Platform SDK Developer's Guide

Migration from Message Broker Application Block Usage

5/4/2025

Contents

Migration from Message Broker Application Block Usage

Introduction

Starting with release 8.5.0, use of the Message Broker Application Block is no longer recommended. This application block is now considered as legacy component and has been deprecated.

This article provides an overview of how to migrate existing applications, and outlines how behavior that was previously handled by the application block should now be implemented.

Tip

If you choose to continue using the Message Broker Application Block, please note that the common interfaces for COM Application Block and Message Broker have been moved to an individual file (`commonsappblock.jar` for Java, `Genesyslab.Platform.ApplicationBlocks.Commons.dll` for .NET) starting with release 8.5.0.

Functional Aspects of the Message Broker Application Block

SubscriptionService + Subscriber Pattern

The broker service is the main component of the application block. It contains several facade service implementations including: `BrokerService`, `EventBrokerService`, `EventReceivingBrokerService`, `RequestBrokerService` and `RequestReceivingBrokerService`.

Subscription Filters

Each `Subscriber` has its own events filter (using the `Predicate<T>` interface). The event broker service applies incoming events to all of a registered Subscriber's filters.

The application block contains several predefined `Message` filters for use with protocol message brokers. For instance, there are:

- `MessageFilter`: it may filter protocol messages by `ProtocolDescription`, `ProtocolId`, or `EndpointName`. It also has ability to be negated.
- `MessageIdFilter`: an extension of `MessageFilter` with the ability to filter specific `MessageId`.
- `MessageNameFilter`: an extension of `MessageFilter` with the ability to filter specific `MessageName`.

- `MessageRangeFilter`: an extension of `MessageFilter` with the ability to filter specific set of `MessageIds`.

The application block also contains helping classes to make composite filters: `AndPredicate` and `OrPredicate`.

Synchronous and Asynchronous Execution

- The application block contains a general purpose synchronous `BrokerService`<T> implementation. It does not use additional threads, does not have any queues, and executes generic events (of type <T>) publishing immediately.
- Asynchronous generic broker service `AsyncBrokerService`<T>. The difference between this broker and the synchronous broker is that events are published with the specified invoker (one or more other threads).
- Old type asynchronous broker services: `EventBrokerService` and `RequestBrokerService`. These services use dedicated internal thread to get events from intermediate queue and pass to invoker.
- `EventReceivingBrokerService` and `RequestReceivingBrokerService`. Compared to the old style brokers, these do not use the additional threads and replace the intermediate queues (implementing receiving interfaces).

Functional Replacements

COM Application Block ConfService Initialization

The first place where you may need to update application code to not use the Message Broker Application Block is with old-style initialization of `ConfService`.

Initialization of *ConfService* with Message Broker usage may look like following:

[+] Java Code Sample

```
Endpoint csEndpoint = new Endpoint(csEPName, csHost, csPort);
ConfServerProtocol csProtocol = new ConfServerProtocol(csEndpoint);
csProtocol.setClientName(clientName);
csProtocol.setClientApplicationType(clientType);
csProtocol.setUserName(username);
csProtocol.setUserPassword(password);
csProtocol.open();

EventBrokerService msgBroker = BrokerServiceFactory.CreateEventBroker(csProtocol);

IConfService confService = ConfServiceFactory.createConfService(csProtocol, msgBroker);
```

[+] .NET Code Sample

```
Endpoint csEndpoint = new Endpoint(csEPName, csHost, csPort);
ConfServerProtocol csProtocol = new ConfServerProtocol(csEndpoint);
csProtocol.ClientName = clientName;
csProtocol.ClientApplicationType = clientType;
csProtocol.UserName =userName;
csProtocol.UserPassword=password;
csProtocol.Open();

EventBrokerService msgBroker = BrokerServiceFactory.CreateEventBroker(csProtocol);
IConfService confService = ConfServiceFactory.CreateConfService(csProtocol, msgBroker);
```

The new initialization approach would be following:

[+] Java Code Sample

```
Endpoint csEndpoint = new Endpoint(csEPName, csHost, csPort);
ConfServerProtocol csProtocol = new ConfServerProtocol(csEndpoint);
csProtocol.setClientName(clientName);
csProtocol.setClientApplicationType(clientType);
csProtocol.setUserName(username);
csProtocol.setUserPassword(password);

IConfService confService = ConfServiceFactory.createConfService(csProtocol);

csProtocol.open();
```

[+] .NET Code Sample

```
Endpoint csEndpoint = new Endpoint(csEPName, csHost, csPort);
ConfServerProtocol csProtocol = new ConfServerProtocol(csEndpoint);
csProtocol.ClientName = clientName;
csProtocol.ClientApplicationType = clientType;
csProtocol.UserName =userName;
csProtocol.UserPassword=password;

IConfService confService = ConfServiceFactory.CreateConfService(csProtocol);
csProtocol.Open();
```

This change eliminates the redundant internal messages queue and the redundant thread.

> **Tip**
>
> Protocol `open()` has to be done after `ConfService` creation. By this way `ConfService` initializes its own internal instance of `MessageHandler`, so, the protocol has to be closed to allow it. And if the protocol instance has initialized custom `MessageHandler`, it will be overridden with the `ConfServices`' internal one.

If there is a need to receive asynchronous protocol messages from `ConfServerProtocol` and `ConfService` pair, your application may use `ConfService.setUserMessageHandler(MessageHandler)` instead of a subscription on the message broker:

[+] Java Code Sample

```
MessageHandler msgHandler = new MessageHandler() {
    public void onMessage(Message message) {
        // do something with incoming async protocol message
    }
};
confService.setUserMessageHandler(msgHandler);
```

[+] .NET Code Sample

```
confService.Protocol.Received += (sender, e) =>
{
   var args = e as MessageEventArgs;
   if ((args != null) && (args.Message!=null))
   {
     // do something with incoming async protocol message
   }
};
```

The message handling method `MessageHandler.onMessage(message)` will be executed using the protocol invoker thread.

Message Broker Component

Broker (Subscribers-Side Replacement)

The most common part of different types of broker services is functionality for message/event passing to service `Subscriber`'s.

So, when we have some broker service instance with initialization of several subscribers like this:

```
broker.register(subscriber1);
broker.register(subscriber2);
broker.register(subscriber3);
```

it would be replaced with function like:

[+] Java Code Sample

```
void doNotifySubscribers(final Message message) {
    if (<subscriber1 filter on 'message'>) {
        // do subscriber1 handling of 'message'
    }
    if (<subscriber2 filter on 'message'>) {
        // do subscriber2 handling of 'message'
    }
    if (<subscriber3 filter on 'message'>) {
        // do subscriber3 handling of 'message'
    }
}
```

[+] .NET Code Sample

```
void doNotifySubscribers(IMessage message) {
    if (<subscriber1 filter on 'message'>) {
        // do subscriber1 handling of 'message'
    }
    if (<subscriber2 filter on 'message'>) {
        // do subscriber2 handling of 'message'
    }
    if (<subscriber3 filter on 'message'>) {
        // do subscriber3 handling of 'message'
    }
}
protocol.Received += (sender, e) =>
{
   var args = e as MessageEventArgs;
   if ((args != null) && (args.Message!=null))
   {
     doNotifySubscribers(args.Message)
   }
};
```

In most cases it is possible to optimize such a function to do not execute all the filters for all incoming messages, but use "if {} else if {} ...", "switch(<>) {}", or, even do not use explicit filtering taking into account specifics of the broker instance like expected set of incoming messages, their types, etc.

Subscribers Filters Replacement

Message Broker Application Block contains several predefined filters for protocol messages filtering.

The messages filters provide several properties for filtering. Each of the properties corresponds to specific attribute of protocol message. So, if some property is initialized and is not null, then it is to be applied for incoming messages filtering.

For example, here is a sample filter logic:

```
Action<Message> action = new Action<Message>() {
    public void handle(final Message message) {
        // do something with 'message'
    }
};
brokerService.register(action, new MessageIdFilter(
        ConfServerProtocol.PROTOCOL_DESCRIPTION, EventObjectUpdated.ID));
```

Such broker would be changed to something like:

[+] Java Code Sample

```
void doNotifySubscribers(final Message message) {
    if (ConfServerProtocol.PROTOCOL_DESCRIPTION.equals(message.getProtocolDescription())
            && (message.messageId() == EventObjectUpdated.ID)) {
        // do something with 'message'
    }
}
```

[+] .NET Code Sample

```
void doNotifySubscribers(IMessage message)
{
  if (ConfServerProtocol.Description.Equals(message.ProtocolDescription) && (message.Id ==
EventObjectUpdated.MessageId))
  {
    // do something with 'message'
  }
}
```

Broker (Service-Side Replacement)

The other side of a broker component is an entrance of messages/events for notification.

On this side we may have several different cases of broker service usage:

It may be a general broker for general event type like `BrokerService<T>`, `AsyncBrokerService<T>`, or some other broker type with explicit events publishing with `broker.publish(event);`.

In this case `broker.publish(event);` may be simply replaced with direct call to the newly created `doNotifySubscribers(event);`.

Usage of `EventReceivingBrokerService` as `MessageReceiver` or `MessageHandler` would be replaced with direct `MessageHandler`:

```
EventReceivingBrokerService evBroker = new EventReceivingBrokerService();
evBroker.register(subscriber1);
evBroker.register(subscriber2);
protocol.setMessageHandler(evBroker);
protocol.open();
```

would be changed to:

[+] Java Code Sample

```
protocol.setMessageHandler(new MessageHandler() {
    public void onMessage(final Message message) {
        if (<subscriber1 filter on 'message'>) {
            // do subscriber1 handling of 'message'
        }
        if (<subscriber2 filter on 'message'>) {
            // do subscriber2 handling of 'message'
        }
    }
});
protocol.open();
```

[+] .NET Code Sample

```
private void OnReceivedHandler(object sender, EventArgs e)
{
  var args = e as MessageEventArgs;
  if ((args != null) && (args.Message != null))
  {
        if (<subscriber1 filter on 'message'>) {
            // do subscriber1 handling of 'message'
        }
        if (<subscriber2 filter on 'message'>) {
            // do subscriber2 handling of 'message'
        }
  }
}


protocol.Received += OnReceivedHandler;
protocol.Open();
```

Custom `MessageHandler` may be shared between several protocols connections just like

`EventReceivingBrokerService`.

Usage of old style "EventBrokerService"

It's a special case of broker service which is based on intermediate messages queue, and it uses extra thread to synchronously read messages from the queue and pass them for handling.

For example:

```
EventBrokerService broker = BrokerServieFactory.CreateEventBroker(protocol);

// or:

EventBrokerService broker = new EventBrokerService(protocol);
broker.activate();
```

Replacing of such kind of broker with `MessageHandler` (see above) eliminates redundant messages queue and the redundant thread.

Request Broker Component

Request broker service is a kind of message broker for handling of clients requests on `ServerChannel` side.

There are two types of request broker services: `RequestBrokerService` and `RequestReceivingBrokerService`.

Actual recommendation for requests handling logic on `ServerChannel` is to do not use any broker service, but explicitly handle incoming requests in accordance to application specific architecture (without additional shared queue).

It may be done with custom implementation of request receiver:

[+] Java Code Sample

```
RequestReceiverSupport rqReceiver = new RequestReceiverSupport() {
    public void processRequest(final RequestContext incomingRequest) {
        // ! pass some task to do something with 'incomingRequest' in separated thread or
thread pool !
        //   like "executor.execute(new RequestHandlerTask(incomingRequest));"
    }
    public RequestContext receiveRequest() throws InterruptedException, IllegalStateException
{
        throw new <Exception>("requests are handled asynchronously");
    }
    // ... implementation for other RequestReceiverSupport methods goes here ...
};
serverChannel.setReceiver(rqReceiver);
serverChannel.open();
```

[+] .NET Code Sample

```
private class CustomRequestReceiver : IRequestReceiverSupport
{
  public IRequestContext ReceiveRequest(TimeSpan timeout)
  {
    throw new InvalidOperationException("requests are handled asynchronously");
  }
  public void ProcessRequest(IRequestContext request)
  { // ! pass some task to do something with 'incomingRequest' in separated thread or thread
pool !
    // for example:
    ThreadPool.QueueUserWorkItem(delegate(object state){
      var context = state as IRequestContext;
      // process request context
    }, request);
  }
  // ... implementation for other RequestReceiverSupport methods goes here ...
}

serverChannel.SetReceiver(new CustomRequestReceiver());
serverChannel.Open();
```