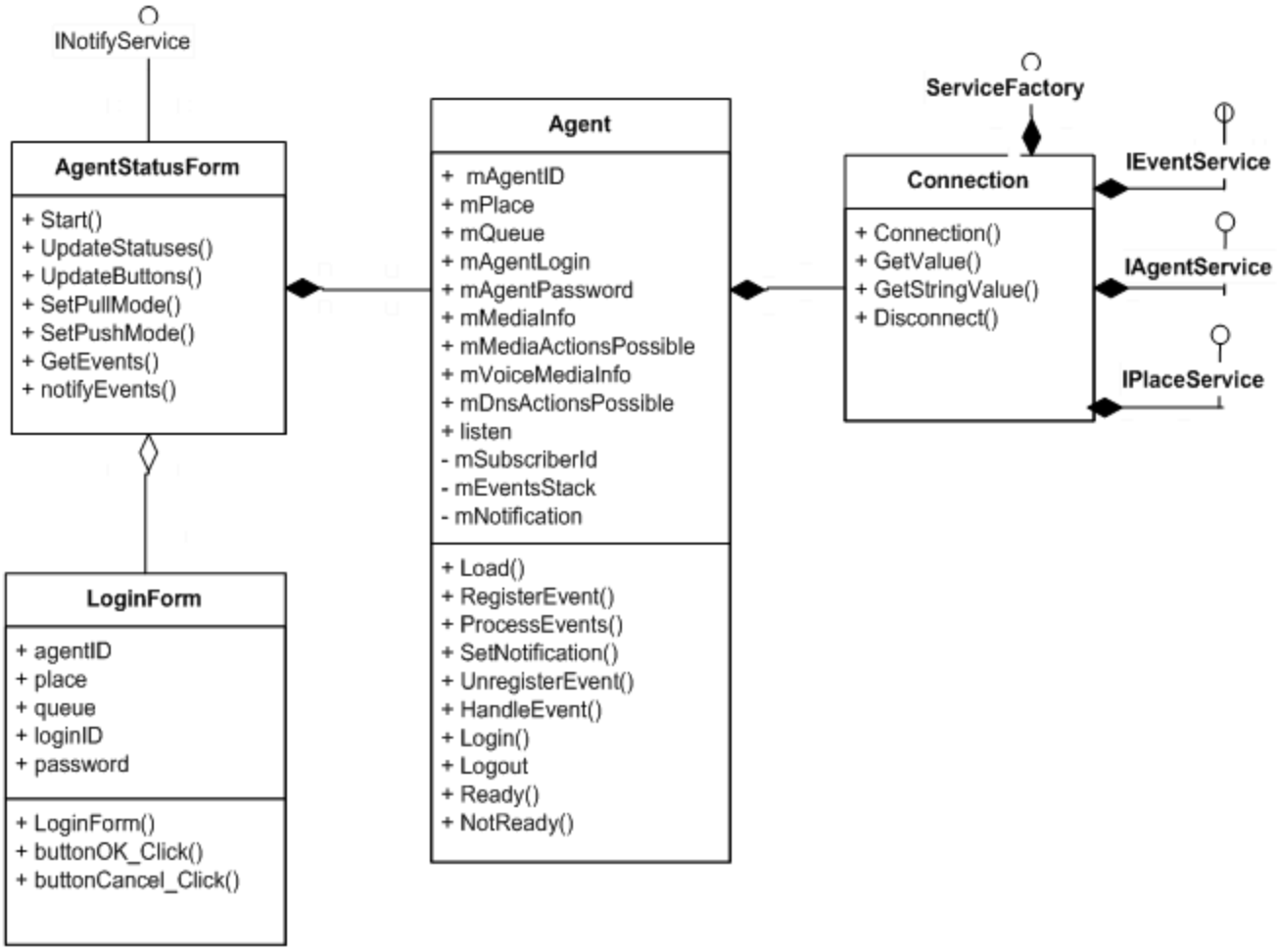
- The `Agent` class uses the services created with a `Connection` instance to:
  - Retrieve agent and place information for a particular agent.
  - Manage DNs and media events (including subscription) on this agent.
  - Perform agent actions on this agent's place.

GUI Component

The `AgentStatusForm` class is a `System.Windows.Forms.Form` class that handles the runtime form presented in The Agent Status Example at Runtime. This class interfaces with the `Agent` class to monitor an agent's place.

Agent Status Classes

The following diagram presents all the classes of the agent status example, with all the relationships existing between classes and the Agent Interaction SDK (Web Services).



Class Diagram of the Agent Status Example

All the classes are available in the following project files of the agent status example:

- `AgentStatusForm.cs` —The source file for the `AgentStatusForm` class.
- `LoginForm.cs` —The source file for the `LoginForm` class.
- `Global.cs` —The source file for the `Agent` and `Connection` classes.

The following subsections present details about these classes.

Class Connection

The `Connection` class manages the connection to the GIS and creates the services used in this application example.

Connection Attributes

The `Connection` attributes include the factory and services as public members:

- `mServiceFactory` —An instance of the factory handling a connection.
- `mAgentService` —An instance of the agent service.
- `mPlaceService` —An instance of the place service.
- `mEventService` —An instance of the event service.

Connection Methods

The `Connection` methods are the following:

- `Connect()` — The constructor; creates the factory and the services.
- `Disconnect()` —Releases the factory.
- `GetValue/GetStringValue()` —Methods for getting the value of an attribute available in a DTO.

Class Agent

The `Agent` class gathers and updates agent data, that is, information about the media and DNs of a place associated with an agent.

Agent Attributes

To access and manage agent and place information, the `Agent` class uses a connection instance—`mConnection` —for accessing services.
The following table divides the other `Agent` attributes into three categories.

Agent Attributes

Attribute Type	Attributes	Description

Agent property	`mAgentId`
	`mAgentLogin`
	`mQueue`
	`mPlace`
	`mAgentPassword`
		These attributes define which agent and place the instance monitors. When the application example creates an `Agent` instance, it first sets these properties.

Agent data (for GUI purposes)	`mMediaInfo`
	`mMediaActionsPossible`
	`mVoiceMediaInfo`
	`mDnsActionsPossible`
		Information collected through the services. These attributes are arrays of agent status, and agent possible actions, for the agent's DNs and media.

Attribute Type	Attributes	Description
		The Agent class updates this information to keep it consistent.
Event management	mEventsStack mSubscriberId mNotification listen	These attributes are for event management purposes.

The Agent class uses agent properties to:

- Fill in parameters in services method calls.
  - When initializing the Agent data. See the source code of the Agent.Load() method.
  - When performing agent actions. See Managing Agent Actions on DNs and Media.
- Subscribe to and get media and voice media events with the event service
  - See Subscribing to Events.

Agent Methods

The Agent class includes methods for:

- Initializing—Load() initializes the agent data corresponding to the agent properties.
- Managing agent actions—Login(), Logout(), Ready(), and NotReady().
- Managing events:
  - RegisterEvent() to subscribe to events corresponding to the agent properties.
  - UnregisterEvent() to unsubscribe from events.
  - HandleEvent() to update with event data.
  - HasEvent/ProcessEvents() to manage the event pull mode.

Class AgentStatusForm

The AgentStatusForm class is a System.Windows.Forms.Form class that handles the form presented in The Agent Status Example at Runtime. This form uses a grid to display the DNs and media of a place associated with an agent, and provides the user with agent actions on the place, such as login, logout, ready, and not ready.
This class sets the agent properties of its mAgent attribute—an Agent instance—and then uses mAgent to monitor the agent and to access his or her agent data.
To update the GUI components of this form and implement user actions, the AgentStatusForm class contains methods that use its mAgent attribute.

AgentStatusForm Methods Using the mAgent Attribute

Methods	Description
`AgentStatusForm()` `Start()`	Initializes the form with agent data.

`UpdateStatuses()`	Displays information about the agent's status on this place, using:
	`mAgent.mMediaInfo`
	`mAgent.mVoiceMediaInfo`

`UpdateButtons()`	Enables and/or disables buttons and menu items, using:
	`mAgent.mMediaActionsPossible`
	`mAgent.mDnsActionsPossible`

`buttonLogin_click()`
`buttonLogout_click()`
`buttonReady_click()`
`buttonNotReady_click()`
	Handlers to implement agent actions on the media and DNs of the agent's place.

`menuItemPull_click()`
`SetPullMode()`
`GetEvent()`
	Handlers and methods that switch to the pull event mode and manage events.

`menuItemPush_click()`
`SetPushMode()`
`notifyEvent()`
	Handlers and methods that switch to the push event mode and manage the notified events.

Class LoginForm

The `LoginForm` class is a dialog box used to input the agent properties. The following attributes correspond to the `Agent` class properties:

```
agentID
place
queue
loginID
password
```

The `AgentStatusForm` class creates a `LoginForm` object when the application starts or when the user selects the agent menu to modify agent properties. See Setting Agent Properties.

Managing Agent Status Data

This section details the implementation of following actions in the agent status example:

- Connecting to the GIS Server.
- Setting Agent Properties.
- Updating Statuses in the Datagrid.
- Updating Buttons in the Form.
- Managing Agent Actions on DNs and Media.

Connecting to the GIS Server

To connect to the .NET Server, the `AgentStatusForm()` constructor creates an `Agent` instance and a `Connection` instance, as shown in the following code snippet:

```
/// Creating Agent and Connection objects for this status form
mAgent = new Agent();
mAgent.mConnection = new Connection();
```

The `Connection()` constructor of the `Connection` class creates a factory that instantiates the connection to the .NET Server:

```
mServiceFactory = ServiceFactory.createServiceFactory(null, null, null);
```

Because the call to ServiceFactory.`createFactory()` does not specify parameters, the factory takes into account the default values set in the `ail-configuration.xml` to connect.
Once the mServiceFactory factory is created, the `Connection()` constructor creates the services that the `Agent` instance uses.

```
mEventService = mServiceFactory.createService(typeof(IEventService), null) as IEventService;
mAgentService = mServiceFactory.createService(typeof(IAgentService), null) as IAgentService;
mPlaceService = mServiceFactory.createService(typeof(IPlaceService), null) as IPlaceService;
```

When the `AgentStatusForm.mAgent.mConnection` attribute is instantiated, `AgentStatusForm` can use its `mAgent` attribute to get data through the services.

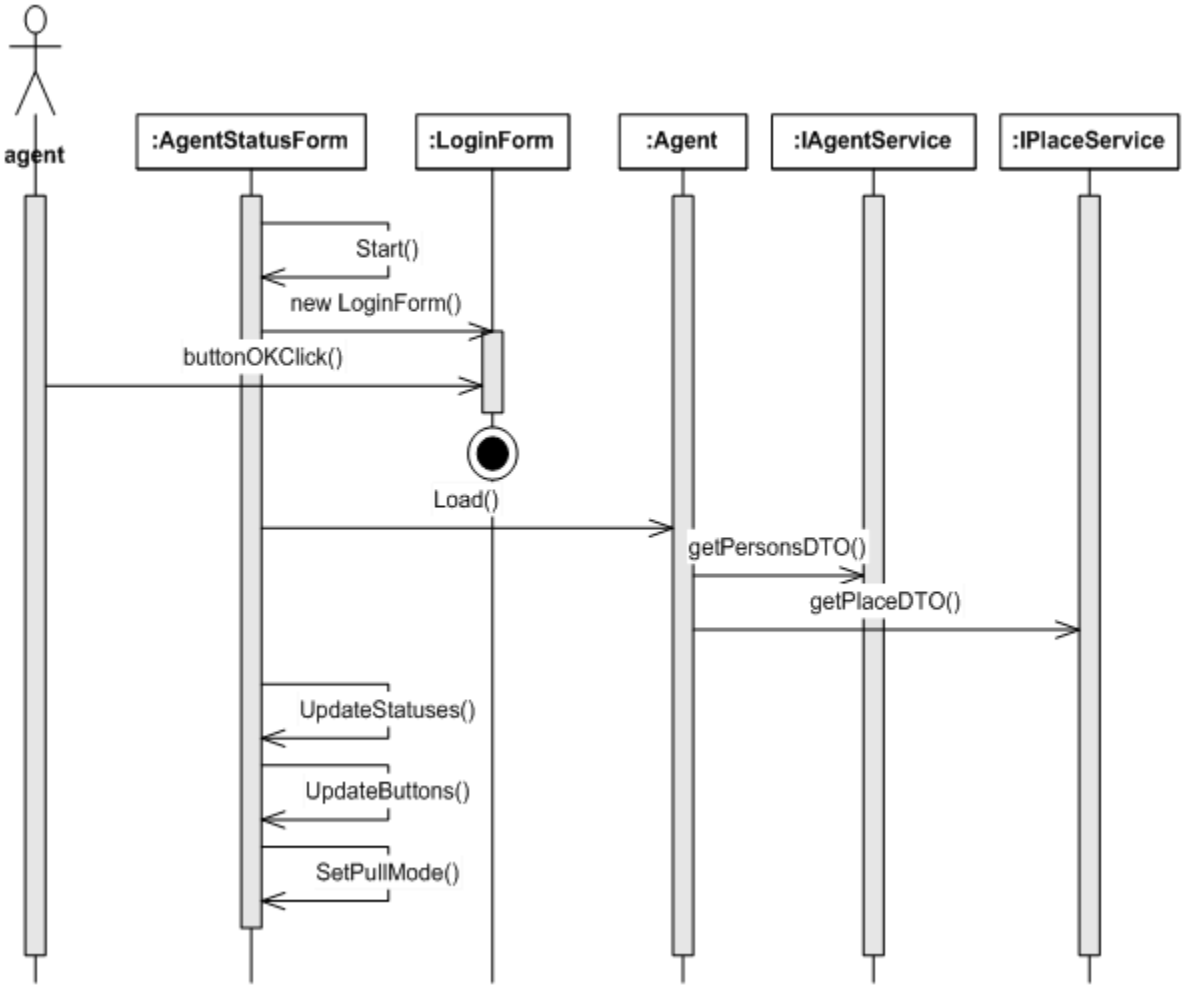
Setting Agent Properties

The `AgentStatusForm` instance needs agent properties to determine which agent's place to monitor. The user inputs these properties when:

- The application has successfully connected to the GIS at startup.
- The user clicks on the `Agent` menu to switch agents and/or places.

In both cases, the application updates to make the displayed information consistent with the entered properties.
To get these properties and then update, the `AgentStatusForm` instance calls the `AgentStatusForm`.Start() method in the constructor at startup, and again in the

`menuItemEditAgent_Click()` handler when the user clicks on the `Agent` menu.
The following diagram shows the sequence for the `AgentStatusForm.Start()` method called in the event push mode.



Getting Agent Properties and Data

The `AgentStatusForm.Start()` method follows this scenario:
Collecting the agent properties with a LoginForm dialog box.

1. Getting the required agent data through its `mAgent` instance of the `Agent` class.
2. Updating the GUI components of the form using the agent data, that is, the attribute values of the `mAgent` instance. See Updating Statuses in the Datagrid and Updating Buttons in the Form.
3. Activating an event mode to listen to events on the monitored agent. For details on event management, see Handling Events.

The following subsections discuss steps 1 and 2.

Getting Agent Properties

To get agent properties in the `AgentStatusForm.Start()` method, the `AgentStatusForm` instance creates and opens a `LoginForm` to fill in its `mAgent` properties, as shown in the following code snippet:

```
/// Getting new Agent properties
LoginForm editAgent = new LoginForm(mAgent.mAgentId,mAgent.mPlace,mAgent.mQueue,
mAgent.mAgentLogin,mAgent.mAgentPassword);

/// If the dialog box result is OK, the application assigns
// the input to Agent attributes
if(editAgent.ShowDialog() == DialogResult.OK)
{
   mAgent.mAgentId = editAgent.agentID;
   mAgent.mPlace = editAgent.place;
   mAgent.mAgentPassword = editAgent.password;
   mAgent.mQueue = editAgent.queue;
   mAgent.mAgentLogin = editAgent.loginID;
   editAgent.Dispose();

   //Updating with the (new) agent properties
   //...
}
```

For further information about the `LoginForm` dialog box, see Class LoginForm.

Getting Agent Data for New Agent Properties

To access agent data, that is, agent statuses and possible actions on media and DNs of the place, `AgentStatusForm` has to update its `mAgent` attribute. The `Start()` method calls the `Agent.Load()` method, as shown in the following code snippet.

```
mAgent.Load();
```

To take into account the (new) agent properties and (re)initialize attributes, the `Agent.Load()` method retrieves attributes for agent and place services in DTOs, as shown here.

```
// Getting the DTO for the agent.
PersonDTO [] agentDTO = mConnection.mAgentService.getPersonsDTO(new string[] { mAgentId },
new string [] { "agent:dnsActionsPossible",
"agent:mediasActionsPossible","agent:availableMedias" } );

//If the agent exists
if(agentDTO != null && agentDTO.Length == 1)
{
   PersonDTO mAgentDTO = agentDTO[0];

   // Getting the DTO for the agent's place.
   PlaceDTO[] placeDTO = mConnection.mPlaceService.getPlacesDTO( new string[] { mPlace }, new
string[] { "place:dns","place:medias" });

   if(placeDTO != null && placeDTO.Length == 1)
   {
      PlaceDTO mPlaceDTO = placeDTO[0];
      // ... Analyzing DTOs' content
   }
}
```

With agent and place DTOs, the method can fill in the following agent data:

- `mAgent.mMediaInfo`
- `mAgent.mMediaActionsPossible`
- mAgent.mVoiceMediaInfo
- mAgent.mDnsActionsPossible

Getting Information About DNs

To determine whether the agent has DNs in his or her place, the `Agent.Load()` method tests the `place:dns` attribute of the place service.
If the agent's place includes voice, this attribute value is not `null`, and associated possible actions are available in the agent:dnsActionsPossible attribute, as shown here:

```
// Getting the DNs for the agent's place.
object o = Connection.GetValue(mPlaceDTO.data, "place:dns");
if(o != null)
{
   mVoiceMediaInfo = (VoiceMediaInfo[])o;

   // Getting the possible agent actions on these DNs.
   o = Connection.GetValue(agentDTO[0].data, "agent:dnsActionsPossible");
   if(o != null)
   mDnActionsPossible = (DnActionsPossible[])o;
}
```

Getting Information About Media

If the agent is already logged in on one (or more) media of the place, the media are available in the Interaction Server. The place service can access the agent media and provides a value for the `place:medias` attribute, as shown in the following code snippet.

```
o = Connection.GetValue(mPlaceDTO.data, "place:medias");
// if the agent is logged in, media exist in the interaction server
if(o != null)
{
   mMediaInfo = (MediaInfo[])o;
   if(mMediaInfo.Length != 0)
   {
      // Getting the possible agent actions on these DNs.
      o = Connection.GetValue(mAgentDTO.data, "agent:mediasActionsPossible");
      if(o != null)
      {
         mMediaActionsPossible = (MediaActionsPossible[])o;
      }
   }
} else {
   /// Media info is not available in the place service
   //...
}
```

Media are not static in the place. If the agent is not logged in on the media of the place, the place service cannot access those in the Interaction Server, and it provides a `null` value for the

place:medias attribute.

To determine whether the agent has media for this place, test the agent:availableMedias attribute and get agent available media names, as shown in this agent status example.

If the place:medias attribute value is null, the Agent.Load() method uses the available media names to create the MediaInfo and MediaActionsPossible arrays, as shown here:

```
o = Connection.GetValue(mAgentDTO.data, "agent:availableMedias");string[] mediaNames =
((string[])o);
mMediaInfo = new MediaInfo[mediaNames.Length];
mMediaActionsPossible = new MediaActionsPossible[mediaNames.Length];

int i=0;
foreach(string mediaName in mediaNames )
{
   mMediaInfo[i] = new MediaInfo();
   mMediaInfo[i].name = mediaName;
   mMediaInfo[i].status = MediaStatus.LOGGED_OUT;
   mMediaActionsPossible[i] = new MediaActionsPossible();

   if(mediaName == "chat")
   {
      mMediaInfo[i].type = MediaType.CHAT;
      mMediaActionsPossible[i].mediaType = MediaType.CHAT;
   } else if (mediaName == "email")
   {
      mMediaInfo[i].type = MediaType.EMAIL;
      mMediaActionsPossible[i].mediaType = MediaType.EMAIL;
   }
   mMediaActionsPossible[i].agentActions = new AgentMediaAction[]{ AgentMediaAction.LOGIN };

i++;
}
```

See Place, DNs, and Media for further information about Place, DNs, and media.

Updating Statuses in the Datagrid

The AgentStatusForm instance updates the status in the datagrid when:

- mAgent gets an event, which may propagate a status change for a DN or a media. See Handling Events.
- mAgent has new agent properties. A new agent or a new place may be monitored. See Setting Agent Properties.

The AgentStatusForm.UpdateStatuses() method reads status information in the mAgent.mVoiceMediaInfo array for DNs, and in the mAgent.mMediaInfo array for media.

In these arrays, each MediaInfo or VoiceMediaInfo object corresponds to a media or voice media of the place, and contains both the identifier and its associated status.

> **Important**

> Refer to the *Agent Interaction SDK 7.6 Services API Reference* for further details about `MediaInfo` or `VoiceMediaInfo` objects.

The following code snippet shows the source code of the `AgentStatusForm.UpdateStatuses()` method.

```
if(mAgent.mVoiceMediaInfo!= null && mAgent.mVoiceMediaInfo.Length !=0)
   foreach(VoiceMediaInfo v in mAgent.mVoiceMediaInfo)
   {
      this.SetStatus(v.dnId,v.status.ToString());
   }
if(mAgent.mMediaInfo!= null && mAgent.mMediaInfo.Length != 0)
   foreach(MediaInfo m in mAgent.mMediaInfo)
   {
      this.SetStatus(m.name,m.status.ToString());
   }
```

See the `AgentStatusForm.cs` file for details about the implementation of the `AgentStatusForm.SetStatus()` method, which updates the appropriate row of the data grid with the provided name and status.

Updating Buttons in the Form

The buttons of the `AgentStatusForm` form are associated with agent actions on the place. To maintain consistency with the agent statuses on the place, the `AgentStatusForm` instance enables and/or disables the buttons when:

- `mAgent` gets an event, which may propagate a change in the possible actions on a DN or a media. See Handling Events.
- `mAgent` has new agent properties. A new agent or a new place may be monitored and possible actions my be different. See Setting Agent Properties.

The `AgentStatusForm.UpdateButtons()` method gets the possible actions in the `mAgent.mDnsActionsPossible` array for DNs, and in the `mAgent.mMediaActionsPossible` array for media.
In these arrays, each `DnsActionsPossible` or `MediaActionsPossible` object contains the list of possible actions for a DN or media of the place.

> **Important**
> Refer to the *Agent Interaction SDK 7.6 Services API Reference* for further details about `DnsActionsPossible` or `MediaActionsPossible` objects.

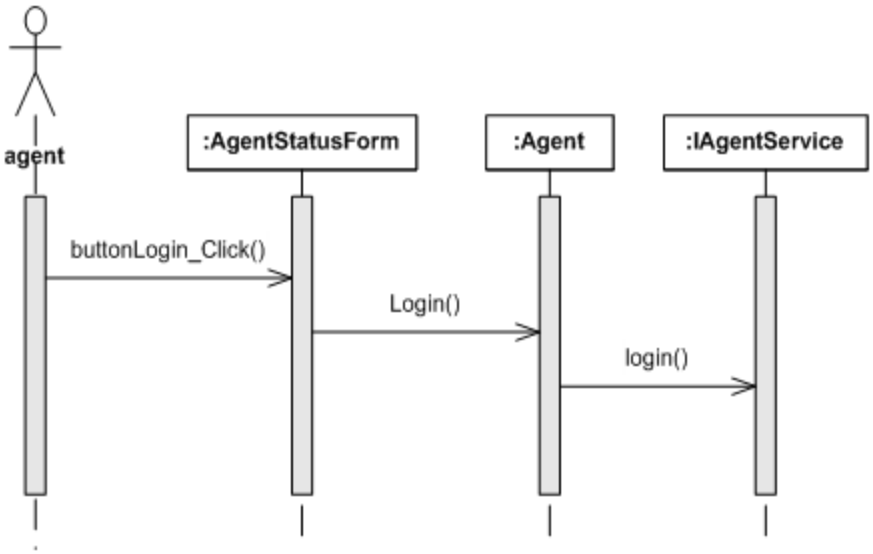
The `AgentStatusForm.UpdateButtons()` method enables a button if the corresponding action is available at least for one media or DN.
The following code snippet shows the source code of the `AgentStatusForm.UpdateButtons()` method.

```
bool login = false;
bool logout = false;
bool ready = false;
bool notReady = false;

///...
/// Testing possible actions for each media

foreach(MediaActionsPossible myMediaActions in mAgent.mMediaActionsPossible)
{
   string msg = "\t"+myMediaActions.mediaType.ToString()+ " - ";
   foreach(AgentMediaAction action in myMediaActions.agentActions)
   {
      msg+=action.ToString()+ " ";
      login = login || (action == AgentMediaAction.LOGIN);
      logout = logout || (action == AgentMediaAction.LOGOUT);
      ready = ready || (action == AgentMediaAction.READY);
      notReady = notReady || (action == AgentMediaAction.NOT_READY);
   }
   Trace(msg);
}
/// testing possible actions for each DN
///...

/// Updating buttons
this.buttonLogin.Enabled = login;
this.buttonLogout.Enabled = logout;
this.buttonNotReady.Enabled = notReady;
this.buttonReady.Enabled = ready;
Trace("Buttons updated");
```

You can change the buttons' logic to better fit your agents' needs. In this example, depending on media and DNs statuses, the application might have two contradictory buttons activated, for instance `Login` and `Logout` .

Managing Agent Actions on DNs and Media

The `AgentStatusForm` class includes four buttons corresponding to the main agent actions: login, logout, ready, and not ready. Each button click calls a handler which manages the call to the correct `Agent` method.
For example, a click on the `buttonLogin` button calls the `buttonLogin_click()` handler, as shown in the sequence below.



Performing a Login Action on the Place

The `AgentStatusForm.buttonLogin_click()` handler calls the `mAgent.Login()` method that implements the call to the agent service, as shown in the following code snippet.

```
public void Login()
{
   LoginVoiceForm myVoiceForm = new LoginVoiceForm();
   ArrayList alDn = new ArrayList();
   foreach(VoiceMediaInfo vmi in mVoiceMediaInfo)
      alDn.Add(vmi.dnId);
   myVoiceForm.dnIds = (string[])alDn.ToArray(typeof(string));
   myVoiceForm.loginId = mAgentLogin;
   myVoiceForm.password = mAgentPassword;
   myVoiceForm.queue = mQueue;
   myVoiceForm.workmode = com.genesyslab.ail.ws.agent.WorkmodeType.MANUAL_IN;
   myVoiceForm.reasons = null;
   myVoiceForm.TExtensions = null;
   MediaForm myMediaForm = new MediaForm();
   myMediaForm.reasonDescription ="Login on all media.";
   mConnection.mAgentService.login(mAgentId, mPlace, myVoiceForm, myMediaForm);
}
```

As shown in the above code snippet, the application attempts a login action on all the DNs and media of the place. For further details about forms for the agent service, see Forms and Agent Actions.
For each successful agent action on a media or a DN, your application shall receive a `MediaEvent` or a `VoiceMediaEvent` event. See Handling Events.
For further details about actions and event flow in the agent service, see Forms and Agent Actions.

Handling Events

The `AgentStatusForm` class gets events through its `mAgent` attribute. The `Agent` class monitors the media voice media events occurring on the place specified in agent properties.

For getting events, the agent status example provides two event modes: pull (the default mode) and push.
This section details how the application example handles events in the following subsections

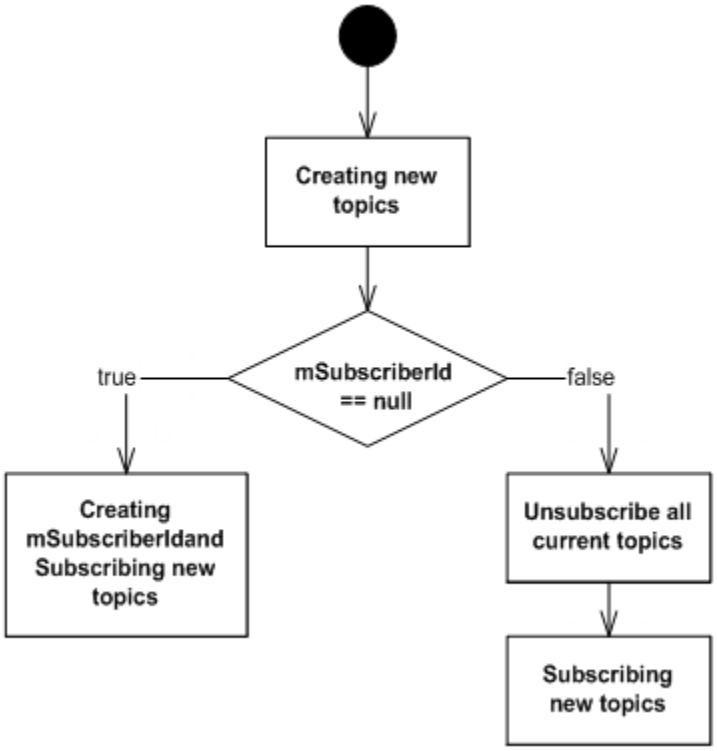
- Subscribing to Events.
- Handling the Pull Mode.
- Handling the Push Mode.
- Handling Event Changes.

Subscribing to Events

The `AgentStatusForm` class monitors media and voice-media events occurring on the agent's place. To ensure the monitoring of the correct place, the `AgentStatusForm`.start() method registers for events when agent properties change, using the `Agent.RegisterEvent()` method.
The `Agent.RegisterEvent()` method, in turn:

- Creates topic objects specifying triggers and filters on the `Agent.mPlace` ID.
- Uses a subscriber identifier—`mSubscriberId` —to subscribe to these topics with the event service.

For further information about topics, triggers, and filters, see Understanding the Event Service.
At the application's startup, the agent instance has no subscriber, so the `Agent.RegisterEvent()` method creates one, as shown below.



Scenario for the Agent.RegisterEvent() Method

> **Important**
> The `Agent` instance uses a unique `mSubscriberId` subscriber for event management.

The following subsections discuss the steps shown in the diagram above:

- Creating Topic Objects.
- Creating a Subscriber ID.
- Subscribing Topics.

Creating Topic Objects

To monitor voice-media and media events, the `Agent.RegisterEvent()` method defines topic objects for the agent service. These topic objects indicate which `Agent.mPlace` place to monitor for each event type—`VoiceMediaEvent` or `MediaEvent`—, as shown here:

```
/// Creating topic objects for the agent service
TopicsService [] topicServices = new TopicsService[1];
topicServices[0] = new TopicsService();
topicServices[0].serviceName = "AgentService";

topicServices[0].topicsEvents = new TopicsEvent[2];
topicServices[0].topicsEvents[0] = new TopicsEvent();

// Creating a topic event for voice media events
topicServices[0].topicsEvents[0].eventName = "VoiceMediaEvent";
topicServices[0].topicsEvents[0].attributes = new
String[]{"agent:voiceMediaInfo","agent:dnActionsPossible"};
topicServices[0].topicsEvents[0].triggers = new Topic[1];
topicServices[0].topicsEvents[0].triggers[0] = new Topic();
topicServices[0].topicsEvents[0].triggers[0].key = "PLACE";
topicServices[0].topicsEvents[0].triggers[0].value = mPlace;
topicServices[0].topicsEvents[0].filters = null;

// Creating a topic event for media events
topicServices[0].topicsEvents[1] = new TopicsEvent();
topicServices[0].topicsEvents[1].eventName = "MediaEvent";
topicServices[0].topicsEvents[1].attributes = new
String[]{"agent:mediaInfo","agent:mediaActionsPossible"};
topicServices[0].topicsEvents[1].triggers = new Topic[1];
topicServices[0].topicsEvents[1].triggers[0] = new Topic();
topicServices[0].topicsEvents[1].triggers[0].key = "PLACE";
topicServices[0].topicsEvents[1].triggers[0].value = mPlace;
topicServices[0].topicsEvents[1].filters = null;
```

Read also Subscribing to the Events of a Service

Creating a Subscriber ID

At application startup, the `Agent.mSubscriberId` and `Agent.mNotification` attributes are `null`. In this case, the `Agent.RegisterEvent()` method creates a subscriber for the application, as shown here.

```
SubscriberResult result =
mConnection.mEventService.createSubscriber(mNotification, topicServices);

if(result.errors == null || result.errors.Length == 0)
{
   mSubscriberId = result.subscriberId;
   mEventsStack = new ArrayList();
}
```

As shown above, the method passes the topics objects and notification at the subscriber's creation. At startup, the `mNotification` parameter is `null` and the event service sets the pull mode for the subscribed events.

Subscribing Topics

If the `Agent.mSubscriberId` attribute is not `null`, the agent instance has already subscribed once. In this case, the `Agent.RegisterEvent()` method first removes the currently-used topic objects, then subscribes with the created topic objects that take into account the new agent properties (see Creating Topic Objects).

```
/// Removing previous topics
this.mConnection.mEventService.unsubscribeAllTopics(this.mSubscriberId);

/// Subscribing for new topics
TopicServiceError[] topicsError = this.mConnection.mEventService.subscribeTopics(
this.mSubscriberId,topicServices);
```

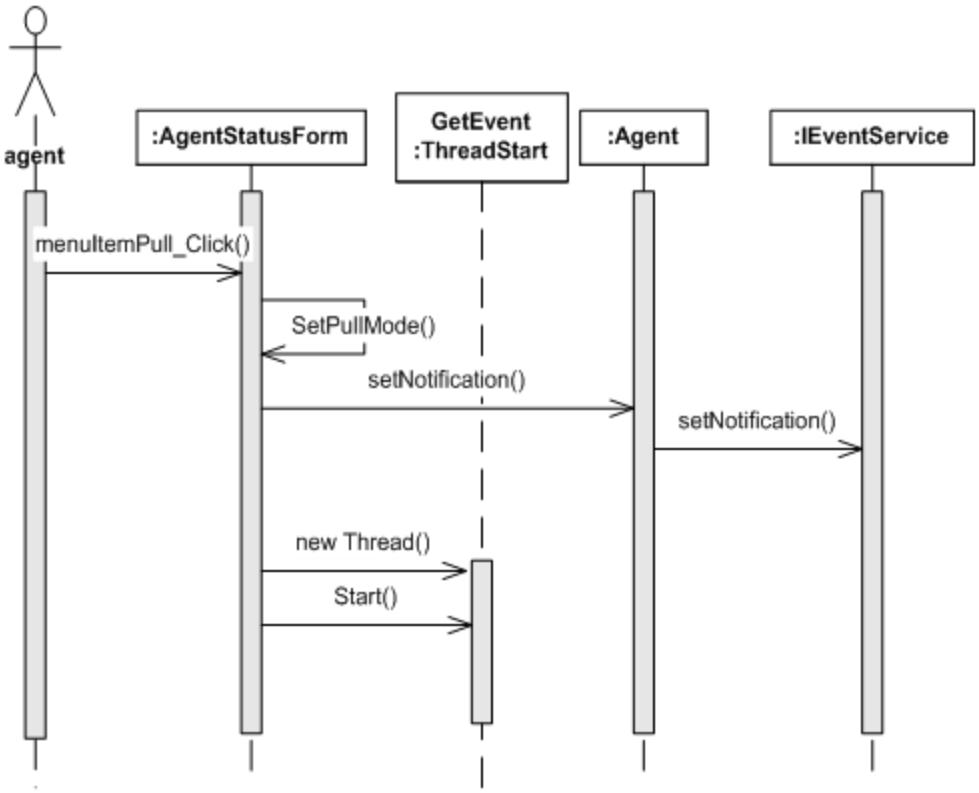
Handling the Pull Mode

This section describes two main actions for handling the pull mode in the following subsections:

- Setting the Pull Mode.
- Pulling Events.

Setting the Pull Mode

To switch to the pull mode, the `AgentStatusForm.SetPullMode()` method first calls the `Agent.SetNotification()` method, then creates a thread that will listen to the subscribed events, as presented in Setting the Pull Mode.



Setting the Pull Mode

Setting a null Notification

To set the pull mode active, the `mAgent.SetNotification()` method sets the `mAgent.mNotification` attribute to `null` and changes notification with the `IEventService.SetNotification()` method, as shown in the following code snippet:

```
public void setNotification(AgentStatusForm notifEndPoint)
{
   if(notifEndPoint!=null)
   {
      listen = false;
      ///... For push mode
   }
   else
   {
      mNotification=null;
      listen = true;
   }
   mConnection.mEventService.setNotification(this.mSubscriberId, mNotification);
}
```

> **Important**
> The `mAgent.listen` boolean indicates whether the current event mode is pull.

Creating a Thread

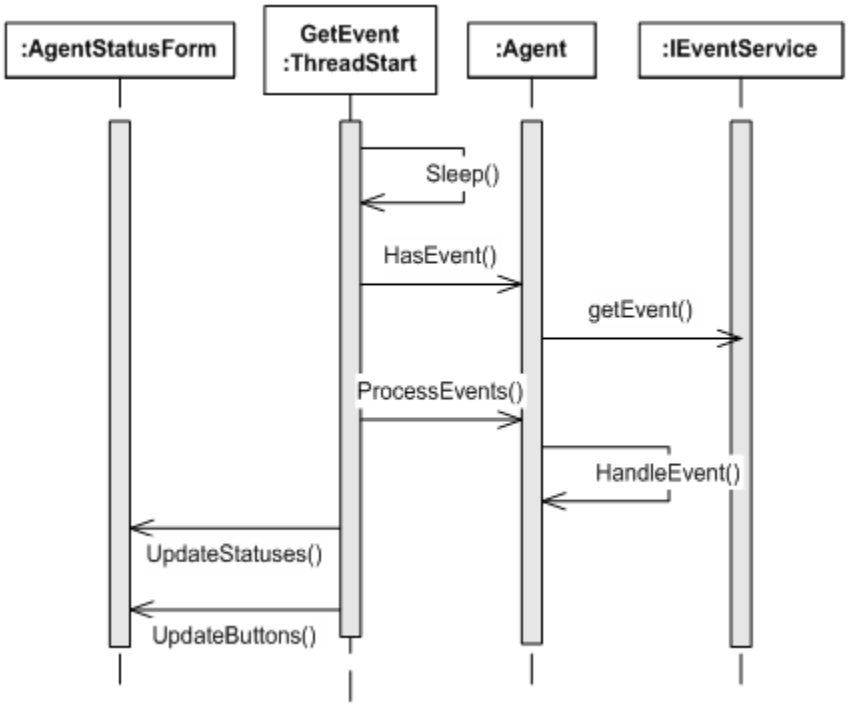
Once the pull mode is active, the `AgentStatusForm.SetPullMode()` method creates a `AgentStatusForm.thEvents` thread that listens for events, as shown in the following code snippet:

```
public void SetPullMode()
{
   //...
   thEvents = new Thread(new ThreadStart(this.GetEvents));
   thEvents.Name ="GetEvents";
   thEvents.Start();
   //..
}
```

For further details about this thread, see Pulling Events, immediately below.

Pulling Events

The `AgentStatusForm.thEvents` thread executes the `AgentStatusForm.GetEvent()` method. It makes periodic calls to the `mAgent.HasEvent()` method that pulls events (if any), as shown in the following sequence.



Periodic Pulling of the GetEvent Thread

The `AgentStatusForm.GetEvent()`thread tests the result that the `mAgent.HasEvent()` method returns. If the result is `true`, the thread calls the `mAgent.ProcessEvents()` method, as shown in the following code snippet.

```
/// Class AgentStatusForm
public void GetEvents()
{
   if(mAgent != null)
   {
      while(mAgent.listen)
      {
         //Pulling events
         if(mAgent.HasEvent())
         {
            //Updating mAgent with pulled events
            mAgent.ProcessEvents();
            // Updating GUI
            UpdateStatuses();
            UpdateButtons();
         }
         else
            Thread.Sleep(1000);
      }
   }
}
```

The call to `mAgent.ProcessEvents()` updates the `mAgent` instance with the data changes propagated in events. Then, the thread updates `AgentStatusForm` with `mAgent` data. See Updating Statuses in the Datagrid and Updating Buttons in the Form.

Agent.HasEvent()

To pull events, the `Agent.HasEvent()` method makes a call to the IEventService.getEvents() method and adds them to the `mAgent.mEventsStack` event stack, as shown in the following code snippet.

```
public bool HasEvent()
{
   try
   {
      com.genesyslab.ail.ws._event.Event [] eventResult
            = mConnection.mEventService.getEvents(mSubscriberId, 0);
      if(eventResult != null && eventResult.Length > 0)
      {
         mEventsStack.AddRange(eventResult);
         return true;
      }
      else return false;
   }
   catch(Exception e)
   {
      return false;
   }
}
```

Agent.ProcessEvents()

The `Agent.ProcessEvents()` method parses the event stack. For each event, it makes a call to the `mAgent.handleEvent()` that updates the `mAgent` instance with the event content.

```
public void ProcessEvents()
{
   /// Managing events if any.
   if(mEventsStack.Count > 0)
   {
      ArrayList eventsStack = new ArrayList(mEventsStack);
      foreach(com.genesyslab.ail.ws._event.Event e in eventsStack)
      {
         /// Removing an event from the stack
         mEventsStack.Remove(e);
         /// Managing the event
         HandleEvent(e);
      }
   }
}
```

For further details about `mAgent.HandleEvent()`, see Handling Event Changes.

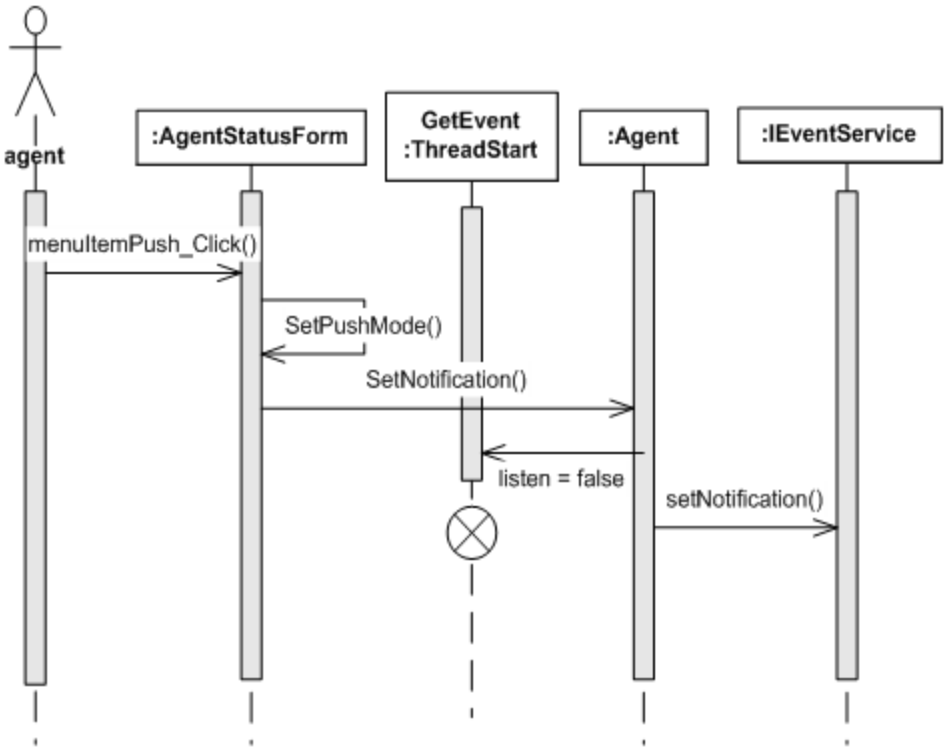
Handling the Push Mode

This section describes the two main actions for handling the push mode in the following subsections:

- Setting the Push Mode.
- Event Notification.

Setting the Push Mode

To switch to the push mode, the `AgentStatusForm.SetPushMode()` method makes a single call to the `Agent.SetNotification()` method, as presented here.



Setting the Push Mode

`AgentStatusForm` inherits `INotifyService` and implements the `notifyEvents()` method that the event service will call in case of events. See Event Notification.
The `AgentStatusForm.SetPushMode()` method passes this to the `Agent.SetNotification()` method, as shown here:

```
private void SetPushMode()
{
   this.mAgent.setNotification(this);
   this.menuItemPull.Checked = false;
   this.menuItemPush.Checked = true;
   Trace("Push mode activated");
}
```

Creating a Notification Instance

To properly set the push mode, the `mAgent.SetNotification()` method first sets the `mAgent.listen` boolean to `false` to stop the pulling thread (see Pulling Events).
Then, the `mAgent.SetNotification()` method creates a `Notification` object with the `AgentStatusForm` instance for notification end point, as shown here.

```
public void setNotification(AgentStatusForm notifEndPoint)
{
   if(notifEndPoint!=null)
```

```
   {
      listen = false;
      mNotification = new Notification();
      mNotification.notificationEndpoint = notifEndPoint;

      if(this.mConnection.mServiceFactory.ServiceFactoryImpl is
com.genesyslab.ail.WebServicesFactory)
         mNotification.notificationType="SOAP_HTTP";
      else
         mNotification.notificationType="JAVA";
   }
   else
   {
      /// For pull mode
      ///...
   }

   mConnection.mEventService.setNotification(this.mSubscriberId, mNotification);
}
```

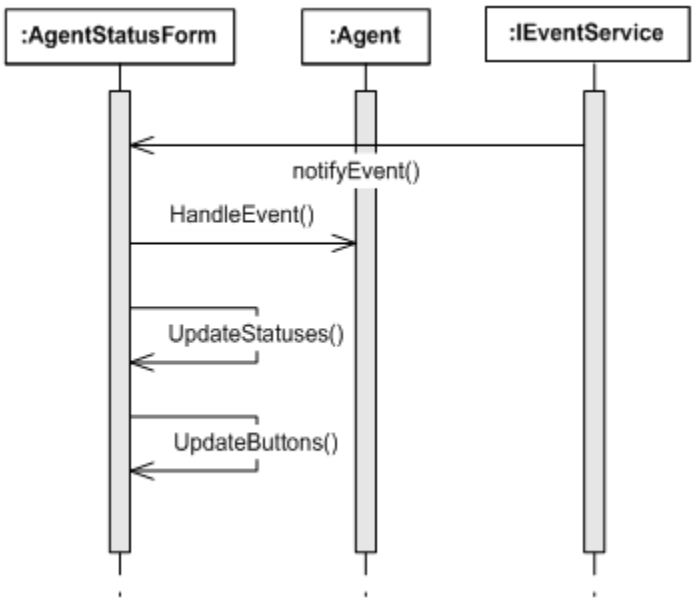
Setting the Notification Type

When setting the notification type for the created `Notification` object, the method tests the `mConnection.ServiceFactory` object to determine which protocol the application uses. It is `SOAP_HTTP` for SOAP with the Genesys Interface Server. For further details, see About the Examples.

After the call to the`IEventService.setNotification()` method, the event service uses the `AgentStatusForm` for event notification. See Event Notification, immediately below.

Event Notification

In the push mode, each time an event occurs, the `AgentStatusForm.notifyEvents()` method is called.



Notification of an Event

For each event notified, the `AgentStatusForm.notifyEvents()` method updates `mAgent` data by calling the `Agent.HandleEvent()` method, then it updates the GUI, as shown in the following code snippet.

```
public void notifyEvents(string subscriberId,
com.genesyslab.ail.ws._event.Event[] events)
{
   if (events == null)
   {
      Trace("NotifyEvents - null");
      return ;
   }
   Trace( "NotifyEvents getEvents : " + events.Length) ;
   foreach( Event evt in events)
   {
      // Updating mAgent content
      mAgent.HandleEvent(evt);
      Trace( "Service :"+ evt.serviceName
         + "Event: "+ evt.eventName
         + "timeStamp:"+ evt.timeStamp);
      //Updating the GUI
      UpdateButtons();
      UpdateStatuses();
   }
}
```

For further details about `Agent.HandleEvent()`, see Handling Event Changes.

Handling Event Changes

Regardless of the event mode—push or pull—the application gets arrays of `com.genesyslab.ail.ws._event.Event` objects. Each `Event` contains an `MediaEvent` or a `VoiceMediaEvent` and published values for the service attributes.
The `Agent.HandleEvent()` method updates `mAgent` agent data with the published values for the agent service attributes and retrieves attribute values by calling the `Connection.GetValue()` method:

```
public void HandleEvent(com.genesyslab.ail.ws._event.Event e)
{
   /// Managing this event
   switch(e.eventName)
   {
      case "VoiceMediaEvent":
      {
         /// Updating the voice media info.
         object o = Connection.GetValue(e.attributes, "agent:voiceMediaInfo");
         if(o != null)
            mVoiceMediaInfo[0] = (VoiceMediaInfo)o;

         /// Updating the possible agent actions on DNs
         o = Connection.GetValue(e.attributes, "agent:dnActionsPossible");
         if(o != null)
         {
            if(mDnActionsPossible == null)
               mDnActionsPossible = new DnActionsPossible[mVoiceMediaInfo.Length];
            mDnActionsPossible[0] = (DnActionsPossible)o;
```

```
            }
        }
        break;
        case "MediaEvent":
        // getting values for mMediaInfo and mMediaActionsPossible
        //...
    }
    break;
}
```

When the `Agent` object is updated, the `AgentStatusForm` instance can use the `Agent` object to update the data grid and the buttons.