Workspace Desktop Edition Developer's Guide

Advanced Customization

5/1/2025

Advanced Customization

Purpose: To provide information about the advanced customization with the Enterprise Service API.
Available since: 8.1.100.14; **updated in:** 8.1.300.17

Contents

Get the Enterprise Service API Reference

The Enterprise Services are core components used by the modules, views, and resources to connect to Genesys Servers and maintain the information flow consistent with the state of Interaction Workspace. The main entry point is available through the `EntrepriseService` property of the `Genesyslab.Desktop.Modules.Core.Model.Agents.IAgent` interface. This interface enables you to access all of the available Enterprise services. All of these services handle the core objects that Interaction workspace creates and displays. Modifications to these objects through Interaction Workspace API should be fine; however, if you create new instances or alter objects through the Enterprise API, your customization is responsible for maintaining the information flow consistent with the data displayed in Interaction Workspace. The following table contains the list of downloadable API References available:

Interaction Workspace Release Version	Enterprise Service Release Version	Release Date	Enterprise Service CHM
8.1.300.17	8.1.300.14	07/31/2012	Download chm
8.1.200.16	8.1.200.10	01/31/2012	Download chm
8.1.100.14	8.1.100.10	07/29/2011	Download chm

If you encounter difficulties with opening the .chm files, please check the known issues and solutions of Dr. Explain.

Get EnterpriseService

The main entry point is available through the `EntrepriseService` property of the `Genesyslab.Desktop.Modules.Core.Model.Agents.IAgent` interface. The `Resolve` methods of the `IEnterpriseServiceProvider` simplify the retrieval of a service instance.

[C#]

```
public MyNewSampleClass(IUnityContainer container, ILogger log)
{
 IAgent myAgent= container.Resolve<IAgent>();
 IEnterpriseServiceProvider enterpriseService = myAgent.EntrepriseService;
 //...
 INameService nameService = enterpriseService.Resolve<INameService>("key");
}
```

- Where `Name` is the service name, and key is the mapping key that is predefined in the native source of the Enterprise API.

Service Name	Service Key	Associated Protocols<ref>Protocols can be used when you are managing channels.</ref>
IAgentService	agentService	• AgentProtocolRequest-"agent" • DeviceProtocolRequest-"device"
IChannelService	channelService	*none*
IDeviceService	deviceService	• DeviceProtocolRequest-"device"
IIdentityService	identityService	• OpenMediaProtocolRequest-"openmedia"
IIMService	IMService	• VoiceProtocolRequest-"voice" • DeviceProtocolRequest-"device" • IMProtocolRequest-"im"
IContactService	contactService	• ContactProtocolRequest --"contacts"
IInteractionService	interactionService	*none*
IChatService	chatService	• OpenMediaProtocolRequest-"openmedia" • WebMediaProtocolRequest -"webmedia"
IOpenMediaService	openmediaService	*none*
IMonitorService	monitorService	• OpenMediaProtocolRequest-"openmedia"
IWorkbinService	workbinService	none • OpenMediaProtocolRequest-"openmedia"
IPSTService	PSTService	none
ICampaignService	campaignService	none
IOutboundService	outboundService	none

<references />

Additional Entry Points

Interaction Workspace API provides additional entry points through properties in the specific classes that are listed in the table below:

Class Name	Property	Description
Genesyslab.Desktop.Modules.OpenMedia.Model.Agents.IAgentMultimedia	EntrepriseAgent	IAgent instance which contains the agent data.
Genesyslab.Desktop.Modules.Core.Model.Interactions.IInteraction	EntrepriseInteractionCurrent	Current interaction processed by Interaction Workspace.
IList<Genesyslab.Enterprise.Model.Interactions.IInteraction>	EntrepriseInteractions	The history of interactions.
Genesyslab.Platform.Commons.Protocols.IMessage	EntrepriseInteractionEvent	The last interaction event.
Genesyslab.Desktop.Modules.OpenMedia.Model.Interactions.Chat.IInteractionChatCommon	EntrepriseChatInteractionCurrent	Current chat interaction processed by Interaction Workspace.
Genesyslab.Desktop.Modules.OpenMedia.Model.Interactions.Email.IInteractionEmail	EntrepriseEmailAttachments	E-mail attachments.
Genesyslab.Desktop.Modules.OpenMedia.Model.Interactions.Email.IInteractionEmail	EntrepriseEmailInteractionCurrent	Current e-mail interaction processed by Interaction Workspace.
Genesyslab.Desktop.Modules.OpenMedia.Model.Interactions.IInteractionOpenMedia	EntrepriseOpenMediaInteractionCurrent	Current open media interaction processed by Interaction Workspace.
Genesyslab.Desktop.Modules.OpenMedia.Model.Interactions.Sms.IInteractionSms	EntrepriseSmsInteractionCurrent	Current sms interaction in page mode processed by Interaction Workspace.
	EntrepriseSmsSessionInteractionCurrent	Current sms interaction in session mode processed by Interaction Workspace.

Enterprise Extensions

The `Genesyslab.Enterprise.Extensions` namespace defines a list of extensions classes which provide the switch-specific action areas of each related service.

Service	Extension	Related features
IIdentityService	AgentServiceExtensions	Manage login, Ready, Not Ready
IDeviceService	DeviceServiceExtensions	Manage the call-forward and Do Not Disturb features.
IIMService	IMServiceExtensions	Manage the messages and

Service	Extension	Related features
		transcripts of instant messaging sessions.
IInteractionService	InteractionServiceExtensions	Manage the requests on interactions (Make the call, answer the call, transfer the call, and so on.)
IMonitorService	PAMExtensions	Manage subscriptions and statistic notifications.

Add the Genesyslab.Enterprise.Extensions namespace to your code to enable the extension methods of your service.

Managing Connections and Channels

Interaction Workspace manages the connections defined in the application configuration. You can access them through the `Genesyslab.Desktop.Modules.Core.SDK.Protocol.IChannelManager`. You can retrieve the connection by passing the configured application name at the registration of the channel, as shown below:

```
IChannelManager channelManager = container.Resolve<IChannelManager>();
Genesyslab.Enterprise.Model.Channel.IClientChannel tserverChannel =
channelManager.Register(("YourApplicationName","MyClientName");
```

Four application types are supported:

- TServer
- StatServer
- InteractionServer
- UCSServer

Through the `IChannelManager` interface, you can open channels for applications of these types without burdening Interaction Workspace. However, if you wish to open new channels for other application types, you can use the `IChannelService` of the Enterprise API. Genesys recommends that you name those channels according to their configuration's application name.

Connect your Channel

1. Retrieve the channel service

```
IChannelService channelService = EnterpriseService.Resolve<IChannelService>("channelService");
```

2. Create a new channel for each connection to open.

```
string channelName = "configName";
TServerConfiguration configuration = new TServerConfiguration(channelName);
configuration.ClientName = channelName;
configuration.Uri = new Uri("tcp://hostname:port");
configuration.WarmStandbyAttempts = 10;
```

```
configuration.WarmStandbyTimeout = 5;
configuration.WarmStandbyUri = new Uri("tcp://hostname:port");
configuration.UseAddp = false;
channelService.CreateChannel(channelName, configuration, SwitchModelType.LucentDefinityG3);
```

3. Register the channel's event handler before you open the connection, to ensure that your application does not miss any events. The following code snippet shows also how to retrieve the channel instance created.

```
Genesyslab.Enterprise.Model.Channel.IClientChannel channel =
channelService.GetChannel(channelName);
//Register for Channel events
channelService.RegisterEvents(channel, new
Action<Genesyslab.Enterprise.Model.Channel.IClientChannel>(ChannelEvent));
```

4. To make the connection to all of the channels, call the `IChannelService.Connect()` method..

```
channelService.Connect();
```

The code snippet uses the `channelName` string as a label to identify your connection. Your application will use this label later to access this channel.

Get the Protocol

The table in Get EnterpriseService provides the key for the protocols that associated with channels. You can retrieve the protocols once they are connected, as shown in the following code snippet.

```
IEnterpriseProtocol media = voiceChannel.EnterpriseProtocols["voice"];
```

Getting Additional Service Events

In the Enterprise API, all services that allow event subscription include the following pair of self-describing methods: `RegisterEvents` and `UnRegisterEvents`. For instance, the following code snippet shows the registration of a `DeviceEvent` handler for the device service:

```
IDeviceService deviceService = EsdkService.Resolve<IDeviceService>("deviceService");
IDevice device = deviceService.CreateDevice("myDevice", DeviceType.Extension);
deviceService.RegisterEvents(device, new Action<IEnvelope<IDN>>(DeviceEvent));
```

To read the envelope content take advantage of the fact that the type of object published is specified in the handler declaration (which must match the registration requirements).

```
protected void DeviceEvent(IEnvelope<IDN> tsp)
{
 if (tsp != null)
 {
 //Retrieve the published object
 IDevice device = (IDevice) tsp.Body;
 System.Console.WriteLine("Name : " + device.Name + " Status: " + device.State.ToString());
 switch (tsp.Header.CurrentContext.ContextState)
 {
```

```
 case ContextStateType.Error:
 //...
 break;
 //...
 }
 }
}
```

Threading Recommendations

When you write your handler code, you should process the event's Envelope in a separate thread that can take appropriate actions. Design your handlers to return as quickly as possible, because the library core works with all handlers sequentially-waiting for each handler to return, before working with the next handler. This recommendation is extremely important to ensure that:

- Your application remains synchronized with up-coming events.
- Your application remains synchronized with the real-time time line of external devices.

Attributes and Filters

You can define callback and filter attributes when declaring your event handlers.

- A callback attribute is used to hard-code the automatic registration of the handler method for a given channel.
- A filter attribute is used to hard-code the filtering of events that your application receives.

Attribute name	Type	Dependency
EnterpriseAgentEvent	Callback, Filter	Channel name
EnterpriseChannelEvent	Callback	Channel Name
EnterpriseDeviceEvent	Callback	Channel Name
EnterpriseFilter	Filter	Object parameters
EnterpriseInteractionEvent	Callback	Channel Name
EnterpriseMonitorEvent	Callback	Channel Name
EnterpriseStrategy	Filter	Strategy instance
EnterpriseService	Filter	 See below

Callback Attribute Syntax

If you use a callback attribute, callback registration is automatic. The following code snippet shows how to use method attributes by defining an interaction event handler for a SIP channel. The first part of the snippet shows the creation of the *TServerSIPChannel* channel. The second part shows the attribute's declaration.

```
//Channel Definition
IChannelService channelService = EsdkService.Resolve<IChannelService>("channelService");
TServerConfiguration myConfiguration = new TServerConfiguration("TServerSIPChannel");
channelService.CreateChannel("TServerSIPChannel", myConfiguration, mySwitchType);
//...
```

```
[EnterpriseInteractionEvent("TServerSIPChannel")]
protected void InteractionEvent(IEnvelope<IInteraction> tsp)
{
 //...
}
```

Filter Attribute Syntax

If you are using filter attributes, the callback registration is not automatic; therefore, you must implement it.

```
//Example of Filters:
//Callback active for the Agent 1001 when status is ready
[EnterpriseFilter("1001", "ready")]
protected void AgentEvent(IEnvelope<IAgent> tsp)
{
 //...
}
//Uses the AgentCallBackFilterStrategy strategy for calling this handler (or not)
[EnterpriseStrategy("genericFilter", typeof(AgentCallBackFilterStrategy))]
protected void AgentEvent(IEnvelope<IAgent> tsp)
{
 //...
}
//Callback active when ready status events.
[EnterpriseAgentEvent("ready")]
protected void AgentEvent(IEnvelope<IAgent> tsp)
{
 //...
}
```