Platform SDK Developer's Guide

Event Handling Using the Message Broker Application Block

4/9/2025

Event Handling Using the Message Broker Application Block

> **Important**
>
> The Message Broker Application Block is considered a legacy product as of release 8.1.1 due to changes to the default event-receiving mechanism. Documentation related to this application block is retained for backwards compatibility. For information about event handling *without* use of the deprecated Message Broker Application Block, refer to the Event Handling article.

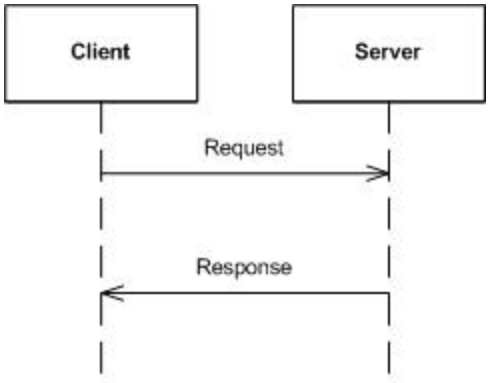
Once you have connected to a server using the Protocol Manager Application Block, much of the work of your application will be to send messages to that server and then handle the events you receive from it.

Genesys recommends that you use the **Message Broker Application Block** for most of your event handling needs. This article shows how to send and receive simple synchronous events without using Message Broker and then discusses how to use Message Broker for asynchronous event handling.

> **Tip**
>
> It is important to determine whether your application needs to use synchronous or asynchronous messages. In general, you will probably use only one or the other type in your application. If you decide to use synchronous messages, you must make sure that your code handles all of the messages you receive from your servers. For example, if you send a `RequestReadObjects` message to Configuration Server, you will receive several `EventObjectsRead` messages, followed by an `EventObjectsSent` message. If your application does not handle all of these messages, it will not work properly.

The messages you send to a server are in the form of requests. For example, you may send a request to log in an agent or to gather statistics. You might also send a request to update a configuration object, or to shut down an application.

In each of these cases, the server will respond with an event message, as shown below.



Some of the requests you send may best be handled with a synchronous response, while others may best be handled asynchronously. Let’s talk about synchronous requests first.

Java

Synchronous Requests

Sometimes you might want a synchronous response to your request. For example, if you are using the Open Media Platform SDK, you may want to log in an agent. To do this, you need to let the server know that you want to log in. And then you need to wait for confirmation that your login was successful.

The first thing you need to do is to create a login request, as shown here:

```
[Java]

RequestAgentLogin requestAgentLogin =
        RequestAgentLogin.create(
                tenantId,
                placeId,
                reason);
```

This version of `RequestAgentLogin.Create` specifies most of the information you will need in order to perform the login, but there is one more piece of data required. Here is how to add it:

```
[Java]

requestAgentLogin.setMediaList(mediaList);
```

Once you have created the request and set all required properties, you can make a synchronous request by using the request method of your ProtocolManagementService object, like this:

```
[Java]

Message response = null;
response = protocolManagementServiceImpl.getProtocol("Interaction_Server_App")
        .request(requestAgentLogin);
```

> **Tip**
>
> For information on how to use the ProtocolManagementServiceImpl class of the Protocol Manager Block to communicate with a Genesys server, see the article on Connecting to a Server.

There are two important things to understand when you use the `request` method:

- When you execute this method call, the calling thread will be blocked until it has received a response from the server.
- This method call will only return one message from the server. If the server returns subsequent messages in response to this request, you must process them separately. This can happen in the example of sending a `RequestReadObjects` message to Configuration Server, as mentioned at the beginning of this article.

The response from the server will come in the form of a `Message`. This is the interface implemented by all events in the Platform SDK. Some types of requests will be answered by an event that is specific to the request, while others may receive a more generic response of `EventAck`, which simply acknowledges that your request was successful. If a request fails, the server will send an `EventError`.

A successful `RequestAgentLogin` will receive an `EventAck`, while an unsuccessful one will receive an `EventError`. You can use a switch statement to test which response you received, as outlined here:

```
[Java]

switch(response.messageId())
{
        case EventAck.ID:
                OnEventAck(response);
        case EventError.ID:
                OnEventError(response);
        ...
}
```

Using Message Broker to Handle Asynchronous Requests

There are times when you need to receive asynchronous responses from a server.

First of all, some requests to a server can result in multiple events. For example, if you send a `RequestReadObjects` message, which is used to read objects from the Genesys Configuration Layer, Configuration Server may send more than one `EventObjectsRead` messages in response, depending on whether there is too much data to be handled by a single `EventObjectsRead`.

In other cases, events may be unsolicited. To continue with our example, once you have received all of the `EventObjectsRead` messages, Configuration Server will also send an `EventObjectsSent`, which confirms that it has completed your request.

To make an asynchronous request, you would use the send method of your `ProtocolManagementServiceImpl` class. For example, you might need to fetch information about

some objects in the Genesys Configuration Layer. Here is how to set up a RequestReadObjects, followed by the send:

[Java]

```
KeyValueCollection filterKey = new KeyValueCollection();
filterKey.addObject("switch_dbid", 113);
filterKey.addObject("dn_type", CfgDNType.CFGExtension.asInteger());
RequestReadObjects requestReadObjects = RequestReadObjects.create(
        CfgObjectType.CfgDN.asInteger(), filterKey);
protocolManagementServiceImpl.getProtocol("Config_Server_App")
        .send(requestReadObjects);
```

This snippet is searching for all DNs that have a type of Extension and are associated with the switch that has a database ID of 113.

There are several ways to handle the response from the server, but Genesys recommends that you use the Message Broker Application Block, which is included with the Platform SDK. Message Broker allows you to set up individual classes to handle specific events. It receives the events from the servers you are working with, and sends them to the appropriate handler class. Message Broker is a high-performance way to hide the complexity of event-driven programming — so you can focus on other areas of your application.

To use the Message Broker Application Block, add the following .jar file to the classpath for your application:

- messagebrokerappblock.jar

This .jar file was precompiled using the default Application Block code, and can be located at: <Platform SDK Folder>\lib.

> **Tip**
>
> You can also view or modify the Message Broker Application Block source code. To do this, open the Message Broker Java source files that were installed with the Platform SDK. The Java source files for this project are located at: <Platform SDK Folder>\applicationblocks\messagebroker\src\java. If you make any changes to the project, you will have to run Ant (or use the build.bat file for this Application Block) to rebuild the .jar archive listed above. After you run Ant, add the resulting .jar to your classpath.

Now you can add the appropriate import statements to your source code. For example:

[Java]

```
import com.genesyslab.platform.applicationblocks.commons.broker.*;
```

In order to use the Message Broker Application Block, you need to create an EventBrokerService object to handle the events your application receives. Since you are using the Protocol Manager Application Block to connect to your servers, as shown in the section on Connecting to a Server, you should specify the ProtocolManagementServiceImpl object in the EventBrokerService constructor:

[Java]

```
EventBrokerService mEventBrokerService = new EventBrokerService(
        (MessageReceiverSupport) protocolManagementServiceImpl
                .getReceiver());
```

You also need to set up the appropriate filters for your event handlers and register the handlers with the EventBrokerService. This allows that service to determine which classes will be used for event-handling. Note that you should register these classes before you open the connection to the server. Otherwise, the server might send events before you are ready to handle them. The sample below shows how to filter on Message ID, which is an integer associated with a particular message:

```
[Java]

mEventBrokerService.register(new ConfObjectsReadHandler(),
        new MessageIdFilter(EventObjectsRead.ID));
mEventBrokerService.register(new ConfObjectsSentHandler(),
        new MessageIdFilter(EventObjectsSent.ID));
mEventBrokerService.register(new StatPackageInfoHandler(),
        new MessageIdFilter(EventPackageInfo.ID));
```

Once you have registered your event-handling classes, you can activate the EventBrokerService and open the connection to your server. In the following snippet, connections are being opened to both Configuration Server and Stat Server:

```
[Java]

mEventBrokerService.activate();

protocolManagementServiceImpl.getProtocol("Config_Server_App")
        .open();
protocolManagementServiceImpl.getProtocol("Stat_Server_App").open();
```

At this point, you are ready to set up classes to handle the events you have received from the server. Here is a simple class that handles the EventObjectsRead messages:

```
[Java]

class ConfObjectsReadHandler implements Action {

        public void handle(Message obj) {
                EventObjectsRead objectsRead = (EventObjectsRead) obj;
                // Add processing here...
        }
}
```

As mentioned earlier, once Configuration Server has sent all of the information you requested, it will let you know it has finished by sending an EventObjectsSent message. Note that this handler has a structure that is similar to the one for EventObjectsRead:

```
[Java]

class ConfObjectsSentHandler implements Action {

        public void handle(Message obj) {
                EventObjectsSent objectsSent = (EventObjectsSent) obj;
                // Add processing here...
        }
}
```

Message Broker only routes non-null messages of the type you specify to your message

handlers. For example, if you send a RequestReadObjects and no objects in the Configuration Layer meet your filtering criteria, you will not receive an EventObjectsRead. In that case, you will only receive an EventObjectsSent. Therefore, you do not need to check for a null message in your EventObjectsRead handler.|2

The EventPackageInfo handler also has a similar structure, but in this case, we show how to print information about the statistics contained in the requested package:

```
[Java]

class StatPackageInfoHandler implements Action {

  public void handle(Message obj) {
    EventPackageInfo eventPackageInfo = (EventPackageInfo) obj;
    if (eventPackageInfo != null)
    {
      int statisticsCount = eventPackageInfo.getStatistics().getCount();
      StatisticsCollection statisticsCollection = eventPackageInfo.getStatistics();

      for (int i = 0; i < statisticsCount; i++)
      {
        Statistic statistic = statisticsCollection.getStatistic(i);

        System.out.println("\nStatistic Metric is: " +
             statistic.getMetric().toString());
        System.out.println("Statistic Object is: " +
            statistic.getObject());
        System.out.println("Statistic IntValue is: " +
            statistic.getIntValue());
        System.out.println("Statistic StringValue is: " +
            statistic.getStringValue());
        System.out.println("Statistic ObjectValue is: " +
            statistic.getObjectValue());
        System.out.println("Statistic ExtendedValue is: " +
            statistic.getExtendedValue());
         System.out.println("Statistic Tenant is: " +
            statistic.getObject().getTenant());
        System.out.println("Statistic Type is: " +
            statistic.getObject().getType());
        System.out.println("Statistic Id is: " +
            statistic.getObject().getId());
        System.out.println("Statistic TimeProfile is: " +
            statistic.getMetric().getTimeProfile());
         System.out.println("Statistic StatisticType is: " +
            statistic.getMetric().getStatisticType());
        System.out.println("Statistic TimeRange is: " +
            statistic.getMetric().getTimeRange());
        }
      }
    }
  }
```

Filtering Messages by Server

Each server in the Genesys environment makes use of a particular set of events that corresponds to the tasks of that server. For example, Configuration Server sends EventObjectsRead and EventObjectsSent messages, among others, while Stat Server's events include EventPackageInfo and EventPackageOpened. Although your applications can identify each of these events by name, it is more efficient to use the ID field associated with an event, which you specify as an int. You can do this by using a MessageIdFilter, as shown here:

```
[Java]

mEventBrokerService.register(new ConfEventErrorHandler(),
        new MessageIdFilter(EventError.ID));
```

However, the integer used for the Message ID of, say, a Configuration Server message, could be same as the integer used for a completely different message on another server. This could lead to problems if your application works with messages from more than one server. For example, if a multi-server application includes a handler that processes a specific type of message from the first server and that message has an ID of 12, any messages from the other servers that also have a Message ID of 12 will be sent by your `MessageIdFilter` to the same handler.

Fortunately, the Platform SDK allows you to filter messages on a server-by-server basis in addition to filtering on `MessageId`. Here is how to set up a Protocol Description object that allows you to specify that you want some of your handlers to work only with events that are coming from Configuration Server:

```
[Java]

ConfServerProtocol confServerProtocol = (ConfServerProtocol)
        protocolManagementServiceImpl.getProtocol("Config_Server_App");
ProtocolDescription configProtocolDescription = null;
if (confServerProtocol != null)
{
        configProtocolDescription =
                confServerProtocol.getProtocolDescription();
}
```

Once you have set up this Protocol Description, you can use it to indicate that you only want to process events associated with that server, in addition to specifying which event or events you want each handler to process:

```
[Java]

mEventBrokerService.register(new ConfEventErrorHandler(),
        new MessageIdFilter(configProtocolDescription, EventError.ID));
```

You are now ready to open the connection to Configuration Server:

```
[Java]

protocolManagementServiceImpl.
        getProtocol("Config_Server_App").open();
```

Using One Handler for Multiple Events

There may be times when you would like to use a single event handler for more than one event. In that case, you can create the handler and then register the appropriate events with it. For example, you might create a handler for both `EventObjectsRead` and `EventObjectsSent`:

```
[Java]

        class ConfEventHandler implements Action {
                ...
        }
```

You might use a case statement inside the handler, in order to process each event appropriately. In any case, once you have set up this handler, all you need to do is register both events with it, as

shown here:

```
[Java]

mEventBrokerService.register(new ConfEventHandler(),
        new MessageIdFilter(configProtocolDescription, EventObjectsRead.ID));
mEventBrokerService.register(new ConfEventHandler(),
        new MessageIdFilter(configProtocolDescription, EventObjectsSent.ID));
```

These are the basics of how to use the Message Broker Application Block. For more information, see the Using the Message Broker Application Block article.

.NET

Synchronous Requests

Sometimes you might want a synchronous response to your request. For example, if you are using the Open Media Platform SDK, you may want to log in an agent. To do this, you need to let the server know that you want to log in. And then you need to wait for confirmation that your login was successful.

The first thing you need to do is to create a login request, as shown here:

```
[C#]

RequestAgentLogin requestAgentLogin =
        RequestAgentLogin.Create(
                tenantId,
                placeId,
                reason);
```

This version of `RequestAgentLogin.Create` specifies most of the information you will need in order to perform the login, but there is one more piece of data required. Here is how to add it:

```
[C#]

requestAgentLogin.MediaList = mediaList;
```

Once you have created the request and set all required properties, you can make a synchronous request by using the `Request` method of your `ProtocolManagementService` object, like this:

```
[C#]

IMessage response =
        protocolManagementService["InteractionServer"].
            Request(requestAgentLogin);
```

> **Tip**

> For information on how to use the ProtocolManagementService class of the Protocol Manager Application Block to communicate with a Genesys server, see the article on Connecting to a Server Using the Protocol Manager Application Block.

There are two important things to understand when you use the Request method:

- When you execute this method call, the calling thread will be blocked until it has received a response from the server.
- This method call will only return one message from the server. If the server returns subsequent messages in response to this request, you must process them separately. This can happen in the example of sending a RequestReadObjects message to Configuration Server, as mentioned at the beginning of this article.

The response from the server will come in the form of an IMessage. This is the interface implemented by all events in the Platform SDK. Some types of requests will be answered by an event that is specific to the request, while others may receive a more generic response of EventAck, which simply acknowledges that your request was successful. If a request fails, the server will send an EventError.

A successful RequestAgentLogin will receive an EventAck, while an unsuccessful one will receive an EventError. You can use a switch statement to test which response you received, as outlined here:

```
[C#]

switch(response.Id)
{
        case EventAck.MessageId:
                OnEventAck(response);
        case EventError.MessageId:
                OnEventError(response);
        ...
}
```

Using Message Broker to Handle Asynchronous Requests

There are times when you need to receive asynchronous responses from a server.

First of all, some requests to a server can result in multiple events. For example, if you send a RequestReadObjects message, which is used to read objects from the Genesys Configuration Layer, Configuration Server may send more than one EventObjectsRead messages in response, depending on whether there is too much data to be handled by a single EventObjectsRead.

In other cases, events may be unsolicited. To continue with our example, once you have received all of the EventObjectsRead messages, Configuration Server will also send an EventObjectsSent, which confirms that it has completed your request.

To make an asynchronous request, you would use the Send method of your ProtocolManagementService class. Here is how to set up a RequestReadObjects, followed by the Send:

```
[C#]

KeyValueCollection filterKey = new KeyValueCollection();
filterKey.Add("switch_dbid", 113);
filterKey.Add("dn_type", (int) CfgDNType.Extension);
RequestReadObjects requestReadObjects =
        RequestReadObjects.Create(
                (int) CfgObjectType.CFGDN,
                filterKey);
protocolManagementService["ConfigServer"].Send(requestReadObjects);
```

This snippet is searching for all DNs that have a type of `Extension` and are associated with the switch that has a database ID of 113.

There are several ways to handle the response from the server, but Genesys recommends that you use the Message Broker Application Block, which is included with the Platform SDK. Message Broker allows you to set up individual handlers for specific events. It receives the events from the servers you are working with, and sends them to the appropriate handler. Message Broker is a high-performance way to hide the complexity of event-driven programming — so you can focus on other areas of your application.

To use the Message Broker Application Block, open the Solution Explorer for your application project and add a reference to the following file:

- Genesyslab.Platform.ApplicationBlocks.Commons.Broker.dll

This dll file is precompiled using the default Application Block code, and can be located at: `<Platform SDK Folder>\Bin`.

> **Tip**
>
> You can also view or modify the Message Broker Application Block source code. To do this, open the Message Broker Visual Studio project that was installed with the Platform SDK. The solution file for this project is located at: `<Platform SDK Folder>\ApplicationBlocks\MessageBroker`. If you make any changes to the project, you will have to rebuild the .dll file listed above.

Once you have added the reference, you can add a using statement to your source code:

```
[C#]

using Genesyslab.Platform.ApplicationBlocks.Commons.Broker;
```

In order to use the Message Broker Application Block, you need to create an `EventBrokerService` object to handle the events your application receives. Declare this object with your other fields:

```
[C#]

EventBrokerService eventBrokerService;
```

Then you can set up the `EventBrokerService` to receive events from the Protocol Manager Application Block's `ProtocolManagementService` class, which you are using to connect to your servers, as shown in the section on Connecting to a Server:

```
[C#]

eventBrokerService = new EventBrokerService(protocolManagementService.Receiver);
```

Now you are ready to set up your event handlers.

Note that there are two ways to do this. In 7.5, when Message Broker was introduced, you needed to use attributes to filter the events you wanted processed by a particular handler. Starting in 7.6, you can still do it that way, but you can also set up your filters in the statement that registers an event handler with the Event Broker service, rather than using attributes that are associated with the handler itself. This new method may perform better than the old way, but we will show you how to use both.

Using Event Handlers Without Attributes

Let us start by setting up a couple of event handlers. First, here is a simple handler for the `EventError` message:

```
[C#]

private void OnConfEventError(IMessage theMessage)
{
        EventError eventError = theMessage as EventError;
        /// Add processing here...
}
```

And here is one for the `EventObjectsRead` message:

```
[C#]

private void OnConfEventObjectsRead(IMessage theMessage)
{
        EventObjectsRead objectsRead = theMessage as EventObjectsRead;
        /// Add processing here...
}
```

As mentioned earlier, once Configuration Server has sent all of the information you requested, it will let you know it has finished by sending an `EventObjectsSent` message. Here is a handler for that:

```
[C#]

private void OnConfEventObjectsSent(IMessage theMessage)
{
        EventObjectsSent objectsSent = theMessage as EventObjectsSent;
        /// Add processing here...
}
```

Now you can set up the appropriate filters for your event handlers and register the handlers with the `EventBrokerService`. This allows that service to determine which classes will be used for event-handling. Note that you should register these handlers before you open the connection to the server. Otherwise, the server might send events before you are ready to handle them. The sample below shows how to filter on Message ID, which is an integer associated with a particular message:

```
[C#]

eventBrokerService.Register(
        this.OnConfEventError,
        new MessageIdFilter(EventError.MessageId));
```

```
eventBrokerService.Register(
        this.OnConfEventObjectsRead,
        new MessageIdFilter(EventObjectsRead.MessageId));
eventBrokerService.Register(
        this.OnConfEventObjectsSent,
        new MessageIdFilter(EventObjectsSent.MessageId));
```

Message Broker only routes non-null messages of the type you specify to your message handlers. For example, if you send a RequestReadObjects and no objects in the Configuration Layer meet your filtering criteria, you will not receive an EventObjectsRead. In that case, you will only receive an EventObjectsSent. Therefore, you do not need to check for a null message in your EventObjectsRead handler.

Filtering Messages by Server

Each server in the Genesys environment makes use of a particular set of events that corresponds to the tasks of that server. For example, Configuration Server sends `EventObjectsRead` and `EventObjectsSent` messages, among others, while Stat Server's events include `EventPackageInfo` and `EventPackageOpened`. Although your applications can identify each of these events by name, it is more efficient to use the ID field associated with an event, which you specify as an int. You can do this by using a `MessageIdFilter`, as shown here:

```
[C#]

eventBrokerService.Register(this.OnConfEventError);
```

However, the integer used for the Message ID of, say, a Configuration Server message, could be same as the integer used for a completely different message on another server. This could lead to problems if your application works with messages from more than one server. For example, if a multi-server application includes a handler that processes a specific type of message from the first server and that message has an ID of 12, any messages from the other servers that also have a Message ID of 12 will be sent by your `MessageIdFilter` to the same handler.

Fortunately, the Platform SDK allows you to filter messages on a server-by-server basis in addition to filtering on `MessageId`. Here is how to set up a Protocol Description object that allows you to specify that you want some of your handlers to work only with events that are coming from Configuration Server:

```
[C#]

ConfServerProtocol confServerProtocol =
        protocolManagementService["Config_Server_App"]
                as ConfServerProtocol;
ProtocolDescription configProtocolDescription = null;
if (confServerProtocol != null)
{
        configProtocolDescription =
                confServerProtocol.ProtocolDescription;
}
```

Once you have set up this Protocol Description, you can use it to indicate that you only want to process events associated with that server, in addition to specifying which event or events you want each handler to process:

```
[C#]

eventBrokerService.Register(
```

```
        this.OnConfEventError,
                new MessageIdFilter(
                        configProtocolDescription,
                        EventError.MessageId));
eventBrokerService.Register(
        this.OnConfEventObjectsRead,
                new MessageIdFilter(
                         configProtocolDescription,
                        EventObjectsRead.MessageId));
eventBrokerService.Register(
        this.OnConfEventObjectsSent,
                new MessageIdFilter(
                        configProtocolDescription,
                        EventObjectsSent.MessageId));
```

You are now ready to open the connection to Configuration Server:

```
[C#]

protocolManagementService["Config_Server_App"].Open();
```

Using One Handler for Multiple Events

There may be times when you would like to use a single event handler for more than one event. In that case, you can create the handler and then register the appropriate events with it. For example, you might create a handler for both EventObjectsRead and EventObjectsSent:

```
[C#]

private void OnConfEvents (IMessage theMessage) {
        ...
}
```

You might use a case statement inside the handler, in order to process each event appropriately. In any case, once you have set up this handler, all you need to do is register both events with it, as shown here:

```
[C#]

eventBrokerService.Register(
        this.OnConfEvents,
                new MessageIdFilter(
                        configProtocolDescription,
                        EventObjectsRead.MessageId));
eventBrokerService.Register(
        this.OnConfEvents,
                new MessageIdFilter(
                        configProtocolDescription,
                        EventObjectsSent.MessageId));
```

Using Attributes with Your Event Handlers

As mentioned above, you can also use attributes to filter your event handlers. It is important to note that this may not perform as well as the method outlined above, but in case you would like to use attributes in your application, here is how to proceed.

When you use attributes, you have to specify the name of the protocol object you are using, and the name of the SDK it is part of, as shown here:

```
[C#]

private const string protocolName = "ConfServer";
private const string sdkName = "Configuration";
```

These values can be determined by accessing the `ProtocolDescription.ProtocolName` and `ProtocolDescription.SdkName` properties of your protocol object. They are also provided in the following table.

SDK	SdkName	Protocol Object	ProtocolName
Configuration Platform SDK	Configuration	ConfServerProtocol	ConfServer
Contacts Platform SDK	Contacts	UniversalContactServerProtocol	ContactServer
Management Platform SDK	Management	• LocalControlAgentProtocol • MessageServerProtocol • SolutionControlServerProtocol	• LocalControlAgent • MessageServer • SolutionControlServer
Open Media Platform SDK	OpenMedia	• InteractionServerProtocol • ExternalServiceProtocol	• InteractionServer • ExternalService
Outbound Contact Platform SDK	Outbound	OutboundServerProtocol	OutboundServer
Routing Platform SDK	Routing	• RoutingServerProtocol • UrsCustomProtocol	• RoutingServer • CustomServer
Statistics Platform SDK	Reporting	StatServerProtocol	StatServer
Voice Platform SDK	Voice	TServerProtocol	TServer
Web Media Platform SDK	WebMedia	• BasicChatProtocol • FlexChatProtocol • EmailProtocol • EspEmailProtocol • CallbackProtocol	• BasicChat • FlexChat • Email • EspEmail • Callback

Table 1: Platform SDK SdkName and ProtocolName Values

You also need to register the methods you will handle your events with. This allows the `EventBrokerService` to determine which methods will be used for event-handling. When registering for event handlers that use attributes, you only specify the name of the event-handling method. In this case, you need to handle three different events. Note that you should register these methods before you open the connection to the server, as shown here. Otherwise, the server might send events before you are ready to handle them:

```
[C#]
```

```
eventBrokerService.Register(this.OnConfEventObjectsRead);
eventBrokerService.Register(this.OnConfEventObjectsSent);
eventBrokerService.Register(this.OnConfEventError);
protocolManagementService["Config_Server_App"].Open();
```

At this point, you are ready to set up methods to handle the events you have received from the server. Here is a simple method that handles the `EventError` message:

```
[C#]

[MessageIdFilter(EventError.MessageId, ProtocolName = "ConfServer", SdkName =
"Configuration")]
private void OnConfEventError(IMessage theMessage)
{
        EventError eventError = theMessage as EventError;
        /// Add processing here...
}
```

Notice that there is a `MessageIdFilter` attribute right before the method body. This attribute indicates that all `EventError` messages for the Configuration Platform SDK's Configuration protocol will be handled by this method.

The attributes and methods for `EventObjectsRead` have a similar structure:

```
[C#]

[MessageIdFilter(EventObjectsRead.MessageId, ProtocolName = "ConfServer", SdkName =
"Configuration")]
private void OnConfEventObjectsRead(IMessage theMessage)
{
        EventObjectsRead objectsRead = theMessage as EventObjectsRead;
        /// Add processing here...
}
```

And so do the attributes and methods for `EventObjectsSent`:

```
 [C#]

[MessageIdFilter(EventObjectsSent.MessageId, ProtocolName = "ConfServer", SdkName =
"Configuration")]
private void OnConfEventObjectsSent(IMessage theMessage)
{
        //protocolManagementService["Config_Server_App"].Close();
        EventObjectsSent objectsSent = theMessage as EventObjectsSent;
        /// Add processing here...

}
```

If you want to process more than one event with a single handler, you can set up multiple attributes for that handler, like this:

```
[C#]

[MessageIdFilter(EventObjectsRead.MessageId, ProtocolName = "ConfServer", SdkName =
"Configuration")]
[MessageIdFilter(EventObjectsSent.MessageId, ProtocolName = "ConfServer", SdkName =
"Configuration")]
```

```
private void OnConfEvents (IMessage theMessage) {
        ...
}
```

These are the basics of how to use the Message Broker Application Block. For more information, see the Using the Message Broker Application Block article.