Agent Interaction SDK Services Developer Guide

The Agent Status Example

12/12/2025

Contents

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.

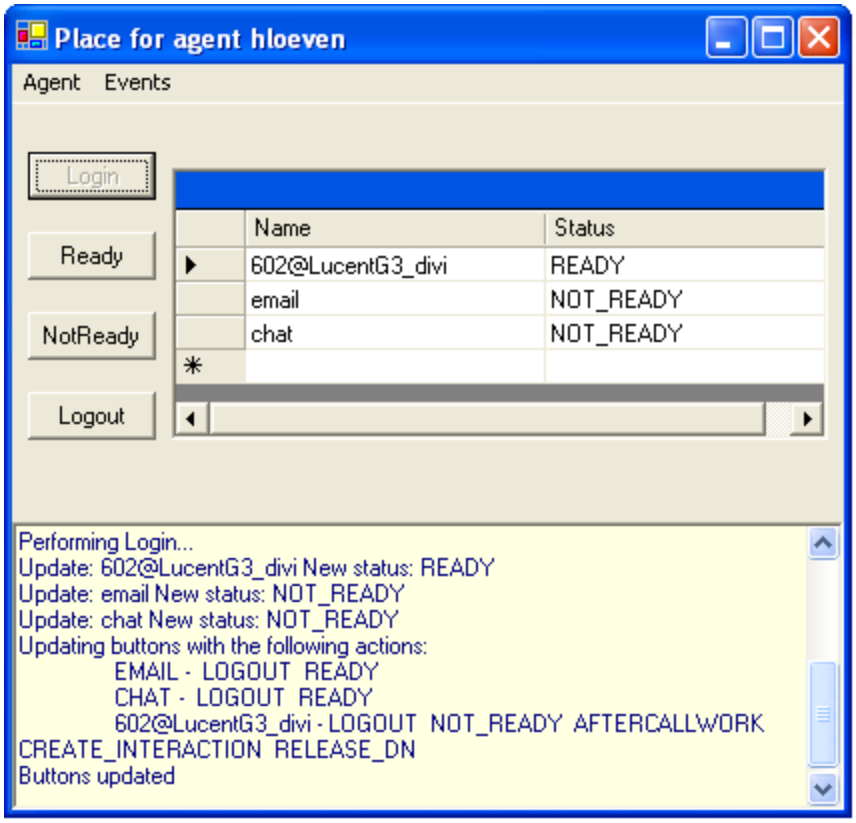
Agent Status Short Description

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.

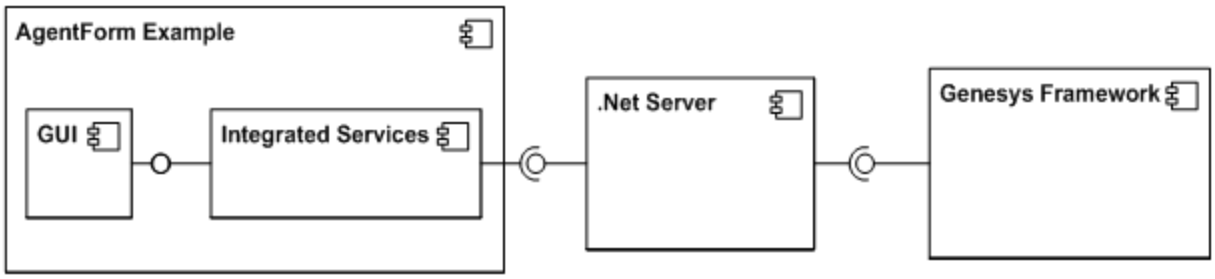


The Agent Status Example at Runtime

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

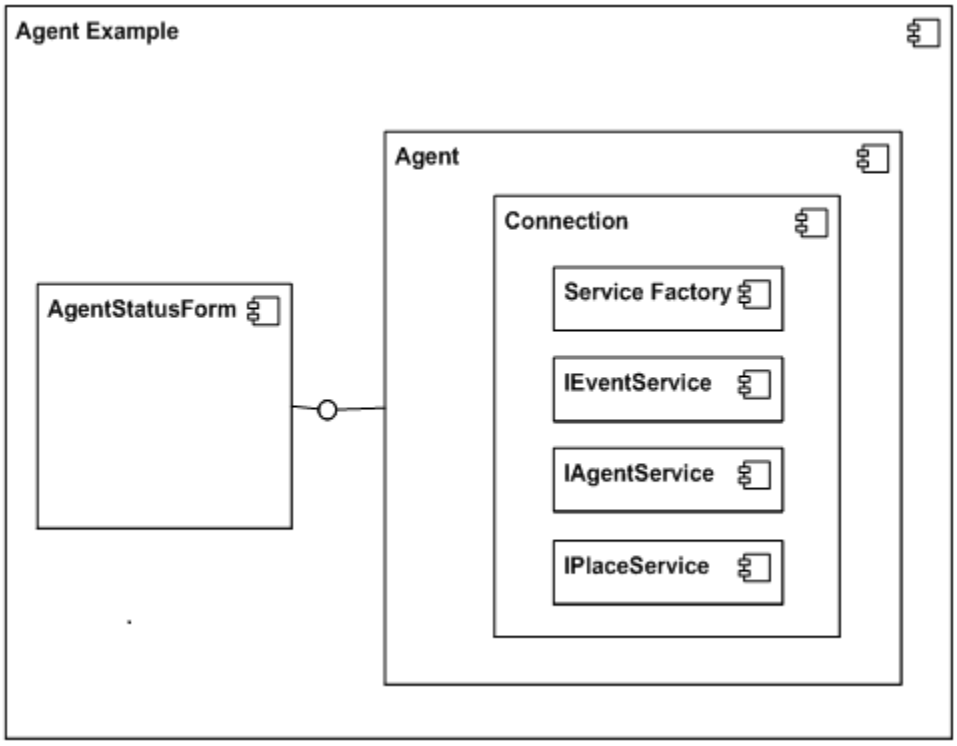
the `Working Directory` option. This ensures that the running application takes into account the correct `ail-configuration.xml` file.

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.

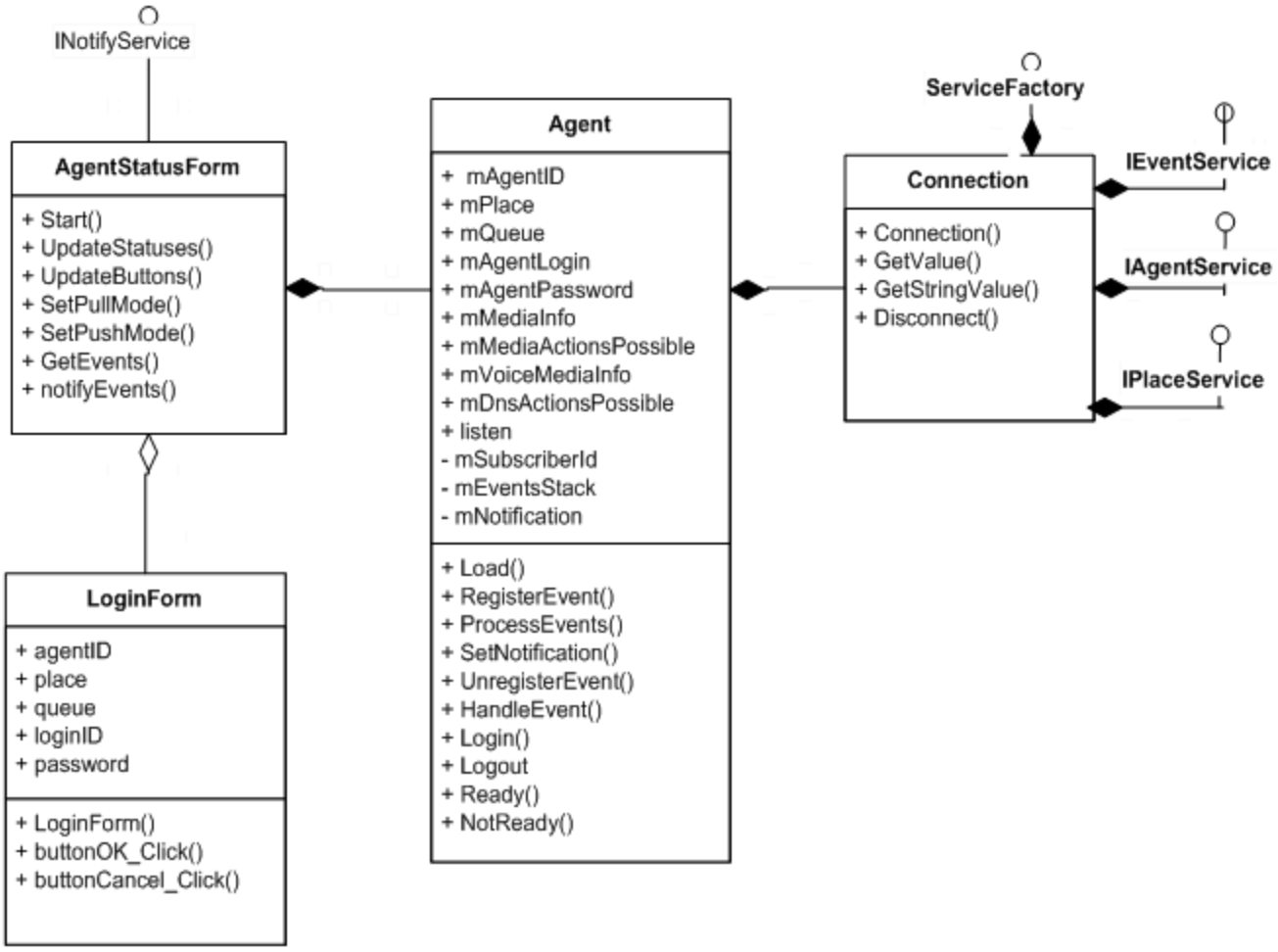
- The `Agent` class uses the services created with a `Connection` instance to:
  - Retrieve agent and place information for a particular agent.
  - Manage DNs and media events (including subscription) on this agent.
  - Perform agent actions on this agent's place.

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.

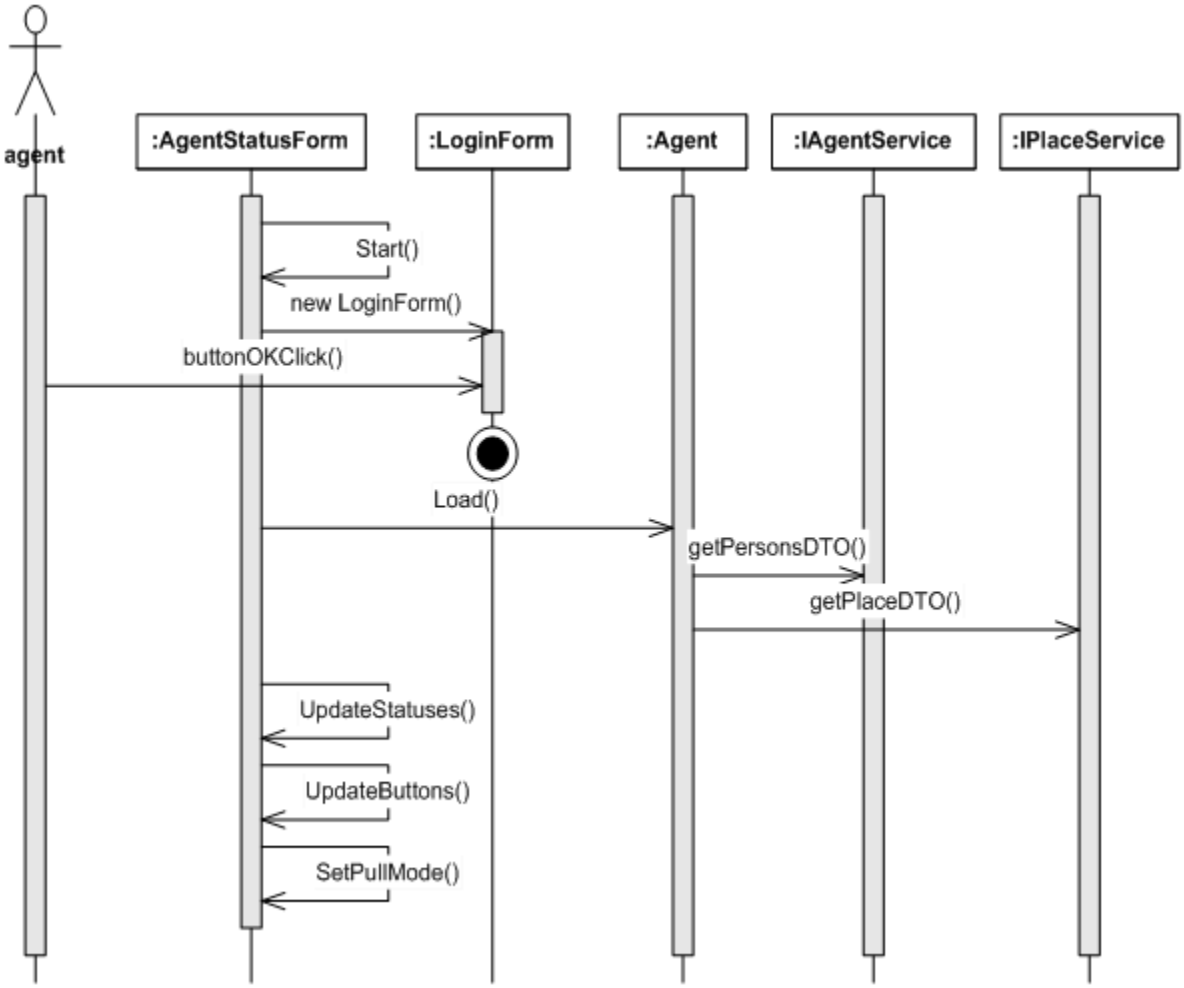
Setting Agent Properties

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.

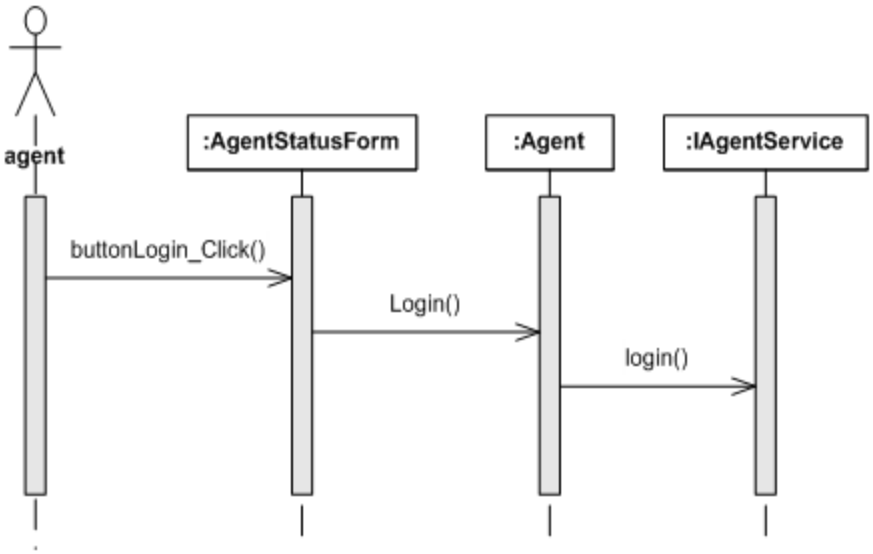
The `AgentStatusForm.UpdateButtons()` method enables a button if the corresponding action is available at least for one media or DN.
The following code snippet shows the source code of the `AgentStatusForm.UpdateButtons()` method.

```
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

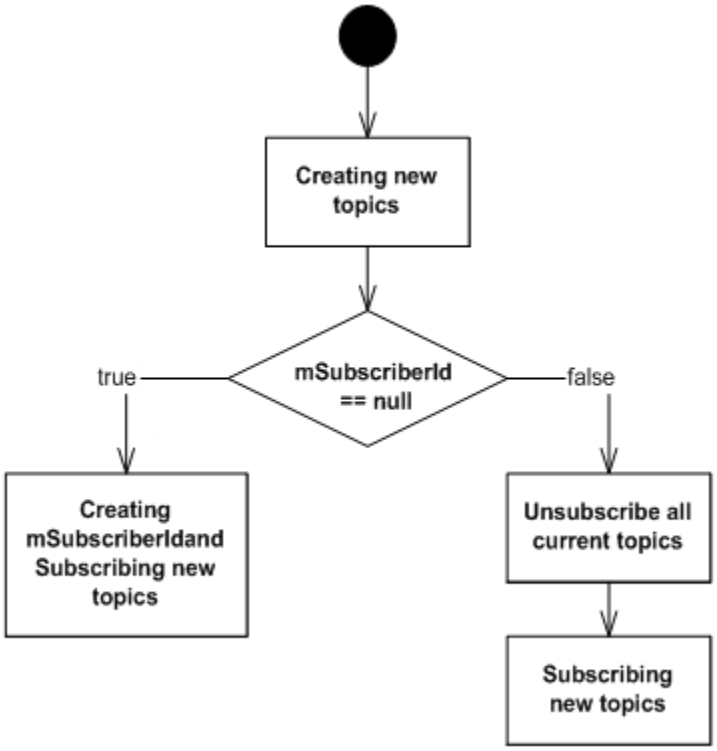
- Subscribing to Events.
- Handling the Pull Mode.
- Handling the Push Mode.
- Handling Event Changes.

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);
```

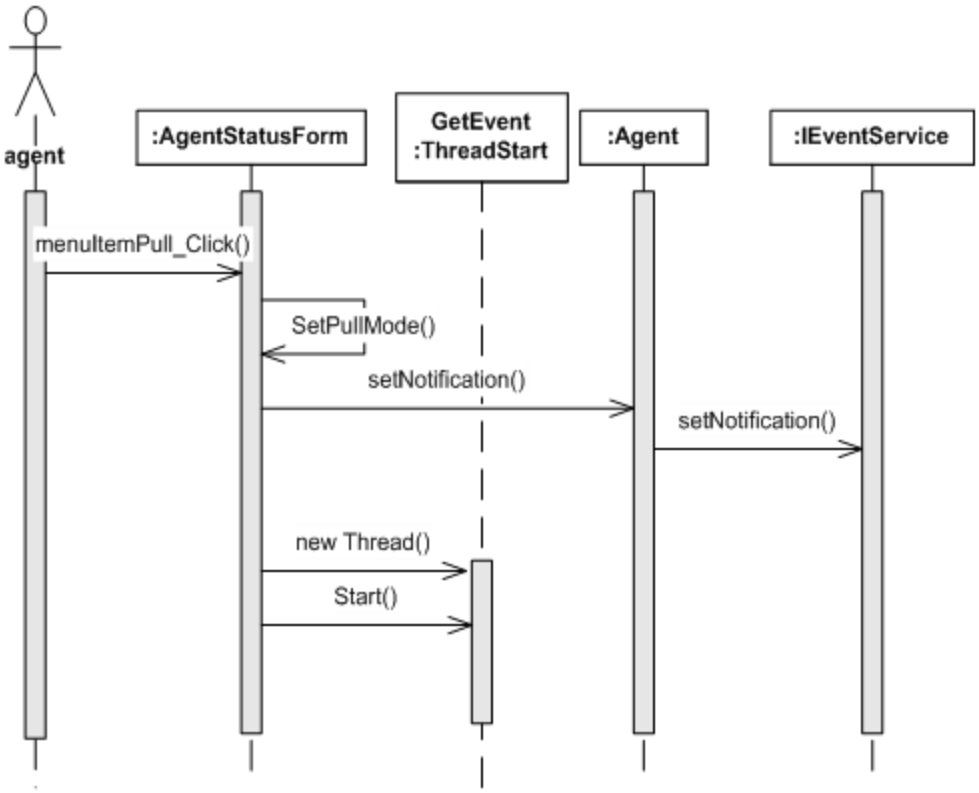
Handling the Pull Mode

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

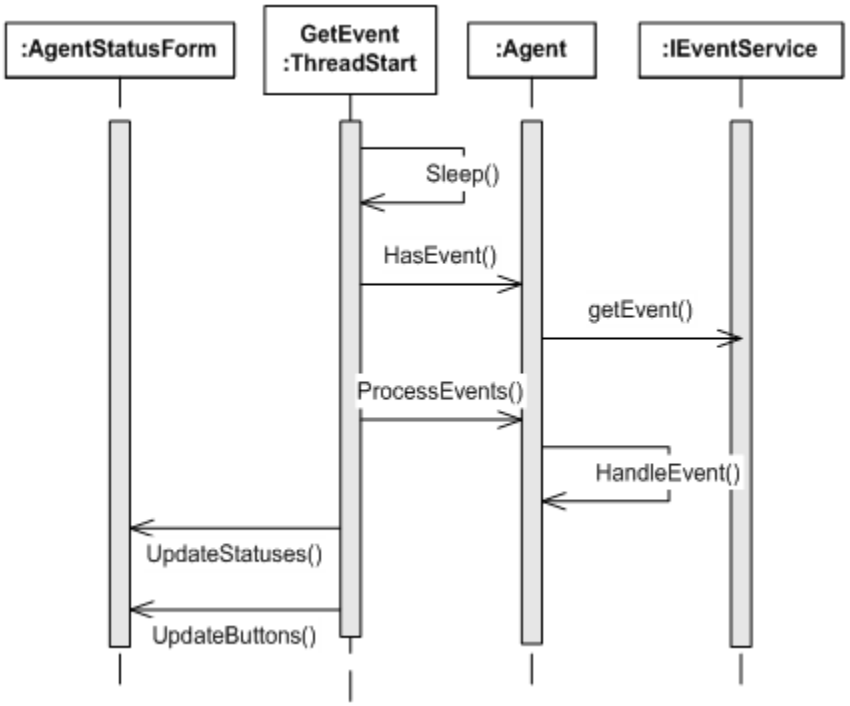
Once the pull mode is active, the `AgentStatusForm.SetPullMode()` method creates a `AgentStatusForm.thEvents` thread that listens for events, as shown in the following code snippet:

```
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.

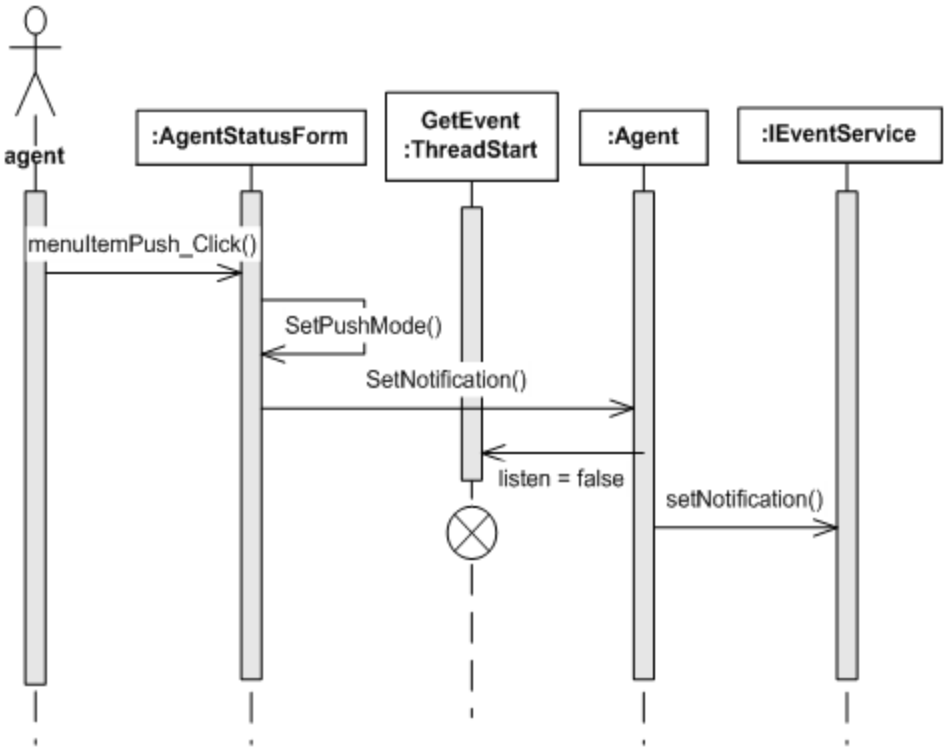
Handling the Push Mode

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);
}
```

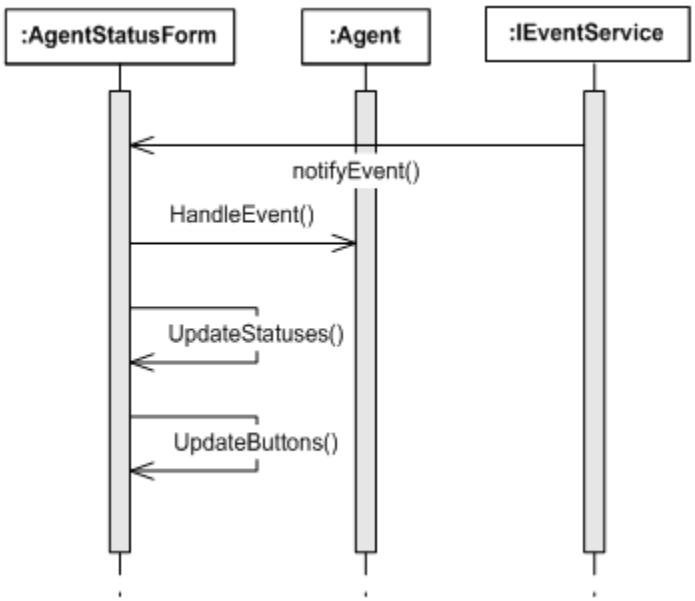
Setting the Notification Type

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.