# Platform SDK Developer's Guide

## Using the Message Broker Application Block

12/13/2025

# Using the Message Broker Application Block

> **Important**
>
> This application block is considered a legacy product starting with release 8.1.1. Documentation is provided for backwards compatibility, but new development should consider using the improved method of message handling.

The Message Broker Application Block is a reusable production-quality component that makes it easy for your applications to handle events in an efficient way. It has been designed using industry best practices and provided with source code so it can be used "as is," extended, or tailored if you need to. Please see the License Agreement for details.

For information on the other application blocks that ship with the Genesys SDKs, consult Introducing the Platform SDK.

## Java

## Installing the Message Broker Application Block

### Software Requirements

To work with the Message Broker Application Block, you must ensure that your system meets the software requirements established in the Genesys Supported Operating Environment Reference Guide, as well as meeting the following minimum software requirements:

- JDK 1.6 or higher

### Building the Message Broker Application Block

To build the Message Broker Application Block:

1. Open the `<Platform SDK Folder>\applicationblocks\messagebroker` folder.
2. Run either `build.bat` or `build.sh`, depending on your platform.

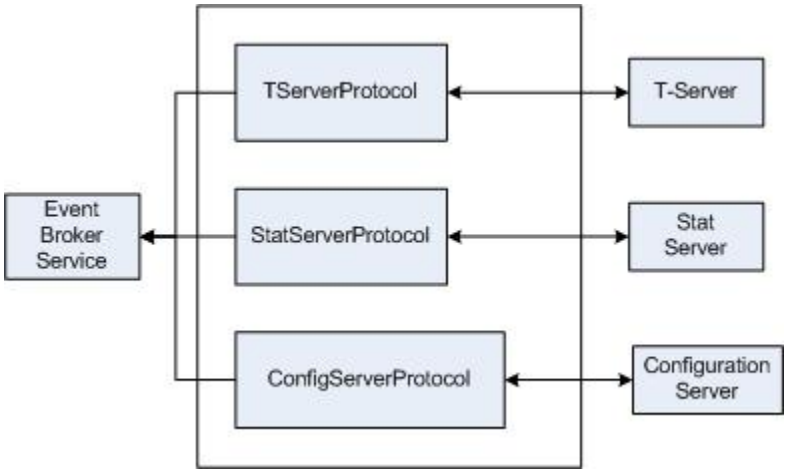
This will create the `messagebrokerappblock.jar` file, located within the `<Platform SDK Folder>\applicationblocks\messagebroker\dist\lib` directory.

### Working with the Message Broker Application Block

You can find basic information on how to use the Message Broker Application Block in the article on Event Handling Using the Message Broker Application Block.

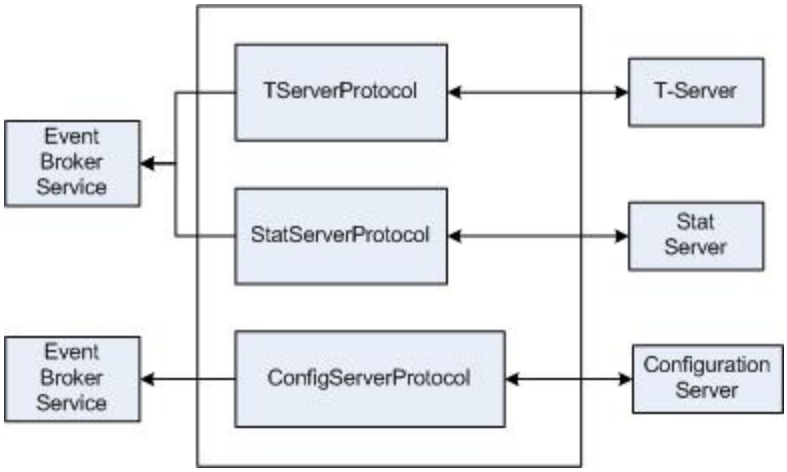
## Configuring Message Broker

When you first work with Message Broker, you will probably use a single instance of `EventBrokerService`. This means that all messages coming into your application will first pass through this single instance, as shown in below. Note that configuration diagrams used here do not show the Protocol Manager Application Block, in order to focus on the architecture of Message Broker.

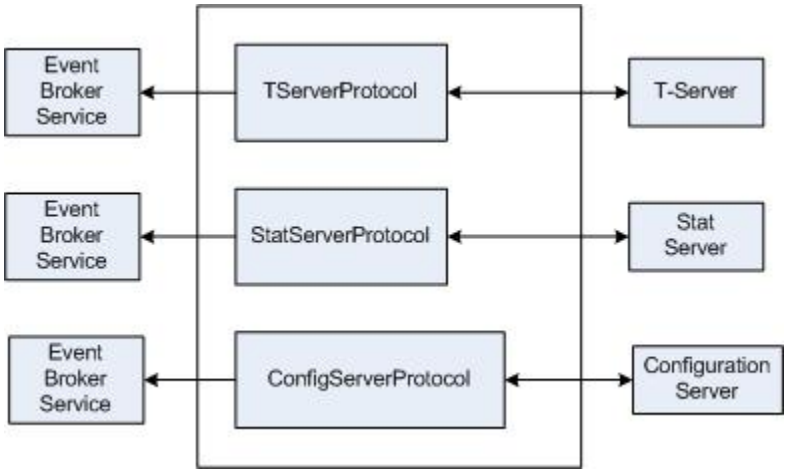



However, there may be high-traffic scenarios that require multiple instances of Message Broker. This might happen if you have one or more servers whose events use so much of Message Broker's processing time that events from other servers must wait for an unacceptable amount of time. In that case, you could dedicate an instance of `EventBrokerService` to the appropriate server.

For example, you may have a scenario in which you frequently receive large volumes of statistics. To handle that situation, you could dedicate an `EventBrokerService` instance to Stat Server. In other situations, you might regularly receive large amounts of Configuration Layer data from Configuration Server. You could handle this in a similar way by giving Configuration Server its own instance of `EventBrokerService`, as shown here:




Sometimes you may have large message volumes for each server, in which case you could use a separate instance of `EventBrokerService` for each server, as shown here.




## Using Message Filters

Message Broker comes with several types of message filters. You can filter on individual messages using `MessageIdFilter` or `MessageNameFilter`. In most cases you will want to use `MessageIdFilter`, as it is more efficient than `MessageNameFilter`. You can also use a `MessageRangeFilter` to filter on several messages at a time.

As shown in the article on Event Handling Using the Message Broker Application Block, you can specify these filters when you register an event handler with the Event Broker Service. Here is a sample of how to set up a `MessageIdFilter`:

```
[Java]

eventBrokerService.register(new StatPackageOpenedHandler(),
        packageEvents);
```

There may be times when you want to process several events in the same event handler. In such cases, you can use a `MessageRangeFilter`, which will direct all of these events to that handler. Here is a sample of how to set up the filter:

```
[Java]

int[] messageRange = new int[] {EventPackageOpened.ID, EventPackageClosed.ID};
MessageRangeFilter packageStatusEvents = new MessageRangeFilter
        (messageRange);
eventBrokerService.register(new StatPackageStatusChangedHandler(),
        packageStatusEvents);
```

Your event handler might look something like this:

```
[Java]

class StatPackageStatusChangedHandler implements Action {

        public void handle(Message obj) {
                // Common processing goes here...
                if (obj.messageId() == EventPackageOpened.ID) {
                        // EventPackageOpened processing goes here...
                } else {
                        // EventPackageClosed processing goes here...
                }
        }
}
```

Some servers use events that have the same name as events used by another server. One example is `EventError`, which is used by just about every server except Stat Server. The Event Handling Using the Message Broker Application Block article shows how to use a Protocol Description object to filter events by server type in order to avoid confusion when handling these events.

There also may be times when you have several instances of a given server in your environment and you want to filter by a specific one. To do this, first specify an `Endpoint` for that server, using a name for the server in the `Endpoint` constructor:

```
[Java]

String statServer1EndpointName = "StatServer1";
Endpoint statServer1Endpoint =
        new Endpoint(statServer1EndpointName, statServer1Uri);
```

Now create the filter:

```
[Java]

MessageIdFilter statServer1EndpointFilter =
        new MessageIdFilter(EventPackageOpened.ID);
```

And set the EndpointName in the filter:

```
[Java]

statServer1EndpointFilter.setEndpointName(statServer1EndpointName);
```

When you register this filter, the handler you specify will only receive messages that were sent from the instance you mentioned above:

```
[Java]

eventBrokerService.register(new StatPackageOpenedHandler_StatServer1(),
        statServer1EndpointFilter);
```

# Architecture and Design

The Message Broker Application Block is designed to make it easy for your applications to handle events in an efficient way.

Message Broker allows you to set up individual classes to handle specific events coming from Genesys servers. It receives all of the events from the servers you specify, and sends each one 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.

> **Tip**
>
> Message Broker has been designed for use with the Protocol Manager Application Block. Protocol Manager is another high-performance component that makes it easy for your applications to connect to Genesys servers. You can find basic information on how to use the Protocol Manager Application Block in the article on Connecting to a Server.

## The Message Broker Application Block Architecture

The Message Broker Application Block uses a service-based API that enables you to write individual methods that handle one or more events.

For example, you might want to handle every occurrence of `EventAgentLogin` with a specific dedicated method, while there might be other events that you wish to send to a common event-handling method. Message Broker allows you write these methods and register them with an event broker that manages them for you.

### Message Filters

Message Broker uses *message filters* to identify specific messages, assign them to specified methods, and route them accordingly.

# Design Patterns

This section gives an overview of the design patterns used in the Message Broker Application Block.

## Publish/Subscribe Pattern

There are many occasions when one class (the subscriber) needs to be notified when something

changes in another class (the publisher). The Message Broker Application Block use the Publish/Subscribe pattern to inform the client application when events arrive from the server.

### Factory Method Pattern

It is common practice for a class to include constructors that enable clients of the class instantiate it. There are times, however, when a client may need to instantiate one of several different classes. In some of these situations, the client should not need to decide which class is being created. In this case, a Factory Method pattern is used. The Factory Method pattern lets a class developer define the interface for creating an object, while retaining control of which class to instantiate.

## How To Properly Manage the EventBrokerService Lifecycle

Unfortunately, a commonly encountered problem is that users create `EventBrokerService` but do not dispose of it properly. `EventBrokerService` exclusively uses an invoker thread to run an infinite cycle with `MessageReceiver.receive()` and incoming messages handling logic. `EventBroker` is created by user code, so it should be disposed by user code as well. Useful methods are `MessageBrokerService.deactivate()` and `MessageBrokerService.dispose()`.

In PSDK 8.1 this class is deprecated and a new one is added to resolve the problem with thread waiting: `EventReceivingBrokerService`. This new class implements the `MessageReceiver` interface and may be used as external receiver for Platform SDK protocols. In this case, we have no intermediate redundant queue and incoming messages are delivered from protocol(s) to handler(s) directly. This class still requires async invoker to execute messages handling, but in this case the invoker is called once per incoming message, so it's thread is not blocked during the `.receive()` operation.

So, `EventReceivingBrokerService` does not need `.dispose()` and is GC friendly.

> **Tip**
> A similar change has been made to `RequestBrokerService`.

Also note that the `Invoker` instance still represents a "costly" resource (thread) and is managed by user code, so proper attention (allocation/deallocation) is required.

**Q:** Does it matter if the event broker service is created by the `BrokerServiceFactory` or not?

**A:** Actually, `BrokerServiceFactory` just creates and activates the corresponding broker instance. So if a broker is created by a call to the factory, it must be disposed of by user code in accordance to its usage there.

## .NET

# Installing the Message Broker Application Block

Before you install the Message Broker Application Block, it is important to review the software requirements and the structure of the software distribution.

## Software Requirements

To work with the Message Broker Application Block, you must ensure that your system meets the software requirements established in the Genesys Supported Operating Environment Reference Guide.

## Building the Message Broker Application Block

The Platform SDK distribution includes a `Genesyslab.Platform.ApplicationBlocks.Commons.Broker.dll` file that you can use as is. This file is located in the `bin` directory at the root level of the Platform SDK directory. To build your own copy of this application block, follow the instructions below:

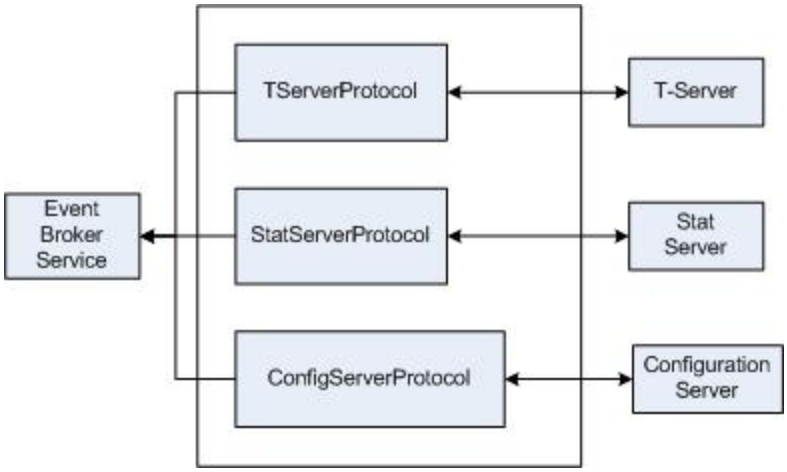
1. Open the `<Platform SDK Folder>\ApplicationBlocks\MessageBroker` folder.
2. Double-click `MessageBroker.sln`.
3. Build the solution.

## Working with the Message Broker Application Block

You can find basic information on how to use the Message Broker Application Block in the article on Event Handling Using the Message Broker Application Block.

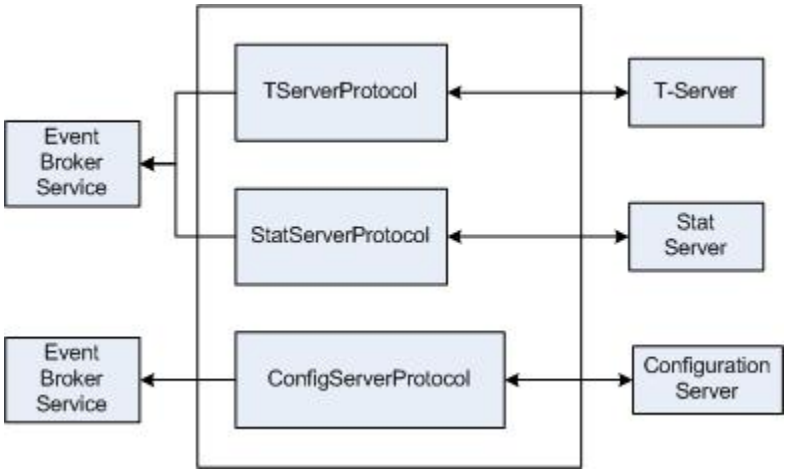
# Configuring Message Broker

When you first work with Message Broker, you will probably use a single instance of `EventBrokerService`. This means that all messages coming into your application will first pass through this single instance, as shown in the figure below. Note that the following configuration diagrams do not show the Protocol Manager Application Block, in order to focus on the architecture of Message Broker.

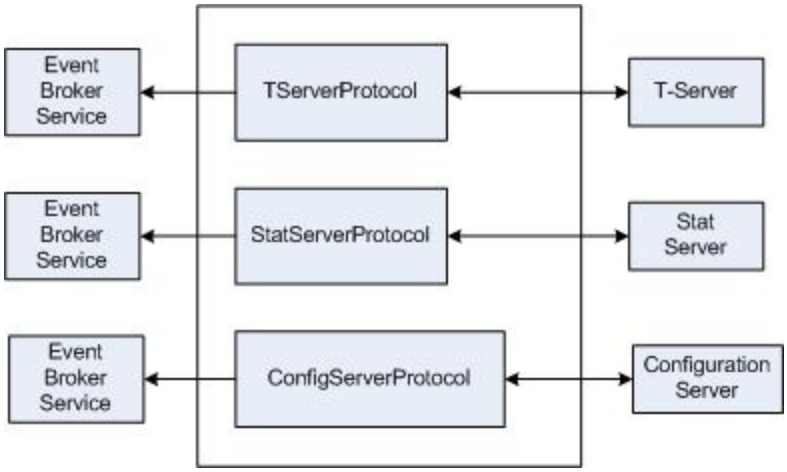



However, there may be high-traffic scenarios that require multiple instances of Message Broker. This might happen if you have one or more servers whose events use so much of Message Broker's processing time that events from other servers must wait for an unacceptable amount of time. In that case, you could dedicate an instance of `EventBrokerService` to the appropriate server.

For example, you may have a scenario in which you frequently receive large volumes of statistics. To handle that situation, you could dedicate an `EventBrokerService` instance to Stat Server. In other situations, you might regularly receive large amounts of Configuration Layer data from Configuration Server. You could handle this in a similar way by giving Configuration Server its own instance of `EventBrokerService`, as shown in the following figure:




Sometimes you may have large message volumes for each server, in which case you could use a separate instance of `EventBrokerService` for each server, as shown here.




## Using Message Filters

Message Broker comes with several types of message filters. You can filter on individual messages using `MessageIdFilter` or `MessageNameFilter`. In most cases you will want to use `MessageIdFilter`, as it is more efficient than `MessageNameFilter`. You can also use a `MessageRangeFilter` to filter on several messages at a time.

As shown in the article on Event Handling Using the Message Broker Application Block in the beginning of this guide, you can specify these filters when you register an event handler with the Event Broker Service. Here is a sample of how to set up a `MessageIdFilter`:

```
[C#]

eventBrokerService.Register(this.OnEventPackageClosed,
        new MessageIdFilter(EventPackageClosed.MessageId));
```

There may be times when you want to process several events in the same event handler. In such cases, you can use a `MessageRangeFilter`, which will direct all of these events to that handler. Here is a sample of how to set up the filter:

```
[C#]

eventBrokerService.Register(this.OnEventPackageStatusChanged, new MessageRangeFilter(new
int[]   {
        EventPackageOpened.MessageId, EventPackageClosed.MessageId}));
```

Your event handler might look something like this:

```
[C#]

private void OnEventPackageStatusChanged(IMessage theMessage)
{
        // Common processing goes here...
        if (theMessage.Id == EventPackageOpened.MessageId)
        {
                // EventPackageOpened processing goes here...
        }
        else
```

```
        {
                // EventPackageClosed processing goes here...
        }
}
```

Some servers use events that have the same name as events used by another server. One example is `EventError`, which is used by just about every server except Stat Server. The Event Handling Using the Message Broker Application Block article shows how to use a Protocol Description object to filter events by server type in order to avoid confusion when handling these events.

There also may be times when you have several instances of a given server in your environment and you want to filter by a specific one. To do this, first specify an `Endpoint` for that server, using a name for the server in the `Endpoint` constructor:

```
[C#]

string statServer1EndpointName = "StatServer1";
Endpoint statServer1Endpoint =
        new Endpoint(statServer1EndpointName, statServer1Uri);
```

Now create the filter:

```
[C#]

MessageIdFilter statServer1EndpointFilter =
        new MessageIdFilter(EventPackageOpened.MessageId);
```

And set the `EndpointName` property of the filter:

```
[C#]

statServer1EndpointFilter.EndpointName = statServer1EndpointName;
```

When you register this filter, the handler you specify will only receive messages that were sent from the instance you mentioned above:

```
[C#]

eventBrokerService.Register(
        this.OnEventPackageOpened_StatServer1, statServer1EndpointFilter);
```

## Architecture and Design

The Message Broker Application Block is designed to make it easy for your applications to handle events in an efficient way.

Message Broker allows you to set up individual classes to handle specific events coming from Genesys servers. It receives all of the events from the servers you specify, and sends each one 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.

> **Tip**
>
> Message Broker has been designed for use with the Protocol Manager Application Block. Protocol Manager is another high-performance component that makes it easy for your applications to connect to Genesys servers. You can find basic information on how to use the Protocol Manager Application Block in the article on Connecting to a Server Using the Protocol Manager Application Block.

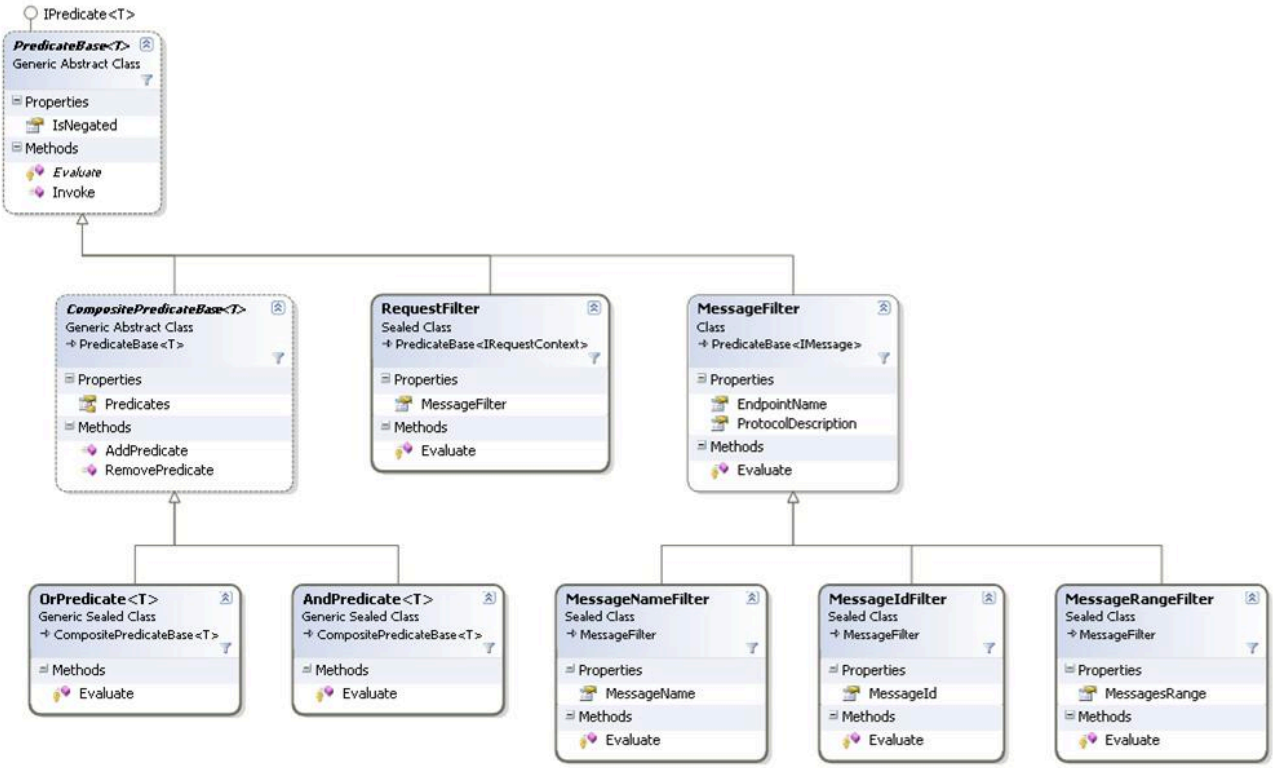
## The Message Broker Application Block Architecture

The Message Broker Application Block uses a service-based API that enables you to write individual methods that handle one or more events.

For example, you might want to handle every occurrence of `EventAgentLogin` with a specific dedicated method, while there might be other events that you wish to send to a common event-handling method. Message Broker allows you write these methods and register them with an event broker that manages them for you.

## Message Filters

Message Broker uses *message filters* to identify specific messages, assign them to specified methods, and route them accordingly. These message filters are shown in greater detail in the figure below.



# Design Patterns

This section gives an overview of the design patterns used in the Message Broker Application Block.

## Publish/Subscribe Pattern

There are many occasions when one class (the subscriber) needs to be notified when something changes in another class (the publisher). Message Broker uses the Publish/Subscribe pattern to inform the client application when events arrive from the server.

## Factory Method Pattern

It is common practice for a class to include constructors that enable clients of the class instantiate it. There are times, however, when a client may need to instantiate one of several different classes. In some of these situations, the client should not need to decide which class is being created. In this case, a Factory Method pattern is used. The Factory Method pattern lets a class developer define the interface for creating an object, while retaining control of which class to instantiate.