Agent Interaction SDK Services Developer Guide

The Event Service

4/5/2026

Contents

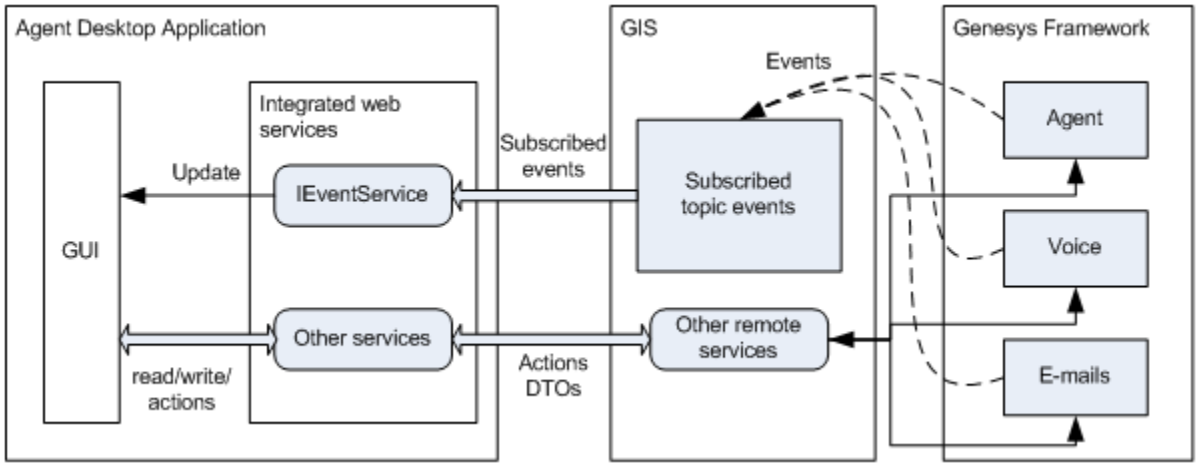
- 1 The Event Service
  - 1.1 Event Service Overview
  - 1.2 Understanding the Event Service
  - 1.3 Handling Topics Objects
  - 1.4 Getting Events
  - 1.5 Event Notification in Java

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.

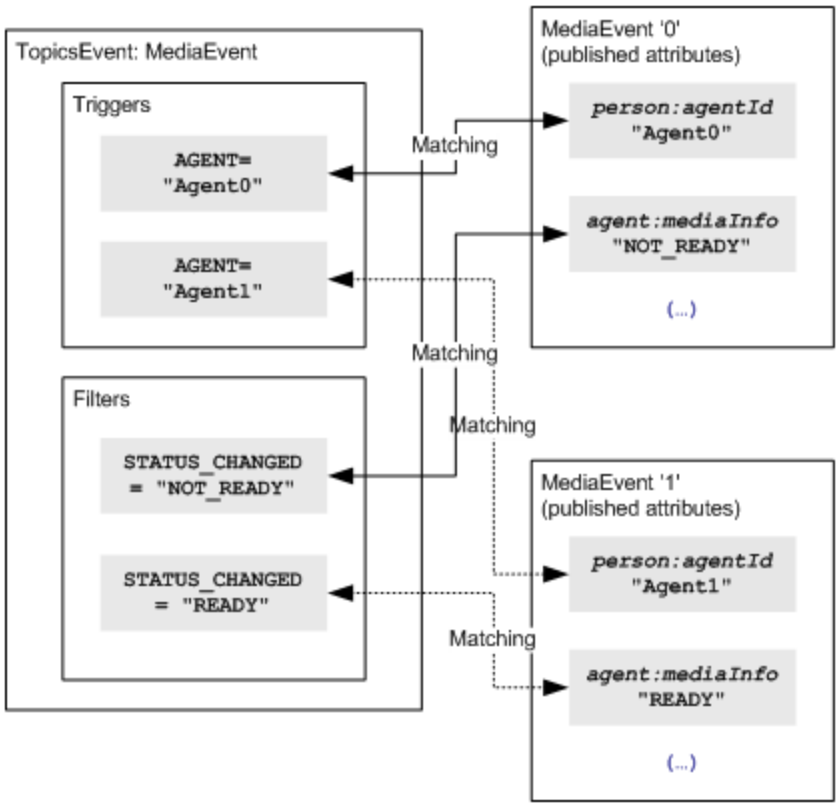
Understanding Triggers and Filters

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.

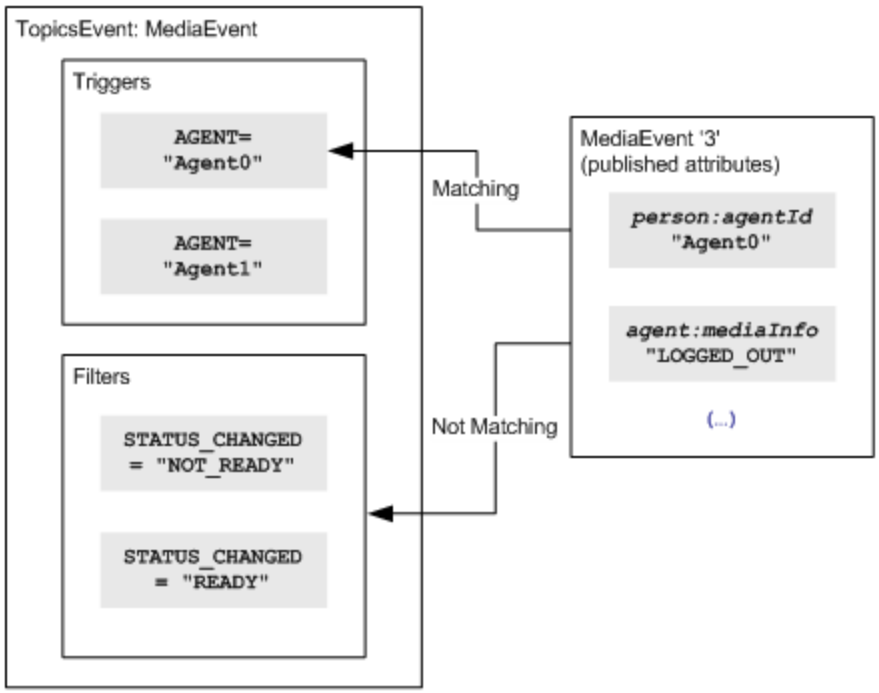
For example, your application can define a filter so as to receive `MediaEvents` only for a specific status change in the media. If `agent0` performs a successful login on certain media, your application might receive a `MediaEvent` due to a status change and associated with the `NOT_READY` agent media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

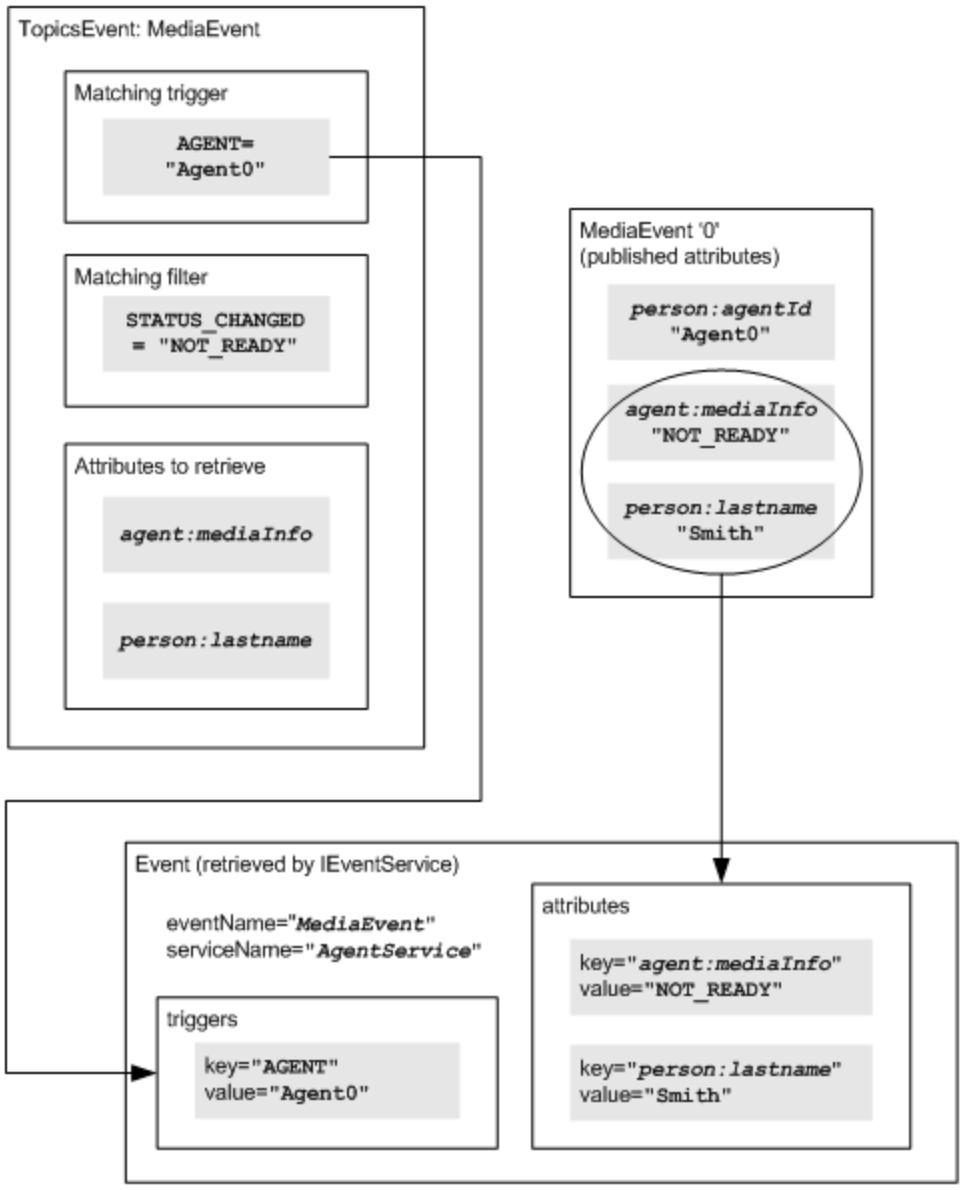
The second figure shows a `MediaEvent` that does not match a `TopicsEvent` defined for `MediaEvent`. Although the event matches the `Agent0` trigger, no filter corresponds.

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`